An operational semantics of interactions for verifying partially observed executions of distributed systems

Erwan Mahe

université
PARIS-SACLAY

Sémantique opérationnelle des interactions pour la vérification d'exécutions partiellement observées de systèmes distribués

An operational semantics of interactions for verifying partially observed executions of distributed systems

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 573, Interfaces: matériaux, systèmes, usages

Spécialité de doctorat: Informatique

Unité de recherche: Université Paris-Saclay, CentraleSupélec,
Mathématiques et Informatique pour la Complexité et les Systèmes,
91190, Gif sur Yvette, France

Référent: CentraleSupélec

Thèse présentée et soutenue à Gif sur Yvette, le 15 Juillet 2021, par

Erwan MAHE

Composition du Jury

Stefan HAAR Directeur de recherche, INRIA	Président
Gwen SALAUN Professeur, Université Grenoble Alpes	Rapporteur & Examinateur
Xavier URBAIN Professeur, Université Claude Bernard Lyon 1	Rapporteur & Examinateur
Bernhard AICHERNIG Maître de Conférence, Université de Graz	Examinateur
Delphine LONGUET Maître de Conférence, Université Paris-Saclay	Examinateur
Ylies FALCONE Maître de Conférence, Université Grenoble Alpes	Examinateur

Direction de la thèse

Pascale LE GALL Professeur, CentraleSupélec	Directeur de thèse
Christophe GASTON Ingénieur-chercheur, CEA List	Coencadrant

Thèse de doctorat

NNT: 2021UPAST062

Remerciements

Je tiens tout d'abord à remercier ma directrice de thèse Pascale Le Gall pour m'avoir encadré tout au long de cette thèse avec beaucoup de bienveillance et de bonne volonté. Elle a toujours été de très bon conseil et aussi très généreuse tant au niveau de son temps que de ses efforts. Je tiens également à remercier Christophe Gaston, mon coencadrant, pour son amitié, ses conseils et aussi pour m'avoir fait me diriger vers ce projet de thèse après ma formation d'ingénieur.

Merci également à Arnault Lapitre et Boutheina Bannour du CEA, avec qui j'ai pu travailler sur différents sujets liés à cette thèse. Arnault m'a donné beaucoup de bonnes idées afin d'améliorer mes implémentations et Boutheina a su apporter son aide, notamment au niveau de l'état de l'art. Je voudrais également mentionner Mathilde Arnaud, Xavier Zeitoun et plus particulièrement Stéphane Salmons qui m'a encadré pendant mon apprentissage au CEA avant la thèse et m'a permis de faire mes premiers pas dans ce milieu.

Je voudrais aussi remercier l'ensemble des participants au projet DisTA et en particulier Delphine Longuet - qui a intégré mon jury de thèse - et Thomas Vergnaud avec qui j'ai travaillé sur le use case Thalès.

Merci à tous les membres de mon jury pour avoir accepté d'évaluer mes travaux; Stefan Haar, qui a accepté de présider ma soutenance, Bernhard Aichernig, Ylies Falcone, Delphine Longuet ainsi que mes rapporteurs Gwen Salaün et Xavier Urbain que je tiens tout particulièrement à remercier pour avoir pris la peine de lire mon manuscrit et pour m'avoir fait part de leurs retours.

Merci à mes collègues et voisins de bureau au laboratoire MICS: Alexandre Goy que je connais depuis maintenant déjà près de 7 ans et qui est un très bon ami, merci pour les memes, la natation, les randonnées et tout le reste. Une pensée aussi pour Romain Pascual, Salim Nibouche, Adrien Dekkers, Yassine Ouali et les autres membres de l'open space. De l'autre côté du laboratoire je tiens à saluer Brice Hannebique, Mahmoud Bentriou, Antonin Della Noce et tous les autres aussi.

Merci également à Marc Aiguier, Safouan Taha et Jean-Philippe Poli avec qui j'ai participé à diverses missions d'enseignement, ainsi qu'aux autres membres des équipes pédagogiques du MICS. Merci aussi au personnel administratif; Suzanne Thuron qui a été d'une grande aide pour les ré-inscriptions et l'organisation de la soutenance, Sylvie Dervin, Fabienne Brosse et Dany Kouoh Etamè.

Je remercie enfin ma mère, qui m'a élevé avec beaucoup d'abnégation, mon frère, et mes amis de la prépa, Issac, Jennifer, Julia et Tom pour les escape games et autres sorties que j'ai beaucoup apprécié.

Résumé

Les langages d'interactions permettent de modéliser le comportement de systèmes distribués via la spécification des échanges asynchrones de messages qui peuvent se produire entre leurs différents sous-systèmes. Ce type de langage est associé à un formalisme graphique intuitif qui permet une compréhension et une prise en main facile. Dans cette thèse nous formalisons ce type de modèles avec un Langage d'Interactions (LI) défini sous la forme d'une algèbre de termes. Ces termes sont construits à partir d'interactions atomiques qui peuvent correspondre à un comportement vide ou l'expression d'évènements de communication atomiques i.e. émissions ou réceptions de messages. Des opérateurs, incluant le séquencement strict et faible, la composition alternative et parallèle ainsi que quatre boucles distinctes, permettent ensuite la spécification de comportements plus complexes et nuancés par composition.

Nous définissons ensuite une sémantique de trace, associant à chaque terme d'interaction un ensemble de séquences d'évènements qui représentent les comportements pouvant être exprimés par le système distribué qui est modélisé. Cette sémantique est formulée dans un premier temps de manière dénotationnelle. La sémantique d'interactions complexes est définie par composition à partir de celles de ses sous-termes, composées à l'aide d'opérateurs algébriques sur les ensembles de traces. Cette formulation prenant la forme d'un morphisme d'algèbre nous pouvons ensuite utiliser les propriétés algébriques des opérateurs sur les ensembles de traces (associativité, commutativité, etc.) pour obtenir des équations reliant des termes d'interactions ayant la même sémantique. Grâce aux techniques de réécriture nous pouvons ensuite définir des formes normales de terme d'interaction. Dans un deuxième temps, nous proposons une formulation opérationnelle de la sémantique de trace, qui permet de rendre les modèles d'interaction exécutables et ouvre la voie à des applications plus poussées en vérification. Ces deux formulations sont prouvées équivalentes.

Les exécutions d'un système distribué peuvent être observées au travers de logs des évènements de communication collectés localement. Sans horloge globale il n'est pas possible de réordonner ces évènements globalement. Analyser une exécution revient donc à analyser un ensemble de traces qu'on appelle une multi-trace. De plus, sur chaque composante locale, il se peut que l'observation ait commencé trop tard ou ait cessé trop tôt. Ainsi, une multi-trace peut correspondre à une observation partielle d'une exécution. Tirant parti de la formulation opérationnelle ainsi que de propriétés et transformations prouvées par rapport à la formulation dénotationnelle nous proposons des algorithmes d'analyse permettant d'identifier une multi-trace comme étant une observation d'un comportement spécifié par une interaction.

Notre approche a été implémentée au sein d'un outil appelé HIBOU qui permet de spécifier et dessiner des interactions, d'explorer leur sémantiques, de calculer des formes normales ou d'analyser des multi-traces. Nous avons étendu notre LI pour inclure des données sous la forme de variables définies localement. Des gardes, expressions booléennes sur les variables, peuvent conditionner l'exécution d'actions et les messages peuvent porter des données exprimées à l'aide des variables. L'extension aux données a été implémentée en utilisant les techniques d'exécution symbolique. Cet outil étendu a été utilisé pour un cas d'usage industriel dans le cadre du projet FUI DisTA.

Abstract

Interaction languages model the behavior of distributed systems via the specification of the asynchronous passing of messages that can occur between their various sub-systems. This type of language is associated to an intuitive graphical formalism which is easy to understand and manipulate. In this thesis, we define such an Interaction Language (IL), which takes the form of an algebra of terms. Those terms are build from atomic interactions which may consists in the empty behavior or the expression of an atomic communication event (emission or reception of messages). Operators, which include strict and weak sequencing, alternative and parallel composition and four semantically distinct loops allow for the specification of more complex and nuanced behaviors via composition.

We then define a trace semantics, which associates to each interaction term a set of sequences of events which represent the behaviors that can be expressed by the distributed system that is modelled. Firstly, this semantics is formulated in a denotational style. This means the semantics of a complex term is defined by composition from those of its subterms using algebraic operators on sets of traces. This formulation taking the form of a morphism between algebras, we can then use algebraic properties of the operators on sets of traces (associativity, commutativity, etc.) so as to obtain equations linking semantically equivalent interaction terms. Thanks to term rewriting techniques we can then define normal forms of interaction terms. A second formulation of the trace semantics, in operational style, allows for further applications in formal verification thanks to making interaction models executable. Those two formulations are proven equivalent.

During the execution of a distributed system, communication logs might be collected locally. Without a global clock, events cannot be reordered globally. Hence analyzing an execution comes back to analyzing a set of local traces that we call a multi-trace. In addition one might start the observation too late or end it too early on any local component. As a result, a multi-trace might correspond to a partially observed execution. Taking advantage of the operational formulation and of some properties proven using the denotational formulation, we define algorithms for identifying multi-traces as being observations of behaviors specified by an interaction model.

We have implemented our approach in a formal verification tool: HIBOU, which allows the specification and drawing of interactions, the exploration of their semantics, the computation of normal forms or the analysis of multi-traces. The IL can be extended to include data in the form of locally defined variables. Guards, formulated as Boolean expressions on variables can condition the execution of individual actions, while messages can carry data. This extension has also been implemented and we use symbolic execution to animate models and perform the analyses. An industrial case study has been carried out in the context of the DisTA FUI project.

Contents

List of Figures

Chapter 1

Introduction

A Distributed System (DS) is of a collection of independent and concurrently running sub-systems, each of which being a locally determined software system or cyber-physical system. Those sub-systems can communicate with each-other and with the outside of the DS via their communication interfaces, on which they can send and receive messages. The transmission of those messages can be ensured by various mechanisms, which we may abstract as a support network. The transmission of those messages can either be synchronous (the emission and reception events are considered simultaneous) or asynchronous (a delay exists, w.r.t. an abstract global clock, between the emission and the reception events) [77].

The nature of DSs implies a number of issues that are not encountered in classical centralized systems. Among those we can note:

- The reliance upon a support network can cause issues, with messages not being delivered (packet loss), or being delivered in an order that is different (w.r.t. a global ordering of events) to the one in which they were originally emitted (out-of-order delivery). This non-determinism motivated, among other things, the conception of modern communication protocols, which can ensure the correct transmissions of data through a complexification of the exchanges (or not, if the data is deemed not important enough to bother, see UDP vs TCP [40]).

- Moreover, due to their support network, DSs are especially vulnerable ("vulnerable by construction" [108]) to malicious acts (more so than centralized systems). Indeed, the use of third-party networks, public networks (e.g. the internet) or certain technologies (wireless communications) may allow the interception and / or the modification (man-in-the-middle attack [45]) of data that is exchanged between the sub-systems of the DS. As for the previous point, handling those security issues can be done at the price of an increase in the complexity of the exchanges that occur within the DS (handshake protocols, etc.).

- The fact that each sub-system runs concurrently w.r.t. all others inherently creates additional non-determinism and possible interleavings between the events that are expected to occur during the

execution of the DS. Moreover, given that most microprocessors nowadays allow multi-threading, several concurrent behaviors involving multiple sub-systems can be executed simultaneously in the DS (for instance, one can have several instances of different behaviors being executed between a client and a server at the same time). This can be the cause of non-deterministic concurrency bugs, which are often referred to as "Distributed Concurrency (DC) bugs" [80]. Those bugs are notably studied in the context of datacenter distributed systems.

- Finally, the absence of a global clock, shared by all sub-systems of the DS renders impossible, at least without dedicated mechanisms handling clock synchronization (countering clock drift), the synchronization of behaviors between sub-systems. In the perspective of analyzing log files, this is particularly problematic given that it impacts the ability to reorder events [77].

Those particularities makes so that DSs are especially difficult to design, model and verify. On the one hand, one-to-one exchanges between any two sub-systems are increasingly complexified by multi-threading and concerns about reliability and security. On the other hand, DSs may contain very high (and which can vary during the execution) numbers of sub-systems. Those intricacies make so that relying on human intuition alone is unreasonable. Methods for ensuring the quality of DSs should be supported by strong mathematical foundations so as to ensure that they behave as intended and should be fully or at least partly automatizable given the potential scale of such systems which incurs increasingly important costs in terms of human effort.

As a result, the development of sound and automatized processes for improving the design of DSs and for formalizing and mechanizing their verification is relevant, both from an academic perspective given the complexity and relative novelty of the subject and from an industrial perspective given the many various applications in which DSs are found.

Indeed, with the rise of such practices and applications as the Internet of Things (IoT) in which hundreds, thousands or hundreds of thousands of devices may communicate so as to perform various tasks, the concrete and practical applications of DSs in the industry grow ever larger. This shift in the industry towards distributed and decentralized systems is (1) motivated by gains in terms of which services can be rendered and in terms of performances and (2) is fueled by advances in the miniaturization of microchips and the reduction of their costs. In particular, the IoT revolution is enabled by the apparition of affordable and easy to use single-board computers [6, 8] and assimilated hardware. The integration of such connected devices in appliances for the large scale collection of data can be used to implement cost-effective preventive maintenance for company assets. In agriculture, sensor data can be used to predict yields and optimize irrigation and the use of fertilizer and pesticides. In supply-chain and logistics, smart tags can facilitate inventory, help avoid shortages and ensure the smooth operation of a supply chain. Moreover, additional sensors can be used to assess the condition of shipped products (e.g. verifying the cold chain for perishables commodities). Smart objects can also replace everyday household appliances so as to provide ease of life functionalities in domestic use (smart homes etc.) [107]. Groups or swarms of autonomous or semi-autonomous commu-

nicating robots (drones) can perform various tasks with for instance applications in search and rescue in hazardous environments [32]. Another interesting prospect is that of smart grids [108] and smart cities in which city or country wide infrastructures are erected so as to rationalize and streamline the production and consumption of electric energy or vehicular traffic.

1.1 Verification & Validation and Formal Verification

Distributed Systems may contain large numbers of devices; on each such device, various programs can be run at any time; and the network through which those devices communicate may be subject to variations in its reliability. As a result, in a DS, there can be various sources of potential bugs i.e. errors, meaning unwanted behaviors that are executed or the unwanted absence of a behavior.

Ensuring the correct functioning of DSs is all the more important in the case of critical applications (healthcare, the aerospace industry or nuclear energy, etc.) or for applications which impact concerns large numbers of people or large amounts of money (e-commerce, social media, etc.). Bugs found within such systems can have tremendous human or financial consequences. Famous examples include the failure of the Ariane V launcher in flight 501 [82] or the crash of NASA's Mars Climate Orbiter [117].

In order to ensure proper functioning, or at least to guarantee a certain level of quality when it comes to the smooth running of DSs, one can (1) improve their design process through some project management methodologies [117] or (2), from a more technical perspective, one can use techniques to verify and validate the system a posteriori. Some of the contributions of this thesis concerns the latter.

Hence, the detection (and correction) of errors in systems is an important part of the process of developing and maintaining those systems. In the industry, such processes are often described under the umbrella term of Verification and Validation (V&V) [106]. This consists in ensuring that products meet specific requirements i.e. properties, functionalities or behaviors. A set of such functional requirements is called a specification (technical standard of a product). However, V&V is not necessarily a certified process and may rely on human intuition. In order to exclude potential human errors, one has to rely on mathematically sound specifications and methods used to verify those. Merging V&V and formal methods is the object of Formal Verification (FV) [28].

Formal Verification includes a variety of techniques that can programmatically manipulate mathematically sound representations of objects (words in a formal language). In software development, FV methods can be used directly on the application's code, given that programming languages are formal languages. However, it is not always possible to do so (third party, closed-source applications, etc.) or, it may not be desirable to do so, as we may want to use FV on another level of abstraction. Fortunately, FV methods can also be used on models i.e abstract representations of the behavior of a system, as long as those models are written using a formal language (formal models).

Specifications of DSs can be formulated using models instead of natural language. Moreover, if those models are formal models, they can be used programmatically in FV. Uses may notably include the verifi-

cation of past executions of the system. This may consist in verifying that a given execution of a system, abstracted and formalized as an object that may be obtained from parsing and translating log files, belongs to the semantics of a formal model of that system. The distributed nature of DSs makes so that a given execution might produce several distinct log files and, as we have mentioned, a lack of synchronization might prevent merging them. Moreover, it might be so that some log files are incomplete. For instance the logging may have started too late or ended too early. As a result the objects resulting from observing executions and that should be analyzed may correspond to "partial" observations in two aspects: (1) that of lacking means to reorder events globally and (2) that of having missing events.

1.2 Modeling Distributed Systems

In order to apply FV techniques to DSs, we need a formal modelling language for DSs. In the context of this thesis, we focus on behavioral models i.e. models which specify the intended behaviors of the system they represent. With that in mind, one can model DSs in essentially two manners. We can either model each individual sub-systems with a local model and then the global model is some form of a composition of those local models. Or we can directly design a global model, which will not detail the inner workings of local sub-systems but rather focus on the exchanges between those sub-systems. In the domain of service oriented computing, the distinction between those two kinds of approaches is well identified [78], with choreography languages being either "interaction-oriented" such as WS-CDL (Web Service Choreography Description Language) [20] or "process-oriented" such as WS-BPL (Web Service Business Process Execution Language) [19].

Another manner to perceive this distinction is to understand the modelling process as either a "top-down" or "bottom-up" process: "process-oriented" approaches are "bottom-up" given that we start from the local components and then the global behavior is emergent while "interaction-oriented" approaches are "top-down" given that the global behavior is directly modelled. This terminology can be notably found in [46] in the domain of multi-agent systems.

1.2.1 Bottom-up approaches

In bottom-up approaches, each component of the DS is modeled locally. Then, inputs and outputs of local models are connected to the other sub-models (or the environment) with which they communicate through a certain communication medium [28, 53, 35] and a global model is thus constructed "from the bottom to the top" (or one can also describe it as "from the local to the global"). The global behaviors that are specified in this manner emerge from the executions of local models.

The communication medium [28, 53, 35] orchestrates the exchange of messages between local models by associating outputs to corresponding inputs. This association can take the form of one or several buffers with different policies (FIFO, random access, etc.) as explained in [35].

Formalisms used in bottom-up approaches include state-transitions modeling languages such as Finite

State Machine (FSM) [41, 35], Labelled Transition System (LTS) [123] or more complex formalisms with data such as Input Output Symbolic Transition System (IOSTS) [34] and so on. In the Universal Modeling Language (UML) [24] a particular type of diagrams called State Diagrams [102] also corresponds to this type of models.

1.2.2 Top-down approaches

In a top-down approach, the exchanges between the sub-systems of the DS are directly modelled. As in the bottom-up case, those exchanges can be modelled by atomic observable events that consist in either the emission or the reception of a given message on a specific sub-system. However, instead of focusing on the local ordering of events on specific sub-systems, those events are scheduled (ordered with an order relation that may be partial) globally. This global scheduling is to be found in the structure of the model itself instead of being a result of the execution of local models orchestrated by a communication medium.

Formalisms that can be described as top-down include Petri nets and its derivatives such as Coloured Petri Net (CPN) [69] and formalisms derived from Message Sequence Chart (MSC) [22] such as Live Sequence Chart (LSC) [48], which may be referred to as Interaction Languages (ILs). In Universal Modeling Language (UML) [24], Sequence Diagrams are included as an adaptation of MSC into UML.

1.2.3 On the choice of a modeling language

Aspects to consider for the choice of a modeling language may include its expressivity (ability to express detailed behavior), its ease of use (linked to the human effort of the modeling process), ease of understanding (when used as documentation) and ability to be used as input for Formal Verification techniques.

Expressive languages of both types can be found. In terms of usability in FV, "process-oriented" approaches are generally more established and benefit from various ready to use tools with for instance SPIN [16], Estelle [11] or LOTOS [13] for Extended Communicating Finite State Machine (ECFSM), PragmaDev Studio [15], JADE [12], Cinderella [9] or OpenGeode [14] for the SDL language, UPPAAL[17] for networks of timed automata with data or Diversity [10] for sets of communicating Input Output Symbolic Transition System (IOSTS). By contrast, tools for FV based on "interaction-oriented" languages are rarer and they mostly involve Petri nets or rely on translations towards "process-oriented" formalisms (cf. Chap.3).

In terms of ease of use and understanding however, "interaction-oriented" approaches have some advantages. According to [29], the boundary between modeling and implementing is quite blurry; both activities being only differentiated by the level of abstraction and the completeness of the objects they produce. We could argue that with "process-oriented" approaches, more effort is overall required, given that each subsystem must be modeled, and that this effort often overlaps that of implementation given that most systems are implemented in a bottom-up development style. In [78], the authors also argue that "interaction-oriented" approaches support a more abstract (global) vision.

In the context of this thesis we have chosen to focus on Interaction Languages (languages of the family

of MSC) for modelling DSs with asynchronous communication. Those languages represent the behavior of distributed systems in an intuitive graphical format while allowing nuanced and detailed specifications.

1.3 Research questions & position of the thesis

1.3.1 Formalisation of Interaction Languages

On the one hand, the particularities of Distributed Systems are such that classical approaches, originally formulated for the design and verification of centralized systems, even though they are well detailed and come a wide array of tools, cannot be transposed for the design and verification of DSs [83]. The formal specification and verification of DSs requires dedicated formalisms and methods.

On the other hand, Interaction Languages are popular and intuitive languages for the (mostly informal) specification of the behaviors of DSs. Interaction models, represented graphically in the fashion of UML Sequence Diagrams are indeed often found in the documentation of software projects. They allow to synthesize in an intuitive manner the explication of communication schemes between entities that would otherwise require the writing of long documents in natural language. Even though human generally find the understanding of such models relatively easy, it is not the case for computers. This is made evident by (1) the lack of support for model-based design and testing approaches that rely on interaction models and (2) more generally by their scarce use in FV. The few existing approaches to the formalisation and use of ILs in FV (notably tooled approaches) mostly rely on complicated translations towards other formalisms which are often constrained and incomplete. One of the key factors to the difficulty of formalizing and tooling interaction models lies in weak sequencing. Weak sequencing correspond to the top to bottom direction in which one reads an interaction diagram and that represents the elapse of time during execution. Weak sequencing, which is a form of locally defined sequentiality, induces specific partial orders between the occurrences of events that cannot be otherwise specified using classical strict sequentiality and interleaving.

As a result of this, the question of providing a formalisation of an expressive IL that can be used in FV naturally comes in mind. We propose to discuss this issue in the light of the domains of formal languages, term rewriting and process calculus.

1.3.2 Methods for analysing executions of Distributed Systems

Innovative uses of IoT systems in critical infrastructure such as smart grids or in healthcare require a high level of confidence and security which in turn requires the use of formally verified approaches for design and test. The scale and potential complexity of such systems also requires methodologies which are extensible and that support automation so as to afford reasonable costs in terms of human effort.

In particular, one might be interested in offline (i.e. a posteriori) analysis of the executions of such systems, that may be observed via the collection of log files. Whether it be for making sure of the smooth operation of a Distributed System that is already deployed or for testing an implementation during the

process of designing a DS, this kind of analysis can be used to increase the confidence in the conformance of a DS with regards to its specification [58]. In the existing literature, offline analysis is most often limited to verifying local conformance on each sub-system (as in [83] using automata) which can also be completed by methods for checking that each instance of a received message has previously been sent (as in [37], which additionally verifies temporal constraints).

In this thesis, we also propose to address the issue of offline analysis by defining algorithms for analyzing observations of some executions of a DS against a formal specification of that DS written as an interaction model. The nature of the approach that we enquire, based on the automated use of models, falls within the issue of defining automatized, reusable and extensible methods for FV based on Interaction Languages. However, in contrast to the state of the art, we aim at defining an approach that is global by construction and cannot be brought back to a conjunction of local verifications.

1.3.3 Partial observations of distributed executions

In relation to the issue of offline analysis, the observations of DS executions that ought to be analyzed may not be complete. This incompleteness Indeed, without mechanisms to synchronize distant loggers (local testers), it is impossible to ensure that each logger starts and ends logging at the right times. This can result in communication events being missed (i.e. not logged) either at the beginning or the end of any locally defined log file. From a more abstract perspective, this corresponds to having an observation of the execution which is only partial.

The current state of the art regarding the analysis of logs against specifications of DSs mainly revolves around the use of Petri Nets or sets of automata. Moreover those analyses are often constrained by hypotheses on the existence of synchronisation points and partial observation is scantily treated. We propose to address this question through the definition of analysis algorithms that may use either (1) simulation steps so as to "fill-in" the unobserved events by simulating the behavior of sub-systems that were not observed (in a specific span of time) or (2) hiding steps which consists in erasing parts of the formal model against which the analysis is performed once, during the analysis, the sub-systems that are modelled by those parts are not observed anymore.

1.3.4 Position of the thesis

Therefore, an avenue of research exists when it comes to the development of Formal Verification techniques based on formal Interaction Languages. In this thesis we aim at the development of such formal methods. In particular, we are interested in verifying partially observed executions of Distributed Systems with regards to formal specifications written as interaction models. In addition to those theoretical concerns on the formalization of such approaches, the thesis also aims at providing practical and concrete tooled solutions. Indeed, as the thesis has been carried out as part of the DisTA (Distributed Test Automation) FUI project financed by the French ministry of Economy and Finance, we have met with practical challenges that comes

with the realization of use cases and proofs of concepts with industrial partners.

1.4 Outline

This thesis is composed of four parts.

The zeroth part, which includes this introduction, contains introductory material:

- Chap.2 presents contextual information on several domains which relate to the works of this thesis
- Chap.3 gives a quick state of the art on the formal semantics associated to Interaction Languages

The first part is dedicated to the definition of our take on an Interaction Language (IL) and of various trace and multi-trace semantics that can be associated to it:

- in Chap.4 we define the syntax of an IL which is particularly expressive when it comes to the scheduling and repetitions of events. Indeed, this IL contains operators for strict and weak sequencing, interleaving, alternatives, and four semantically distinct loop operators for expressing nuances in the repetition of behaviors. We then associate this formal language with a trace semantics, associating each interaction to a set of traces, which are globally defined sequences of observed events. A first formulation of that semantics is given in denotational style: we use algebraic operators on sets of traces so as to construct the semantics of complex terms by composition of those of simpler sub-terms. This then allows us to define a process to compute normal forms (within a class of semantically equivalent terms) of interactions using ordered and class rewriting.
- in Chap.5 we define a structural operational semantics for our IL. This small-step semantics relies on making interaction models executable, with each step of execution consisting in the expression of a communication action and transforming the interaction into a "follow-up" which specifies what "remains to be executed". This formulation of a trace semantics is inspired by process calculus. We then prove the equivalence of this second semantics with regards to the denotational semantics which serves as a mathematical foundation (Coq proof in [88]). The contribution of this operational semantics and its proof of correctness w.r.t. the denotational is the object of [95] (submitted in April 2021).
- in Chap.6 we redefine the operational semantics in the style of functional programming languages with a twist. That twist consists in separating concerns between the determination of which actions are immediately executable and the actual computation of the "follow-up" interaction to their execution. The first concern is translated into the formulation of a "frontier of execution" while the second is transposed as an "execution function". This "algorithmization" of the operational semantics ("execution semantics") is then declined into variants in which intermediate interactions are simplified on the fly to ensure working with compact terms. One of those variants notably uses the normalization of intermediate interaction terms. We then prove the equivalence of the execution semantics and its

variants w.r.t. the first two semantic (Coq proof in [88]). This verified functional-style semantics serves as a basis for the later tool implementations.

- in Chap.7 we extend the notion of the semantics of interactions from trace semantics to various "multi-trace" semantics, which take into account the distributed nature of observed behaviors and the potentiality for a partial observation of those behaviors.
  - We define an algebraic multi-trace semantics in denotational style using algebraic operators on sets of multi-traces and compare it to a projected multi-trace semantics obtained by projection of the trace semantics. Due to weak sequencing, those two semantics do not generally coincide except in two particular cases. The semantics that is "correct" in intention is the projected multi-trace semantics. However, the algebraic multi-trace semantics benefits from interesting properties due to its definition. We may notably take advantage of this in the cases in which both semantics coincide.
  - Given the nature of multi-traces as sets of sequences the usual notions of prefixes and substrings have to be extended so as to capture the fact that events might be missing at the beginning or end of each local trace component. Accordingly, we define prefixes and slices in the sense of multi-traces and then defined three variants of the projected multi-trace semantics: (1) that of the projections of prefixes of accepted traces, (2) that of prefixes (in the sense of multi-traces) of accepted multi-traces (prefix closure) and (3) that of slices (in the sense of multi-traces) of accepted multi-traces (slice closure). Those semantics cover cases of "partial observation" when collecting multi-traces from observing executions of DSs.

The second part is devoted to the definition of various algorithms for analyzing multi-traces against interaction models, including in case of partial observation:

- in Chap.8 we discuss the problem of analysing multi-traces. This chapter constitutes an introduction to the second part of the thesis.
- in Chap.9 we define a first algorithm for recognizing accepted multi-traces.
  - This algorithm constitutes a solution for the membership problem of the projected multi-trace semantics i.e. determining whether or not a given multi-trace belongs to that semantics w.r.t a given interaction. The underlying problem incidentally corresponds to an instance of the problem of offline analysis i.e. the verification of executions of DSs against formal specifications expressed as interactions. In order to define this algorithm, we take advantage of the operational formulation of the trace semantics of interactions which makes available an "execution relation" which may be used to simultaneously execute an action in an interaction and consume it from the head of a given multi-trace component.
  - We then prove the correctness of this algorithm i.e. that is is indeed a solution to the aforementioned membership problem (Coq proof in [87]).

  - In addition, we take an interest in the complexity class of the membership problem and prove that it is NP-hard.

  Our initial approach to defining such an algorithm, for the analysis of accepted global traces and their prefixes, is described in [93] (published in ETAPS-FASE-2020). In [94] (published in SAC-SVT-2021) we extend this initial algorithm to cover the analysis of accepted multi-traces. In Chap.9 we synthesize and extend both previous approaches.

- in Chap.10 we define the notion of hiding interactions and propose two applications:
  - one for defining a second algorithm which can recognize prefixes (in the sense of multi-traces) of accepted multi-traces. This algorithm uses hiding steps to do so. Hiding steps correspond to erasing parts of the model during the analysis once the sub-systems which those erased parts model are no longer observed. We then prove the correctness of this algorithm.
  - another for improving the efficiency of analysis algorithms via a reduction in their search space. We reduce this search space by cutting some branches which are guaranteed to yield a $Fail$ verdict (i.e. the multi-trace cannot be recognized as accepted when exploring those branches). The determination of those branches to cut is made using a notion of local frontier which is that of which actions can be executed next on a specific sub-system.

In the wake of those theoretical results, we present extensions of the formalisms and the implementation of tools in the third part of the thesis:

- in Chap.11 we present immediate extensions which include:
  - the definition of additional (unproven) algorithms which make use of simulation steps to recognize prefixes and slices of accepted multi-traces. Simulation steps correspond to simulating the behavior of sub-systems which are not observed (not yet observed or not observed anymore) by executing relevant parts of the model.
  - the addition of a co-region construct to facilitate the design of interaction specifications
  - a discussion on the ease of implementation of the results of this thesis and the presentation of a process by which to implement the analysis algorithms which we have presented in an efficient and configurable manner
- in Chap.12 we present the first tool HIBOU (for Holistic Interaction Behavioral Oracle Utility) which is a direct implementation of most theoretical results from the first two parts of the thesis. In addition, HIBOU proposes:
  - a user friendly interface (Command Line Interface and entry language)
  - features for graphically representing interactions, the computation of their normal forms, their semantics and graphs relating to analyses of multi-traces

  - various heuristics dedicated to the smooth exploration of execution trees (for computing semantics) and analysis graphs (for analysing multi-traces)

- in Chap.13 we motivate and introduce an extension of our IL and FV framework to treat interaction models enriched with data. This takes the form of (1) having lifelines being associated to typed variables which can be created and modified at runtime, (2) having the execution of communication actions being preceded or followed by the execution of statements consisting in variable assignment and guards which constrain the current values of variables and condition the execution of said actions and (3) having messages being a more precise abstraction of exchanged information with the addition of typed message parameters. We also discuss extending multi-trace analysis to multi-traces with concrete data as parameters of messages.

- finally, in Chap.14 we present the second tool HIBOUX (for HIBOU with symbolic eXecution) which implements that which we have discussed in the previous chapter. HIBOUX extends the functionalities of HIBOU to treat interaction models enriched with data and uses symbolic execution to execute and animate those enriched models and to analyse multi-traces with typed message parameters. We then present an industrial use-case, provided by Thales, on which HIBOUX was used as part of the DisTA FUI project.

Chapter 2

Context

Contents

In this chapter, we provide elements of context for some fields of study which relate to the contributions of this thesis:

- in Sec.2.1 we provide a quick and shallow definition of formal languages and their semantics. Indeed, in the context of this thesis, we aim at the formalisation of some specific modelling language.

- in Sec.2.2 we introduce notions related to term algebras, equational logic and term rewriting which prove helpful for the definition of formal languages and their semantics.

- in Sec.2.3 we give a quick overview of Interaction Languages

- in Sec.2.4 we present the basics of process calculi, which is the study of the behaviour of parallel or distributed systems by algebraic means.

- in Sec.2.5 we introduce Coq, a software tool for machine-checked theorem proving and we illustrate its use on the example process algebra from Sec.2.4.

2.1 Formal Languages and Formal Semantics

Languages are used to communicate information. Different languages may share many similarities that may be abstracted so as to define the nature of a language. This need of abstraction and formalization leads, with contributions and exchanges with the fields of linguistics, mathematics and then computer science, to the definition and hierarchisation of "formal languages" ([116] p.1-9).

2.1.1 Formal languages

In all generality, a formal language L is a subset of the words that can be formed from a given alphabet Σ (not necessarily finite). A word is a finite sequence of symbols or "letters" taken from the alphabet Σ that can be characterized as the concatenation of its constituent letters. The set of all possible words that can be obtained in this manner is noted Σ^* and called the "free monoid". The use of the the "$*$" symbol is associated with the "Kleene Star" operator, which roughly means zero or more repetitions, but finitely many ([116] p.10-12).

Definition 2.1: Free Monoids and words

For any set (alphabet) Σ the free monoid Σ^* generated by Σ contains words on Σ and is defined by:

- $\epsilon \in \Sigma^*$ is the empty word
- $\Sigma \subset \Sigma^*$ i.e. each individual letter as a word
- for any words u and v in Σ^*, the word denoted by $u.v$ or more simply uv is also in Σ^*

The "." concatenation operator that is thus induced is such that:

- it admits ϵ as a neutral element
- it is associative

For any words u and v in Σ^* we say that:

- u is a prefix of v and we denote it by $u \leq v$ iff there exists $w \in \Sigma^*$ such that $v = u.w$
- u is a subword of v iff there exists w and w' such that $v = w.u.w'$

For any set V of words, we say that V is closed by prefixes if it contains all the prefixes of all its elements.

Not all those words are necessarily "well-formed" w.r.t. the formal language. In the definition of a formal language, the manner in which those words are selected is not given. Rules for the systematic generation (e.g. by a computer) of all well-formed words from an alphabet are usually given by a "formal grammar" [116] which consists of sets of symbols and production rules which can be used to create increasingly complex words.

Formal grammars can be categorized using the Chomsky hierarchy [116] according to their expressivity. Type-3 grammars, which can generate regular languages are the simplest while type-0 grammars can generate the more complex recursively enumerable languages. In the context of this thesis, we will be only interested in type-2 "context-free" grammars.

2.1.2 The semantics of formal languages

The grammars of formal languages describe the *syntax* of words i.e. which words are or are not well-formed. By contrast, the study of the meaning of words and languages relates to their *semantics*. This is particularly relevant in the context of programming languages and modelling languages, which are particular kinds of formal languages.

Programs that can be written in a programming language are well-formed words of that language. However the outputs that can be produced by the program once compiled and executed on a computer are intractably complex and cannot generally be inferred from its code. We can define the formal semantics of

a programming language so as to understand, describe and explain the complex relationships that can exist between its code, its inputs and its outputs [125, 72, 64].

As illustrated on Fig.2.1, a semantics of a language can be defined by mapping objects of the syntactic domain (the *syntax*) into objects of a semantic domain. The definition of what constitutes a *semantic domain* may vary. For instance, in the case of programming languages (the syntactic domain being that of well-formed programs) it may consist in sequences of strings observed of the standard output of a program.

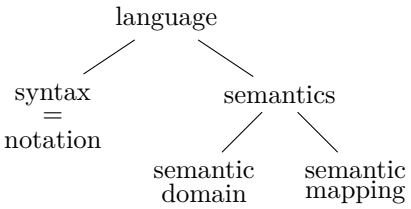


Figure 2.1: The structure of a language

There is a variety of different manners to define formal semantics. [72] proposes the following categories:

- "translational semantics" ([72] p.187-222), which consist in providing a semantics for a high-level language via a translation towards a lower level language which is already equipped with a semantics.

- "operational semantics" ([72] p.223-270), which rely on defining a manner to execute the words of the language. In particular, "small-step operational semantics" also called "structural operational semantics" ([72] p.238-261) explicit atomic computation steps that constitute the execution of a word.

- "denotational semantics" ([72] p.271-340), which map concepts of the language into some mathematical objects (numbers, sets, tuples, functions, etc.). The resulting mathematical object is called a "denotation". Denotational semantics are compositional. They infer the meaning of a word by building denotations recursively based on the structure of the language's words. By that is meant that the denotation of a word is constructed out of the denotation of its "sub-words".

- various other types of semantics can be described such as "fixed-point semantics" ([72] p.341-394) or "axiomatic semantics" ([72] p.395-442) but those are outside the scope of this thesis.

When it comes to modelling languages, the question of what constitute their semantics do not have an obvious answer, as explained in [64]. Indeed, there is a wide variety of modelling languages that can be used to express many different concepts; for instance a structural model has a purpose which is entirely different from that of a behavioral model. Nevertheless, we can roughly use the same categorisation of semantics when it comes to describing modelling languages and their semantics. In particular, in the context of this thesis (1) we only consider behavioral models i.e. models which specify the intended behaviors of the system they represent and (2) we are mainly interested in denotational and operational semantics while various translational semantics are found in the related state of the art.

2.2 Equational logic & rewriting

Words of a formal language, and more particularly in the case of context-free languages, can be understood as terms built by composition from basic building blocks and using operators. This compositionality in the structure of terms may then allow the definition of denotational semantics as morphisms between the syntactic domain of terms and a given semantic domain. Considering terms which have the same semantics and reasoning on syntactic transformations that preserve said semantics can then lead to the definition of classes of equivalence and normal forms of terms.

In this section we present term algebras, equational logic and term rewriting which will enable us to produce such results for our Interaction Language (IL).

2.2.1 Term algebras

In this section we present term algebras as a means to formalize this understanding of words of a language as terms.

In Def.2.2 we define the notion of operation symbols and how they can constitute "signatures" of term algebras which we then define in Def.2.3.

Definition 2.2: Signature

A signature $\mathcal{F}$ is a set of operation symbols: $\mathcal{F} = \bigcup_{j\geq 0} \mathcal{F}_j$ such that for any integer $j \geq 0$, the set $\mathcal{F}_j$ is that of symbols of arity j. For any such symbol f, we denote by $|f|$ its arity. Symbols of arity 0 are called constants.

An operation symbol may correspond to an operator in a mathematical formula. For instance the addition "+" operator or the conjunction "$\wedge$" operator could be defined as operation symbols of arity 2.

Operation symbols of arity 0 correspond to the basic building blocks which we mentioned earlier. Continuing on the example of mathematical formulae, the constant integer 1 or the constant truth value $\top$ could be defined as operation symbols of arity 0.

Definition 2.3: Term Algebras

Given a signature $\mathcal{F}$ and a set $\mathcal{X}$ of variables that is disjoint from $\mathcal{F}$, the term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is the smallest subset of $(\{(,)\} \cup \mathcal{F} \cup \mathcal{X})^*$ such that:

- $\mathcal{X} \cup \mathcal{F}_0 \subset \mathcal{T}_{\mathcal{F}}(\mathcal{X})$
- for any symbol $f \in \mathcal{F}$ of arity $n > 0$ and for any terms $t_1, \cdots, t_n$ from $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$, we have that $f(t_1, \cdots, t_n) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$

The ground term algebra $\mathcal{T}_{\mathcal{F}}$ can be defined as $\mathcal{T}_{\mathcal{F}}(\emptyset)$ and contains all terms without variables.

This manner of constructing elements of a term algebra, called first-order terms [71], can be described as a definition by closure. Indeed, variables and constants form basic building-blocks and symbols of higher arity can be used as construction rules for the formulation of more complex elements.

This inductive construction also allows us to define functions and to make proofs using structural induction [25]. The principle of a proof by structural induction is that once a property is proven for all base elements and, if this property is conserved by the application of the construction rules corresponding to symbols of arity $n > 0$, then it is proven for all terms.

Variables (via the set $\mathcal{X}$ of variables) can be used in the construction of a term so as to abstract some non-determined sub-term. We can draw a distinction between terms without variables, which are called ground terms, and terms in which there are variables. We may use the notation $var(t)$ to designate the set of variables that occur on a given term t, as defined in Def.2.4.

Definition 2.4: Variables of a term

For any term $t \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$, we denote by $var(t) \in \mathcal{P}(\mathcal{X})$ its set of variables defined by:

- if $t \in \mathcal{X}$ then $var(t) = \{t\}$
- if $t \in \mathcal{F}_0$ then $var(t) = \emptyset$
- if $t = f(t_1, \cdots, t_n)$ with $n > 0$ and $f \in \mathcal{F}_n$ then $var(t) = \bigcup_{j \in [1,n]} var(t_j)$

Another result stemming from the domain of term algebras is that of describing terms as trees. This understanding of terms as trees, which nodes are uniquely associated to positions [71, 51], allows the description of operations on those terms. In Def.2.5 we define the notion of tree (as understood in the context of term algebras [71]) and positions.

In Def.2.5, we define, for any term $t \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$, the set $pos(t)$ as the set of its positions. As in [51, 71], we denote by $t(p)$ the node at position p and by $t_{|p}$ the sub-term at position p. The definition of positions as words on integers is sometimes referred to as the Dewey Decimal Notation [51, 43].

Definition 2.5: Trees and positions

Given a signature $\mathcal{F}$ and a set $\mathcal{X}$ of variables, a tree on $\mathcal{F} \cup \mathcal{X}$ is a partial application $t : \mathbb{N}_+^* \to \mathcal{F} \cup \mathcal{X}$ defined over a finite domain $pos(t)$ of $\mathbb{N}_+^*$ such that:

- $pos(t)$ is closed by prefixes
- for any $p \in pos(t)$ and for any $j \in \mathbb{N}_+$, we have $p.j \in pos(t)$ iff $t(p) = f \in \mathcal{F}$ and $1 \leq j \leq |f|$

Let us remark that, in a ground term t (i.e. without variables), nodes $t(p)$ at leaf positions hosts constants (elements of $\mathcal{F}_0$) while inner nodes hosts operation symbols of higher arity [51]. In all generality one could define infinite trees. However, in Def.2.5 we have required that they be defined on a finite domain of positions

and hence are finite trees. The set of finite trees on $\mathcal{F} \cup \mathcal{X}$ is isomorph w.r.t. the set of terms $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$. As a result we may describe freely elements of a term algebra as either terms or trees.

In Def.2.6 we define the notion of term replacement, with the notation $t[s]_p$ being used to indicate the term that results from replacing the sub-term $t_{|p}$ at position p within t by s.

Definition 2.6: Term replacement

For any $(t, s) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$, for any $p \in pos(t)$, the term $t[s]_p$ is defined inductively by:

- $t[s]_\epsilon = s$
- $f(t_{|1}, \cdots, t_{|n})[s]_{j.p} = f(t_{|1}, \cdots, t_{|j}[s]_p, \cdots, t_{|n})$ for any $n > 0$, $j \in [1, n]$ and $f \in \mathcal{F}_n$

2.2.2 $\mathcal{F}$-algebras

In the same manner as a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ can be used to describe the syntactic domain of a language, we can define an algebra to represent the semantic domain of said language. In this section we define $\mathcal{F}$-algebras for that purpose.

As explained in [51, 25], given a signature $\mathcal{F}$ of operation symbols, a $\mathcal{F}$-algebra consists of (1) a non-empty set $\mathbf{A}$ which is a "domain of values" that we may call the "carrier" of the algebra, and (2) a family $\mathcal{F}^{\mathcal{A}}$ (indexed by $\mathcal{F}$) of operations on $\mathbf{A}$. We formalize this definition in Def.2.7.

Definition 2.7: $\mathcal{F}$-algebras

An $\mathcal{F}$-algebra is a structure $\mathcal{A} = (\mathbf{A}, \mathcal{F}^{\mathcal{A}})$ where:

- $\mathbf{A}$ is a non-empty set called the carrier of the algebra
- $\mathcal{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathcal{F}\}$ such that for each operation symbol $f \in \mathcal{F}$:
  - if $|f| = 0$ then $f^{\mathcal{A}}$ is an element of $\mathbf{A}$
  - if $|f| = n > 0$ then $f^{\mathcal{A}}$ is a total function $f^{\mathcal{A}} : \mathbf{A}^n \to \mathbf{A}$

A purpose of $\mathcal{F}$-algebras, as explained in [51], is to attach meaning to terms of $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$. This can be done through the definition of "environments" [25] which map terms $t \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ to values in the domain of values $\mathbf{A}$ (carrier of the $\mathcal{F}$-algebra) thanks to the operations from $\mathcal{F}^{\mathcal{A}}$. We formally define environments in Def.2.8.

Definition 2.8: Environments

Given a mapping $\rho : \mathcal{X} \to \mathbf{A}$, we denote by $\overline{\rho} : \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \to \mathbf{A}$ and we call an "environment" the unique homomorphism that extends ρ such that:

$$\forall\ f \in \mathcal{F},\ \text{of arity } n,\ \overline{\rho}(f(t_1, \cdots, t_n)) = f^{\mathcal{A}}(\overline{\rho}(t_1), \cdots, \overline{\rho}(t_n))$$

The semantic mapping evoked in Sec.2.1 that relates words to their meaning can then be defined as an "environment" as defined in Def.2.8. More particularly, in the case of ground terms, we can define the semantics of elements of a ground term algebra $\mathcal{T}_{\mathcal{F}}$ as the unique such "environment" between $\mathcal{T}_{\mathcal{F}}$ and $\mathcal{A}$. This unique homomorphism is sometimes described as the initial homormorphism or the homomorphism between the initial algebra and the free algebra [74] associated to $\mathcal{F}$.

2.2.3 Preliminaries on binary relations

Binary relations relate terms of a term algebra. For instance we might be interested in defining a relation on terms that are not necessarily syntactically equal but that have the same semantics. In any case, the purpose of this section is to introduce binary relations and their properties.

Definition 2.9: Binary relations

A binary relation $\mathcal{R}$ on a set A is a subset of $A \times A$. We may use the infix notation $x\mathcal{R}y$ to denote elements $(x, y) \in \mathcal{R}$.

For any two binary relations $\mathcal{R}_1$ and $\mathcal{R}_2$ on the same set A, we may denote by $\mathcal{R}_1 \circ \mathcal{R}_2$ the composition of both relations such that $\mathcal{R}_1 \circ \mathcal{R}_2 = \{(x, z) \in A \times A \mid \exists\ y \in A,\ x\mathcal{R}_1 y \wedge y\mathcal{R}_2 z\}$.

Also, given that binary relations are defined as sets, we may say that a relation is included in another and use classical set theoretic notations $\subset$, $\subseteq$, $\subsetneq$ and $\not\subseteq$ to describe relations between binary relations.

In Def.2.10 we summarize basic properties that may describe specific binary relations.

Definition 2.10: Basic properties of binary relations

A binary relation $\mathcal{R}$ on a set A is said to be:

- symmetric if for any $(x, y) \in A^2$ we have that $x\mathcal{R}y$ implies $y\mathcal{R}x$
- antisymmetric if for any $(x, y) \in A^2$ if we have both $x\mathcal{R}y$ and $y\mathcal{R}x$ then this implies that $x = y$
- reflexive if for any $x \in A$ we have $x\mathcal{R}x$
- antireflexive if for any $x \in A$ we do not have $x\mathcal{R}x$
- transitive if for any $(x, y, z) \in A^3$ if we have both $x\mathcal{R}y$ and $y\mathcal{R}z$ then this implies that $x\mathcal{R}z$

Binary relations that satisfy specific combinations of the properties from Def.2.10 may be described by specific terms, some of which are given in Def.2.11.

Definition 2.11: Specific kinds of binary relations

A binary relation $\mathcal{R}$ on a set A is said to be:

- a quasiorder if it is reflexive and transitive
- an order if it is reflexive, transitive and antisymmetric
- a strict order if it is antireflexive, transitive and antisymmetric
- an equivalence relation if it is reflexive, symmetric and transitive

As implied by Def.2.11 the notion of orders can be described by binary relations. We may use the notation $\geq$ to describe a quasiorder. For any such quasiorder $\geq$ on a set A, and for any two elements x and y of A we may equally denote by $y \leq x$ the fact that $x \geq y$. Likewise we may use the notations $>$ and $<$ to describe a strict order.

In addition, quasiorders, orders and strict orders can be described as being either total or partial. For any order $>$ on a set A, $>$ is said to be total if, for any two distinct elements x and y of A we have either $x > y$ or $x < y$. An order is partial if it is not total. For a partial order $>$ on a set A, we may denote by $x \not< y$ the fact that we do not have $x < y$, which, given that the order is partial do not necessarily equates having $x > y$.

A strict order $>$ on a set A is said to be well-founded or noetherian if there exists no infinite descending chain $x_1 > x_2 > \cdots$ of elements of A. In other words this equates to having the existence (but not necessarily the unicity) of a minimal element for any non-empty subset B of A i.e. $\forall\, B \subseteq A$, $(B \neq \emptyset) \Rightarrow (\exists\, m \in B,\, \forall\, x \in B, x \not< m)$.

We may use the notation $\sim$ to designate an equivalence relation. An equivalence relation $\sim$ on a set A defines a partition of A into disjoint equivalences classes. For any $x \in A$, x belongs to a certain class which may be denoted by $[x]_\sim$ and defined as $\{y \in A \mid x \sim y\}$. The set of all equivalence classes thus defined is called the quotient set of A by $\sim$ which we may denote by $A/\sim = \{[x]_\sim \mid x \in A\}$.

Definition 2.12: Equivalence classes

Given $\sim$ an equivalence relation on A we denote by:

- $[x]_\sim = \{y \in A \mid x \sim y\}$ the equivalence class of an element $x \in A$
- $A/\sim = \{[x]_\sim \mid x \in A\}$ the set of equivalence classes thus defined

Binary relations can be canonically extended into larger relations which may satisfy the properties from Def.2.10. We define those larger relations, called closures, in Def.2.13.

Definition 2.13: Closures of binary relations

Let us consider a binary relation $\rightarrow$ on a set A. We may then denote by:

- $\leftarrow$ its inverse relation i.e. $\leftarrow = \{(y,x) \mid (x,y) \in \rightarrow\}$
- $\leftrightarrow$ its symmetric closure i.e. the smallest symmetric relation s.t. $\rightarrow \subseteq \leftrightarrow$
  which exactly is $\rightarrow \cup \leftarrow$
- $\overset{0}{\rightarrow}$ the identity relation on A and $\overset{1}{\rightarrow} = \rightarrow$ and, for any $n > 0$, $\overset{n+1}{\rightarrow} = \overset{n}{\rightarrow} \circ \rightarrow$
- $\overset{+}{\rightarrow}$ its transitive closure i.e. the smallest transitive relation s.t. $\rightarrow \subseteq \overset{+}{\rightarrow}$
  which exactly is $\bigcup_{n=1}^{\infty} \overset{n}{\rightarrow}$
- $\overset{*}{\rightarrow}$ its reflexive and transitive closure i.e. the smallest reflexive and transitive relation s.t. $\rightarrow \subseteq \overset{*}{\rightarrow}$
  which exactly is $\bigcup_{n=0}^{\infty} \overset{n}{\rightarrow}$
- $\overset{*}{\leftrightarrow}$ its symmetric, reflexive and transitive closure (which always is an equivalence relation)

Binary relations can be defined on term algebras. For a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ defined w.r.t. a signature $\mathcal{F}$, we say that a binary relation $\mathcal{R}$ on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is $\mathcal{F}$-compatible if it satisfies the definition from Def.2.14. A $\mathcal{F}$-compatible binary relation is compatible with the algebraic structure of the term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$.

Definition 2.14: $\mathcal{F}$-compatible binary relation

A binary relation $\mathcal{R}$ on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is $\mathcal{F}$-compatible if for any operation symbol $f \in \mathcal{F}_n$ of arity $n > 0$ and for any terms $x_1, \cdots, x_n$ and $y_1, \cdots, y_n$ we have:

$$(\forall\ j \in [1,n],\ x_j \mathcal{R} y_j) \Rightarrow (f(x_1, \cdots, x_n) \mathcal{R} f(y_1, \cdots, y_n))$$

In the more general context of abstract algebra, an equivalence relation that is defined on and compatible with an algebraic structure is said to be a congruence relation. In the particular case of $\mathcal{F}$-algebras, a $\mathcal{F}$-compatible equivalence relation on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is said to be a congruence relation on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$.

As per [25, 51], a substitution is a mapping from variables (i.e. $\mathcal{X}$) to terms (i.e. $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$) that is extended to terms i.e. as an endomorphism of $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$. In practice, a substitution operates the systematic replacement of specific variables (all occurrences of those variables) within a term.

Definition 2.15: Substitutions

A mapping $\phi : \mathcal{X} \to \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is extended to terms as a substitution $\phi^\dagger$ s.t. for any term t:

- if $t \in \mathcal{X}$ then $\phi^\dagger(t) = \phi(t)$
- if $t \in \mathcal{F}_0$ then $\phi^\dagger(t) = t$
- if $t = f(t_1, \cdots, t_n)$ with $f \in \mathcal{F}_n$ given $n > 0$ then $\phi^\dagger(t) = f(\phi^\dagger(t_1), \cdots, \phi^\dagger(t_n))$

We may abusively denote by ϕ either the initial mapping ϕ or the substitution $\phi^\dagger$.
We denote by $\mathrm{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X}))$ the set of all substitutions on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$.

We call the domain of a substitution ϕ the set $Dom(\phi) = \{x \in \mathcal{X} \mid \phi(x) \neq x\}$ which includes variables that are effectively substituted when applying the substitution. A substitution might not replace all or any variables at all; for instance the identity can be described as the substitution on the empty domain. We can also remark that substitutions replace all occurrences of all variables of their domain at once. As a result we may denote a substitution using its domain: for instance, given ϕ such that $Dom(\phi) = \{x_1, x_2\}$, $\phi(x_1) = t_1$ and $\phi(x_2) = t_2$, we might denote ϕ by $\langle x_1 \mapsto t_1, x_2 \mapsto t_2 \rangle$.

Also, substitutions do not generally commute. For instance $\langle x_1 \mapsto t_1, \cdots, x_n \mapsto t_n \rangle$ is not necessarily equivalent to the composition of all $\langle x_j \mapsto t_j \rangle$. This is due to the fact that all variables are replaced at once and the fact that applying a substitution $\langle x_j \mapsto t_j \rangle$ might introduce new instances of variables $x_{j'}$ $(j' \neq j)$.

Binary relations on terms can also be said to be stable under substitution, as defined on Def.2.16.

Definition 2.16: Binary relations on terms that are stable under substitution

A binary relation $\mathcal{R}$ on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is stable under substitution if for any substitution $\phi \in \mathrm{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X}))$ and for any terms $x, y \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ we have:

$$x \mathcal{R} y \Rightarrow \phi(x) \mathcal{R} \phi(y)$$

In Def.2.13 we have seen that basic properties of binary relations (Def.2.10) can be used to define larger relations by closure. This can also be applied for the properties of being $\mathcal{F}$-compatible and stable under substitution. For any binary relation $\mathcal{R}$ on terms of $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$:

- the closure under substitution of $\mathcal{R}$ is the smallest relation that is stable under substitution and which includes $\mathcal{R}$
- the congruence closure of $\mathcal{R}$ is the smallest congruence relation which includes $\mathcal{R}$

2.2.4 Equational logic

In this section we present how we can relate syntactically distinct terms of a term algebra. In particular we can apply those results so as to define equivalence classes of syntactically distinct interaction terms which

all have the same semantics (as per the association to a given $\mathcal{F}$-algebra).

For any two terms l and r from $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$, we denote by $l \approx r$ an "equation" or "equational axiom" in which l might be called the "left-hand side" and r may be called the "right-hand side". The $l \approx r$ equation is a predicate that we may suppose in a certain context and is not equivalent to the syntaxic equality "$l = r$".

Definition 2.17: Axiom systems

Given a signature $\mathcal{F}$ and a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$, an equation is a pair $(l, r) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \times \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ that we may denote by $l \approx r$.
A set of equations $E \subset \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \times \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is an "axiom system".

Given an axiom system E (defined in Def.2.17), an equation $x \approx y$ is deducible from E, which we denote by $E \vdash x \approx y$ iff $x \approx y$ can be obtained from successive transformations of equations from E using the six deduction rules that constitute equational logic [68] and that are given in Def.2.18.

Definition 2.18: Equational Logic

Considering an axiom system $E \subset \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \times \mathcal{T}_{\mathcal{F}}(\mathcal{X})$, we define the relation $E \vdash$ of deducibility or provability modulo E as follows:

$$\forall\ (x \approx y) \in E, \qquad \frac{}{E \vdash (x \approx y)}$$

$$\forall\ x \in \mathcal{T}_{\mathcal{F}}(\mathcal{X}), \qquad \frac{}{E \vdash (x \approx x)}$$

$$\forall\ (x, y) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2, \qquad \frac{E \vdash x \approx y}{E \vdash y \approx x}$$

$$\forall\ (x, y, z) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^3, \qquad \frac{E \vdash x \approx y \qquad E \vdash y \approx z}{E \vdash x \approx z}$$

$$\forall\ (x, y) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2, \forall\ \phi \in \mathrm{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X})), \qquad \frac{E \vdash x \approx y}{E \vdash \phi(x) \approx \phi(y)}$$

$$\forall\ f \in \mathcal{F}_j, \forall\ (x_k, y_k)_{k \in [1,j]} \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^{2j}, \qquad \frac{E \vdash x_1 \approx y_1 \qquad \cdots \qquad E \vdash x_j \approx y_j}{E \vdash f(x_1, \cdots, x_j) \approx f(y_1, \cdots, y_j)}$$

For any axiom system E, we may denote by $\approx_E$ the relation such that for any terms x and y, we have $E \vdash x \approx_E y$ i.e. such that x and y are provably equal in the theory specified by E [68] (using the aforementioned rules of equational logic). The reader may have remarked that this means that $\approx_E$ is the reflexive, symmetric, transitive, $\mathcal{F}$-compatible and stable by substitution closure of the relation E. Indeed, in the rules from Def.2.18 the 2nd corresponds to reflexivity, the 3rd to symmetry, the 4th to transitivity, the 5th to being stable by substitution and the 6th to being $\mathcal{F}$-compatible.

2.2.5 Equational bases

As we have seen in Sec.2.2.2, $\mathcal{F}$-algebras can be defined so as to attach meaning [51] to terms of a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$. As a result, we have, on the one hand a description of the meaning of terms through $\mathcal{F}$-algebras (semantic domain) and, on the other hand a description of relations between terms (syntax) with equational

logic. In this section we present how we can bridge the gap between those two kinds of descriptions with the study of equational bases [25].

The interpretation of terms from a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ by a $\mathcal{F}$-algebra $\mathcal{A}$ induces a congruence relation denoted by $=_{\mathcal{A}}$ [25] on $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$, which we define in Def.2.19. Indeed, we may say that $x =_{\mathcal{A}} y$ whenever $\mathcal{A}$ attaches the same meaning to both x and y. x and y can then be said to be semantically equal [68] which means that we have $\overline{\rho}(x) = \overline{\rho}(y)$ for every environment $\overline{\rho}$ (i.e. whichever values are given to the variables from $\mathcal{X}$).

Definition 2.19: Congruence relation induced by a $\mathcal{F}$-algebra

Let us consider a signature $\mathcal{F}$, a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ and a $\mathcal{F}$-algebra $\mathcal{A} = (\mathbf{A}, \mathcal{F}^{\mathcal{A}})$.
The relation $=_{\mathcal{A}} \subseteq \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \times \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is defined by:

$$\forall\ (x, y) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2, \quad (x =_{\mathcal{A}} y) \ \Leftrightarrow\ (\forall\ \rho \in A^{\mathcal{X}},\ \overline{\rho}(x) = \overline{\rho}(y))$$

Let us remark that in the case of ground algebras $\mathcal{T}_{\mathcal{F}}$ the definition of $=_{\mathcal{A}}$ is straightforward given that there exists a single homomorphism between $\mathcal{T}_{\mathcal{F}}$ and $\mathcal{A}$.

As explained in [25], it is sometimes (but not necessarily) possible to offer a syntactic characterization of a relation $=_{\mathcal{A}}$ via the definition of an axiom system E which may serve as an equational base of $=_{\mathcal{A}}$ i.e. such that $\approx_E$ is equal to $=_{\mathcal{A}}$. In Def.2.19 we explain how a relation defined on the syntax $\approx_E$ can be characterized w.r.t. a relation defined on the semantics $=_{\mathcal{A}}$.

If an equational base E of a relation $=_{\mathcal{A}}$ is found then, issues related to semantic equality can be entirely brought back to reasoning on the syntax. Indeed, as explained in [68], the validity problem $x =_{\mathcal{A}} y$ i.e. whether or not two terms have the same semantics, can be brought back to the question of whether or not $x \approx_E y$ holds, which can be determined via the application of equational logic (and more particularly term rewriting [51], as we will see in the following).

Definition 2.20: Sound & complete axiom systems

Let us consider a signature $\mathcal{F}$, a term algebra $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ and a $\mathcal{F}$-algebra $\mathcal{A} = (A, \mathcal{F}^{\mathcal{A}})$.
An axiom system $E \subseteq \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \times \mathcal{T}_{\mathcal{F}}(\mathcal{X})$ is said to be:

- sound w.r.t. $\mathcal{A}$ iff $\approx_E \subseteq =_{\mathcal{A}}$
- complete w.r.t. $\mathcal{A}$ iff $=_{\mathcal{A}} \subseteq \approx_E$
- an equational base of $\mathcal{A}$ iff $\approx_E = =_{\mathcal{A}}$

Determining whether or not an axiom system E is an equational base for a $\mathcal{F}$-algebra $\mathcal{A}$ i.e. for the relation $=_{\mathcal{A}}$ can be separated into two problems as per Def.2.20:

1. determining whether or not E is sound w.r.t. $\mathcal{A}$.

2. determining whether or not E is complete w.r.t. $\mathcal{A}$.

The problem of proving that an axiom system is sound w.r.t. a $\mathcal{F}$-algebra $\mathcal{A}$ is most often simple. However, proving that it is complete is more problematic. Results and open questions relating to this problem are notably discussed in [25]. In the context of this thesis however, we will simply be interested the first problem.

2.2.6 Basics of term rewriting

Equational logic allows us to navigate within the congruence classes $[\,]_{\approx_E}$ of a term algebra via the succession of atomic transformations in which a certain equation from E may be applied in a certain direction, at a certain position, and modulo a certain substitution.

This can be quite useful, as it allows for instance to determine whether or not two terms $(t_\alpha, t_\beta) \in \mathcal{T}_\mathcal{F}(\mathcal{X})^2$ are related by a congruence relation $\approx_E$. Indeed this then equates to finding a path $t_\alpha \approx_E t_1 \approx_E t_2 \approx_E \cdots \approx_E t_n \approx_E t_\beta$ in which each step consists of an atomic transformation derived from E as described above.

However, finding such a path requires knowing which transformations should be applied at which positions and in which order. A brute force exploration of all sequences of transformations is out of the question due to the exponential complexity costs. On specific example, the use of human intuition is possible, however automation is possible with term rewriting, which is the object of this section.

One of the main purposes of term rewriting is to automatize the process of finding paths $t_1 \approx \cdots \approx t_n$ between related terms, and the means by which this automatization is enabled relies on orienting the equations i.e. imposing a direction for the application of syntactic transformations derived from equations.

In the last section we have seen that the base building blocks of equational logic are (unordered) equations. Similarly, term rewriting is based on a particular kind of oriented equations that are called rewrite rules [51] or reduction rules [68].

Definition 2.21: Rewrite Rules

For any two terms $(l, r) \in \mathcal{T}_\mathcal{F}(\mathcal{X})^2$, the ordered pair (l, r) is said to be a rewrite rule if:

- $l \notin \mathcal{X}$
- $var(r) \subseteq var(l)$

We may then denote any such reduction rule (l, r) by $l \rightsquigarrow r$.

Applying a rewrite rule $l \rightsquigarrow r$ on a term $x \in \mathcal{T}_\mathcal{F}(\mathcal{X})$ requires that a sub-term of x "matches" the left-hand side l of the rule. This means that there exists a position $p \in pos(x)$ and a substitution $\phi \in \text{Sub}(\mathcal{T}_\mathcal{F}(\mathcal{X}))$ such that $x_{|p} = \phi(l)$. We may then say that $x_{|p}$ is an instance of the left-hand side. Then, the application of the rule yields $x[\phi(r)]_p$ i.e. we replace the sub-term at position p by $\phi(r)$. We may also say that we replace the instance $\phi(l)$ of the left-hand side by the corresponding instance $\phi(r)$ of the right-hand side [51].

We can remark that unlike equations, which are unordered, rewrite rules, which are ordered, only allow to replace instances of the left-hand side l by corresponding instances of the right-hand side r and not the other way around.

A Term Rewriting System (TRS) is then defined as a finite set of rewrite rules (we may define TRSs with infinitely many rules [51] but this is outside the scope of this thesis).

A TRS R is associated to a rewrite relation $\rightarrow_R$ which specifies atomic transformations corresponding to the application of a rewrite rule. As explained in [70], the closure $\rightarrow_R^*$ may be called the derivation of $\rightarrow_R$ because it includes all sequences of transformations that may be derived from $\rightarrow_R$. Also, $\leftrightarrow_R^*$, which we may also denote by $\approx_R$ may be called the equational theory generated by R when considered as a set of equations (i.e. we remove the constraint on the orientation of rules) [70].

A rewrite relation $\rightarrow_R$ may relate infinitely many terms. As a result it may be impossible to represent it in-extenso. The TRS R can then be described as a finite representation of $\rightarrow_R$.

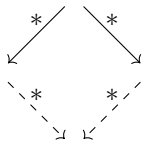
Definition 2.22: Term Rewriting Systems & rewrite relations

A Term Rewriting System (TRS) is a set R of rewrite rules.
Any such TRS R specifies a one-step rewrite relation $\rightarrow_R \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$ such that for any two terms x and y in $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ we have $x \rightarrow_R y$ iff there exists a rule $l \rightsquigarrow r \in R$, a position $p \in pos(x)$ and a substitution $\phi \in \text{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X}))$ such that $x_{|p} = \phi(l)$ and $y = x[\phi(r)]_p$.

Whenever we have $x \rightarrow_R y$ with $x_{|p} = \phi(l)$ and $y = x[\phi(r)]_p$ we may say that $x_{|p}$ is a redex and that x is reducible by $\rightarrow_R$. If, considering all rules $l \rightsquigarrow r \in R$, there are no redexes in x, then there exists no y s.t. $x \rightarrow_R y$ which we may denote by $x \not\rightarrow_R$ and we may say that x is irreducible modulo R. For any term x, if we have another term y such that $x \rightarrow_R^* y$ then we may say that y is a descendent of x and that x is an ancestor of y. If, in addition, y is irreducible, then y is said to be a normal form of x [51].

For any term x and any TRS R, it is not guaranteed that a normal form of x exists. We may indeed never reach an irreducible term. If this is the case then there must exist an infinite chain $x \rightarrow_R \cdots$ starting from x. Therefore, we can reciprocally conclude that a normal form must exist if there are no infinite chains $x \rightarrow_R \cdots$ starting from x. As the reader may have noticed, this correspond to having $\rightarrow_R$ being a noetherian binary relation, which we have defined in Sec.2.2.3. In the context of term rewriting, we may rather say that the TRS R is terminating (Def.2.23).

As a result, if R is terminating then this guarantees the existence of a normal form for any term x. In fact, termination is also called strong normalization because "normalization" states that there exists at least one finite sequence of applications of $\rightarrow_R$ leading towards a normal form. Termination is a stronger property because it states that that all sequences of applications of $\rightarrow_R$ eventually lead to a normal form.



confluence

Figure 2.2: A representation of confluence

Definition 2.23: Termination of a TRS

Given R, a reduction system, $\rightarrow_R$ is "terminating" (also called "strongly normalizing" or "noetherian") if there exist no infinite series $(t_j)_{j\geq 1}$ of terms such that for any $j \in \mathbb{N}$, $t_j \rightarrow_R t_{j+1}$. This means that any reduction sequence must eventually terminate.

Given that there can exist many possible applications of rules derived from R, there can exist many paths $x \overset{*}{\rightarrow}_R \cdots$ starting from x. As a result, even if R is terminating, the uniqueness of the normal form is not guaranteed.

We have seen that the existence of normal forms is linked to the property of termination (which is even stronger than that as we have seen). Similarly, the uniqueness of a normal form can be linked to a property of "confluence". Confluence refers to the ability to find the same resulting terms when taking different paths resulting from the application of $\rightarrow_R$.

Confluence, that we define in Def.2.24 can be described with the diagram from Fig.2.2. In this diagram, plain arrows correspond to the hypothesis while dashed arrows correspond to conclusions. A relation is confluent if for any three terms x, y and z such that we have $x \overset{*}{\rightarrow}_R y$ and $x \overset{*}{\rightarrow}_R z$ i.e. such that y and z have a common ancestor x then it implies that y and z have a common descendant x' such that $y \overset{*}{\rightarrow}_R x'$ and $z \overset{*}{\rightarrow}_R x'$.

Definition 2.24: Confluence of a TRS

A TRS R is confluent if $\overset{*}{\leftarrow}_R \circ \overset{*}{\rightarrow}_R \subseteq \overset{*}{\rightarrow}_R \circ \overset{*}{\leftarrow}_R$

When a TRS R is both terminating and confluent, then it is said to be a convergent. In this case, for any term t, we have both the existence and the unicity of a normal form that we may denote[1] by $R(t)$.

Let us now go back to our initial problem, which is that of determining whether or not two terms t_α and t_β are related by a congruence $\approx_E$ of base E. We have seen that it suffices to find a path $t_\alpha \approx_E t_1 \approx_E \cdots \approx_E t_n \approx_E t_\beta$. Let us then consider a TRS R such that each rule $l \rightsquigarrow r \in R$ verifies that either $l \approx r \in E$ or $r \approx l \in E$ i.e. such that each rule correspond to a specific orientation of an equation taken from E. As a result, we have that $\rightarrow_R \subset \approx_E$ and therefore $\overset{*}{\leftrightarrow}_R \subseteq \approx_E$ because $\approx_E$ is a congruence. Let us now suppose that R is convergent. As a result the normal forms $R(t_\alpha)$ and $R(t_\beta)$ are uniquely defined, and,

[1] we use the notation $R(t)$ from [51] so as not to overlap with the $\downarrow$ notation from process calculus that we use in Chap.5

if $R(t_\alpha) = R(t_\beta)$ then we have $t_\alpha \overset{*}{\rightarrow}_R R(t_\alpha) = R(t_\beta) \overset{*}{\leftarrow}_R t_\beta$ and therefore $t_\alpha \approx_E t_\beta$. This means that $R(t_\alpha) = R(t_\beta) \Rightarrow t_\alpha \approx_E t_\beta$ but the reciprocate may not be necessarily true. For it to be true we would need that $\overset{*}{\leftrightarrow}_R = \approx_E$. This can be ascertained when every equation of E is found in R in a certain orientation.

2.2.7 Ordered rewriting & rewriting modulo theories

Let us consider the example from Fig.2.3. In this example, we consider a term algebra of Boolean expressions with the "$\vee$" disjunction operator. Given t_1, t_2 and t_3 three terms, we describe the transformation of $(t_1 \vee t_2) \vee (t_3 \vee t_1)$ into $t_1 \vee (t_2 \vee t_3)$. This transformation is possible with the following TRS:

$$R = \left\{ \begin{array}{llll} (x \vee y) \vee z \rightsquigarrow x \vee (y \vee z) & (1) & x \vee y \rightsquigarrow y \vee x & (2) \\ x \vee (y \vee z) \rightsquigarrow (x \vee y) \vee z & (3) & x \vee x \rightsquigarrow x & (4) \end{array} \right\}$$

Indeed, the sequence of transformations described on Fig.2.3 consists in the application of applying rule (1) at $p = \epsilon$ then rule (2) at $p = 22$, then rule (3) at $p = 2$, then rule (2) at $p = 21$, then rule (1) at $p = 2$, then rule (3) at $p = \epsilon$, then rule (4) at $p = 1$.

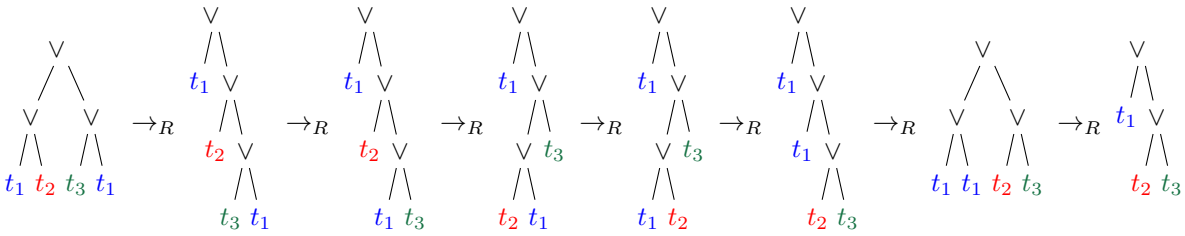


Figure 2.3: A sequence of transformations making use of associativity and commutativity

We can see that we have managed to "simplify" the initial term through some applications of $\rightarrow_R$. However the TRS R is not terminating. Indeed, given variables x, y and z, we have the existence of the infinite chain $x \vee y \rightarrow_R y \vee x \rightarrow_R x \vee y \rightarrow_R \cdots$, where we repeat indefinitely the application of rule (2) at $p = \epsilon$. Another example of an infinite chain would be $x \vee (y \vee z) \rightarrow_R (x \vee y) \vee z \rightarrow_R x \vee (y \vee z) \rightarrow_R \cdots$, where we indefinitely repeat applying rule (3) and then rule (1) at $p = \epsilon$.

The TRS not being terminating, we cannot guarantee that the process may yield any normal form. And, as a result, we cannot automatize the process described in Fig.2.3, which aims at "simplifying" such terms.

As explained in [109], properties of operation symbols such as commutativity and associativity are challenging when it comes to the definition of terminating rewrite systems. In this section we explain how we can use ordered rewriting and rewriting modulo theories as a two part solution to tackle those issues. In particular, we show that the human-guided process described on Fig.2.3 can be automatized.

Ordered rewriting

An Ordered Rewriting System (ORS) [51] is a pair $(E, >)$ denoted as $E >$ where E is an axiom system (i.e. a set of unordered equations) and $>$ is a rewrite ordering (i.e. an antireflexive, transitive, antisymmetric,

$\mathcal{F}$-compatible binary relation that is stable under substitution).

In ordered rewriting we apply equations from E in either order i.e. either from left to right or from right to left. However their application is only enabled under specific conditions [51]. As in the case of TRSs, the application of a rewrite rule $l \rightsquigarrow r$ consists in having a transformation $x \to y$ so that there exists a position $p \in pos(x)$ and a substitution $\phi \in \text{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X}))$ such that $x_{|p} = \phi(l)$ and $y = x[\phi(r)]_p$. However, with ordered rewriting the application of such a transformation is conditioned by having $\phi(l) > \phi(r)$.

As a result, equations from E can be applied in whichever direction agrees with the ordering $>$ on terms. Let us for instance suppose that we have an equation $f(x,y) \approx f(y,x) \in E$, which corresponds to the commutativity of symbol f. Then, if $f(x,y) > f(y,x)$, we may apply $f(x,y) \to f(y,x)$. Also, given that $>$ is a rewrite ordering we have that $f(x,y) > f(y,x)$ implies that $f(y,x) \not> f(x,y)$ and hence $f(x,y) \not\to f(y,x)$ i.e. we cannot apply the rule in the other direction.

Definition 2.25: Ordered Rewriting System

An Ordered Rewriting System (ORS) is a pair $(E, >)$ denoted as $E >$ where E is an axiom system and $>$ is a rewrite ordering. An ORS $E >$ specifies a one-step rewrite relation $\to_{E>} \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$ s.t. $\forall\ (x,z) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$:

$$(x \to_{E>} y) \Leftrightarrow \left(\begin{array}{l|l} \exists\ (l \rightsquigarrow r) \text{ s.t. } ((l \approx r) \in E) \vee ((r \approx l) \in E), & (x_{|p} = \phi(l)) \\ \exists\ p \in pos(x), & \wedge\ (y = x[\phi(r)]_p) \\ \exists\ \phi \in \text{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X})) & \wedge\ (\phi(l) > \phi(r)) \end{array} \right)$$

We formally define ORSs in Def.2.25. As for TRSs, ORSs are associated to rewrite relations. We denote by $\to_{E>}$ the rewrite relation associated to the ORS $E >$. Resulting from that definition, we have that, whenever $x \to_{E>} y$, we must have $x > y$ because the transformation $x \to_{E>} y$ decreases the rank of a sub-term x_p at a certain position p (without modifying the remainder of the term i.e. its context), and because $>$ is $\mathcal{F}$-compatible.

Also, as explained in [51], the ordered-rewriting relation $\to_{E>}$ verifies $\to_{E>} \subseteq (\approx_E \cap >)$. The fact that it is contained in $\approx_E \cap >$ (as sets of related terms) is explained by the fact that a transformation specified by $\to_{E>}$ can only occur if (1) it corresponds to an equation from E and (2) if it decreases the order of the involved terms.

In particular, if $>$ is a total order, then $\to^*_{E>} = (\approx_E \cap \geq)$ (with $\geq = (> \cup =)$) and hence $\leftrightarrow^*_{E>} = \approx_E$ because $\approx_E$ is a congruence and, $>$ being total, $(> \cup = \cup <)$ relates all terms.

Also, given that $\to_{E>} \subseteq >$, in order to have a terminating rewrite system, it suffices that $>$ be a noetherian order (defined in Sec.2.2.3).

Let us now go back to our example from Fig.2.3. Let us consider a rewrite ordering $>$ such that:

- $t_1 < t_2 < t_3$
- and for any variables x, y and z:

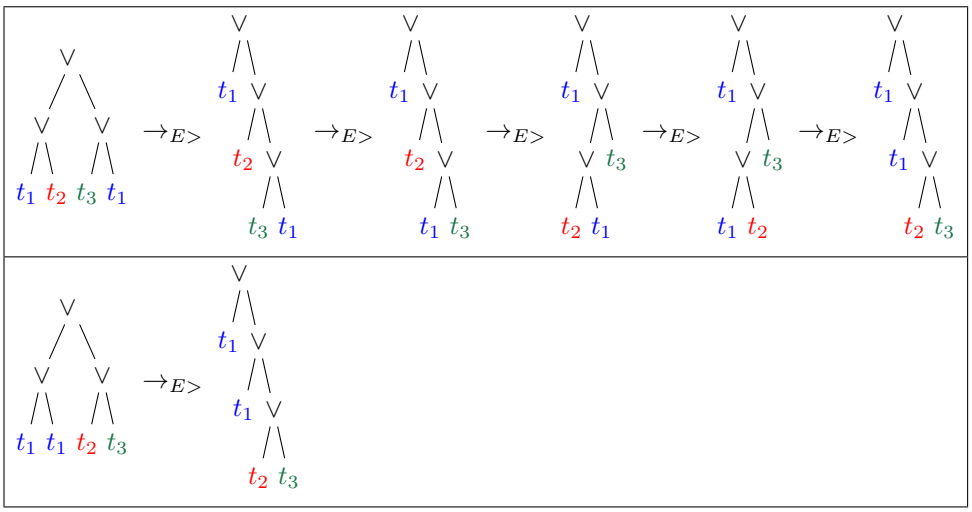


Figure 2.4: Reduction of associative and commutative expressions using an ORS

– if $x < y$ then $(x \vee y) \vee z > x \vee (y \vee z)$

– if $x > y$ then $(x \vee y) \vee z < x \vee (y \vee z)$

– if $x < y$ then $y \vee x > x \vee y$

Then, given $E = \{x \vee (y \vee z) \approx (x \vee y) \vee z,\ x \vee y \approx y \vee x\}$, the two sequences of transformations by $\rightarrow_{E>}$ represented on the two rows of Fig.2.4 hold. Furthermore, supposing that t_1, t_2 and t_3 are irreducible, the last term that is reached (at the right-most of each row) is irreducible.

In particular, Fig.2.4 also demonstrates that the first terms of each row are related by E. Indeed, given that $\rightarrow_{E>} \subseteq (\approx_E \cap >)$ we have that $\leftrightarrow^*_{E>} \subseteq \approx_E$ and hence the terms are related by E because we have:

$$(t_1 \vee t_2) \vee (t_3 \vee t_1) \ \rightarrow^*_{E>}\ t_1 \vee (t_2 \vee t_3) \ \leftarrow^*_{E>}\ (t_1 \vee t_1) \vee (t_2 \vee t_3)$$

Class rewriting

Let us consider an axiom system T, which in this context is called an equational theory and a TRS R. Let us recall that T contains a (finite) number of unordered equations while R contains a (finite) number of ordered rewrite rules. We can generalize the notion of term rewriting as follows [51]: for any two terms x and z, we may say that x rewrites by R into z modulo T which we denote by $x \rightarrow_{R/T} z$, if there exists a substitution $\phi \in \mathrm{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X}))$, a term $y \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})$, a position $p \in pos(y)$ and a rule $l \rightsquigarrow r \in R$ such that $x \approx_T y[\phi(l)]_p$ and $y[\phi(r)]_p \approx_T z$.

In effect, what is done here is that the TRS R is applied, not on the set of terms $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$, but instead on the quotient set $\mathcal{T}_{\mathcal{F}}(\mathcal{X})/T$ of the congruence classes $[\ ]_{\approx_T}$.

This type of rewriting may be called rewriting modulo theories [96] (because, on this example, the rewriting by R is done modulo T) or class rewriting (or congruence class rewriting) [51] because of the previous remark. We formally define Class Rewriting Systems (CRSs) in Def.2.26.

Definition 2.26: Class Rewriting System

A Class Rewriting System (CRS) is a pair (R, T) denoted as R/T where R is a TRS and T is an axiom system. A CRS R/T specifies a one-step rewrite relation $\rightarrow_{R/T} \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$ s.t. $\forall\ (x, z) \in \mathcal{T}_{\mathcal{F}}(\mathcal{X})^2$:

$$(x \rightarrow_{R/T} z) \Leftrightarrow \left(\begin{array}{l|l} \exists\ (l \rightsquigarrow r) \in R,\ \exists\ \phi \in \mathrm{Sub}(\mathcal{T}_{\mathcal{F}}(\mathcal{X})), & (x \approx_T y[\phi(l)]_p) \\ \exists\ y \in \mathcal{T}_{\mathcal{F}}(\mathcal{X}),\ \exists\ p \in pos(y) & \wedge\ (y[\phi(r)]_p \approx_T z) \end{array} \right)$$

Let us now go back to our example from Fig.2.3. Let us then consider the equational theory $T = \{x \vee (y \vee z) \approx (x \vee y) \vee z,\ x \vee y \approx y \vee x\}$ and the TRS $R = \{x \vee x \rightsquigarrow x\}$. The application of the CRS R/T on the initial term (on the top left of Fig.2.3) of the example then yields the final term (on bottom right of Fig.2.3).

This is illustrated on Fig.2.5. Indeed, we have seen previously that using ordered rewriting, we can ascertain that the two terms t_α and $t\beta$ on the top left of Fig.2.5 are related by $\approx_T$. Then, we immediately have that the rule $x \vee x \rightsquigarrow x$ can be applied on t_β s.t. we have $t_\beta \rightarrow_R t_\gamma$, as illustrated on the top right of Fig.2.5. From those two hypotheses, using the definition of rewriting modulo theories, we can conclude that $t_\alpha \rightarrow_{R/T} t_\gamma$, as illustrated on the bottom of Fig.2.5.

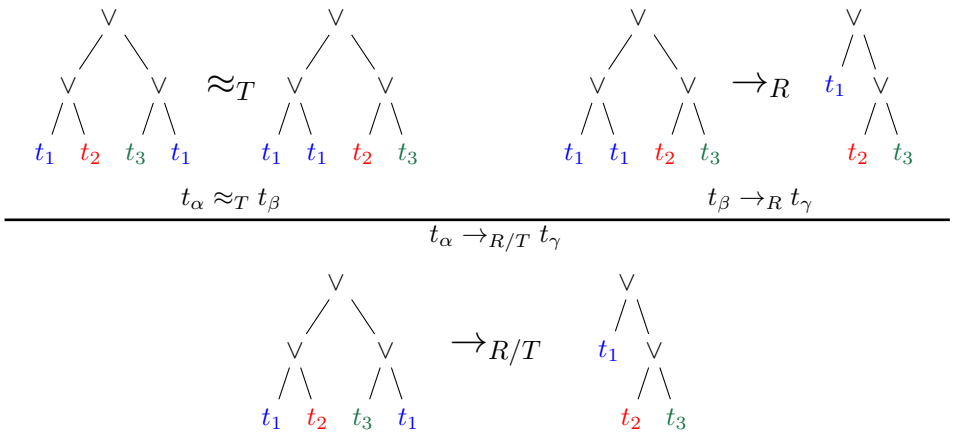


Figure 2.5: Solution of the problem of Fig.2.3 using ordered rewriting and rewriting modulo theories

As a result, with $\rightarrow_{R/T}$, we have managed to obtain the same result as the human-guided sequence of transformations presented on Fig.2.3. This constitutes an automation of this process because the resulting term (at the bottom right of Fig.2.5) is irreducible w.r.t. $\rightarrow_{R/T}$ and is therefore a normal form.

Let us now make an important remark which is that, whenever the equations that constitute the equational theory T correspond to properties of associativity and commutativity of the operation symbols (which was the case in our example), we may use the term Associative-Commutative Rewriting (AC-R) [70, 96] to describe $\rightarrow_{R/T}$.

AC-R is an interesting subject of study because of its use for defining rewrite systems that terminate on term algebras with associative and commutative operators but also because whenever an equational theory

T only contains equations corresponding to those properties then it enjoys additional properties (w.r.t. other equational theories). For instance, as mentioned in [70], if T only contains equations of AC then $\approx_T$ is always decidable i.e. for any two terms x and y we can always determine whether or not $x \approx_T y$. This is because the AC-congruence classes of finite terms are finite. Using any total rewrite ordering on the terms, we can then always unify AC-equivalent terms.

2.3 Modeling Distributed Systems with Interaction Languages

We have evoked in the Introduction (Chap.1) various manners to model DSs. In the context of this thesis however, we have chosen to focus on Interaction Languages (ILs) which propose models that (1) specify the behavior of DSs in terms of their internal and external communications and (2) can be drawn in a graphical and intuitive manner. In this section, by introducing specific ILs we intend to explain the general principles of interactions and to give a quick overview of existing languages.

We present at first the family of Message Sequence Chart (MSC) and its sub-languages with, in Sec.2.3.1 Basic Message Sequence Chart (BMSC), in Sec.2.3.2 High-Level Message Sequence Chart (HMSC) and in Sec.2.3.3 MSCs with inline expressions which are generally simply called MSCs. Then, in Sec.2.3.4 we present Sequence Diagrams from the Universal Modeling Language (UML).

2.3.1 Basic Message Sequence Charts

Basic Message Sequence Chart (BMSC) [97], include the most basic constructs from the Message Sequence Chart (MSC) standard [22]. BMSCs describe some asynchronous communications that can happen within and without a DS in terms of the exchange of messages, which are abstracted as simple labels. Those exchanges consist of observable atomic events corresponding to the emission and / or reception of said messages. A BMSC specifies possible successions of those atomic events, no two distinct events being able to occur simultaneously. Those events occur on the interface of specific sub-systems of the DS. Given that BMSCs do not have to be complete specifications, we can abstract a group of sub-systems as a single actor within the communication scheme that is described. Those abstractions correspond to loci from which we can observe parts of the exchanges that occur within (i.e. with other such loci) or without (i.e. with "the environment") the DS. In BMSC, those abstractions are named instances [97, 59, 98].

Now that we have introduced what BMSCs aim to model, let us explain how it is done. BMSCs are represented graphically. In those diagrams, each instance is denoted by a vertical line. Those vertical lines are aligned and spaced horizontally. They all start at the top of the diagram and end at its bottom. On the top of each instance, its name can be written so as to distinguish it from the others. On the vertical axis representing a given instance are represented the atomic communication actions (emissions and receptions) that are expected to occur on the corresponding locus. In BMSCs, there exists a total order (locally w.r.t. a instance) which defines the order in which those events are supposed to occur. This order is represented graphically by the top to bottom direction. An event represented below another must occur after it.

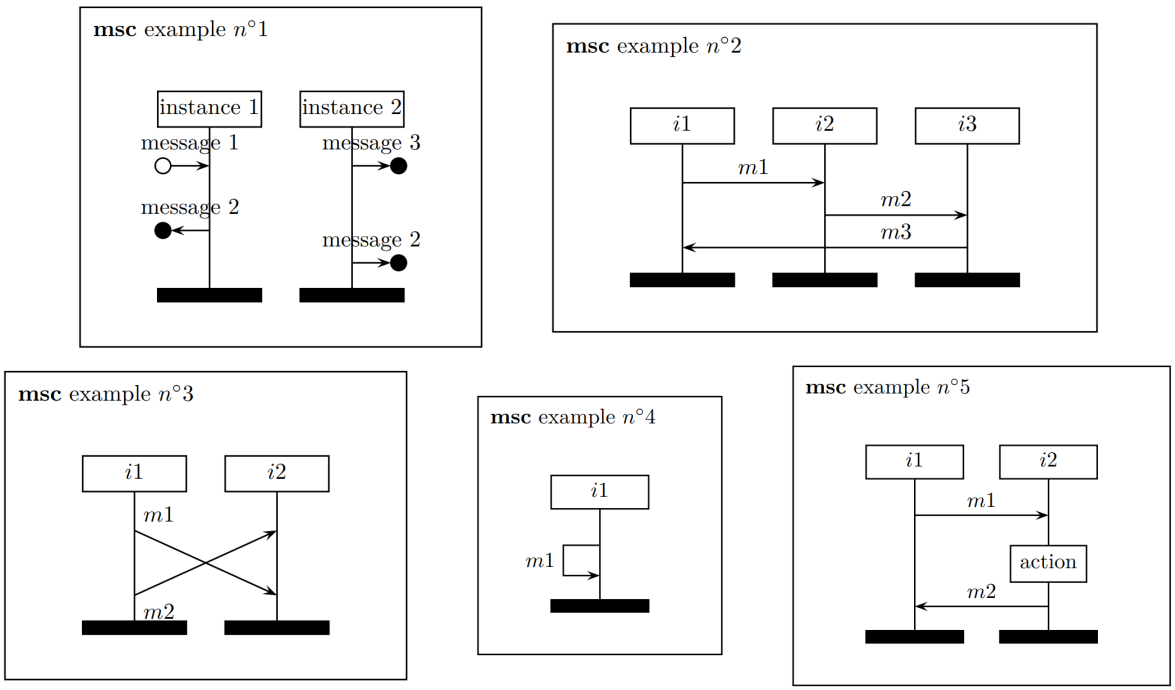


Figure 2.6: Examples of Basic Message Sequence Charts

On Fig.2.6 we have drawn some example BMSCs. In the one on the top-left of Fig.2.6, we have two instances named "instance 1" and "instance 2". On "instance 1", a message named "message 1" is received from the environment and then another message named "message 2" is send towards the environment". Likewise, two other atomic communication events : the emission of "message 3", and then that of "message 2" are represented on the "instance 2" instance on the right.

Also, there is a-priori no correlation between the execution of events occurring on different loci. On the example in the top-left of Fig.2.6, the emission of "message 2" by "instance 2" can occur indifferently before or after the reception of "message 1" by "instance 1". Indeed, the top to bottom order only applies for actions that occur on a same instance. As such, let us remark that the vertical positioning of the actions under "instance 1" w.r.t. those under "instance 2" does not matter. In this example, we could have drawn the two actions on the right above the two on the left, on the same lines, interleaved, or under them.

An order between events occurring on different instances (which are independent) can only exist (be enforced) by a causal relation such as the one linking the emission of a message to its reception (of the same message). In the drawing of BMSCs, we use plain arrows that may link an emission action on a certain instance, to a corresponding reception action on another instance. Those plain arrows represent the concept of the passing of a message, with the logical implication that the reception action must occur after the emission action (given that we model asynchronous communications). We illustrate this in the example in the top-right of Fig.2.6. Here, for instance, the message "m1" is send from instance "i1" towards "i2". In the global ordering of atomic actions, the reception event occurs after the emission event. Also, by construction, given that the emission of "m2" must occur after the reception of "m1" on "i2", the passing of "m1" must occur entirely before the passing of "m2" can start. Likewise, the passing of "m3" from "i3" to

"i1" can only start once "m2" has been received by "i3". Those chained / interlinked precedence relations makes so that only some successions of events globally correspond to what is specified by the BMSC. In the example on the top-right of Fig.2.6, even though instances "i1", "i2" and "i3" represent independent and concurrently running processes, there is a single possible succession of events due to the interlinked causal relations between receptions and subsequent emissions of messages. That succession is the passing of "m1" (emission then reception) then that of "m2" and then that of "m3".

Let us keep in mind however, that in most cases, many different successions of events can be specified by a single BMSC. It is the case in the example on the top-left of Fig.2.6, where we have 6 different possible successions of events.

Usually, plain arrows, representing the passing of messages, are drawn horizontally between two instances. However, exceptions are allowed in some formalisations of BMSCs, to represent the overtaking of a message passing w.r.t. another. This is illustrated on the bottom-left of Fig.2.6, where message "m1" and "m2" are both send by instance "i1" to instance "i2". However, although the emission of "m1" occurs before that of "m2", here, the reception of "m1" must occur after that of "m2". Another exception concerns messages that an instance may send to itself. In this case, we may draw an arrow that exits the instance (towards the left or the right) and then bends back towards the origin instance to land below the original emission action, as is illustrated on the example in the middle of the bottom row of Fig.2.6. Also, in some formalisations of BMSCs, the definition of some other local actions, which are neither emission nor reception events are possible, as illustrated on the bottom-right of Fig.2.6.

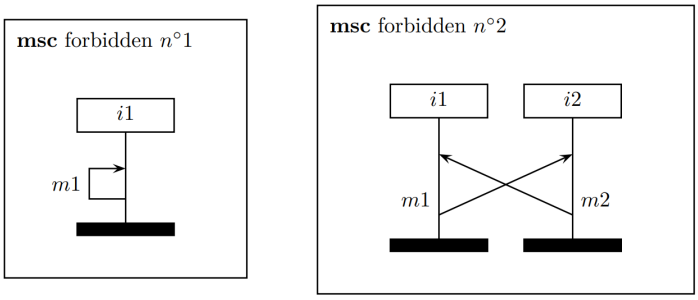


Figure 2.7: Two cases of forbidden constructions in BMSCs

Let us also note that some constructions are explicitly forbidden in BMSCs (although it may be possible to draw them and it may be possible to construct them syntactically in some formalisations of BMSCs). Those constructions violate a principle which states that an emission action may not be causally dependent (directly or though other actions) w.r.t. the reception action it enables. Fig.2.7 illustrates constructions that violate this principle. On the example on the left, the emission of message "$m1$" is drawn below the reception of that same message "$m1$"; here the violation is direct. On the example on the right, the emission of "$m1$" must come after the reception of "$m2$"; however, the emission of "$m2$" must come after the reception of "$m1$".

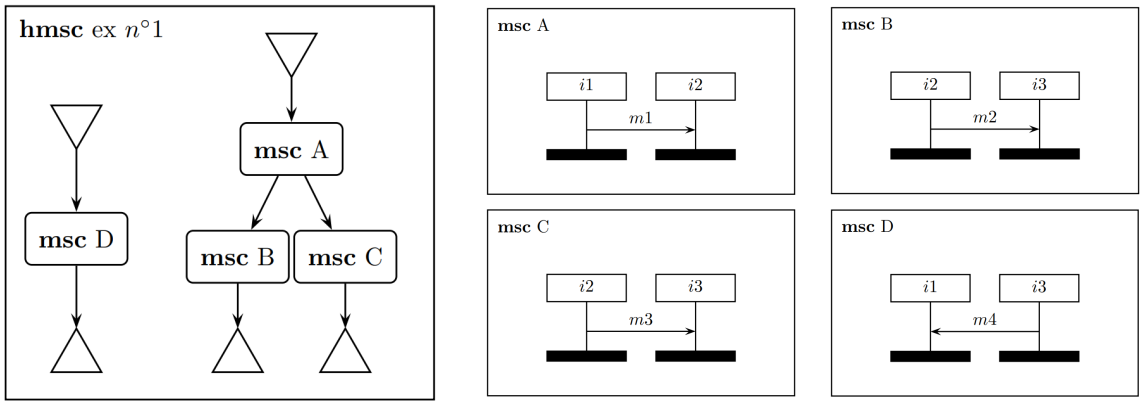


Figure 2.8: Example of High-Level-MSC

2.3.2 High Level Message Sequence Charts

High-Level Message Sequence Chart (HMSC) [98] allows the description of more complex scenarios via the composition of BMSCs within graph-like structures.

A HMSC presents itself (i.e. can be drawn) as a graph in which individual nodes can be: start nodes (denoted by $\bigtriangledown$), end nodes (denoted by $\bigtriangleup$), connector nodes (denoted by $\bigcirc$) or reference nodes, containing a reference to a BMSC.

HMSCs constitute directed graphs. Nodes are linked by arrows which represent the expected order of execution of the behaviors specified by individual BMSC nodes. An arrow originating from a reference to BMSC "A", and targeting a reference to BMSC "B" signifies that we expect the behavior specified by BMSC "A", followed by the one specified by BMSC "B". When there are several outgoing arrows from a single node (and/or can be represented as an arrow that splits), this means that there is a alternative in the choice of the following behavior, which can be specified by either one of the targeted BMSCs. Also, given that the graph structure of HMSCs allows cycles, one can specify the repetition of behaviors and hence it is possible to represent arbitrarily long behaviors.

An HMSC graph can be divided into (one or several) connected subgraphs (called "components"), each of which having exactly one start node (denoted by $\bigtriangledown$). Components represent processes that are executed in parallel w.r.t. one another.

Let us consider the example from Fig.2.8. In this HMSC we have 2 components. The component drawn in the left specifies the execution of a single BMSC which is "D". The expected behavior is therefore the emission of "m4" by "i3" and followed by its reception by "i1". The component drawn in the right involves three BMSCs "A", "B" and "C". In a first time, "A" is expected to occur (i.e. the emission of "m1" by "i1" followed by its reception by "i2". Then, because of the split arrow, we have either "B" or "C" i.e. the transmission of either "m2" or "m3" from "i2" to "i3". Given that the two components are expected to be executed in parallel, various interleavings are allowed.

In the example from Fig.2.9, we modified our previous example so as to express that the behaviors specified by individual components can be repeated (here it is once or more). We can use connector nodes

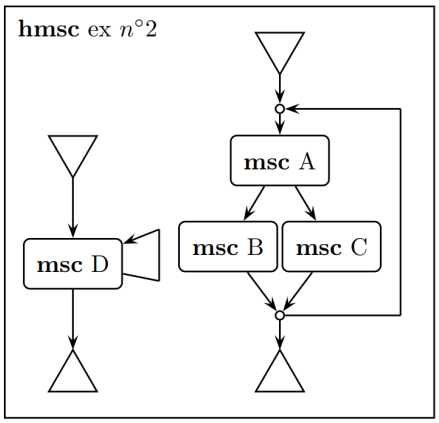


Figure 2.9: Example of High-Level-MSC with repetitions and connector nodes

(denoted by ◯) as a syntactic sugar to draw more easily readable diagrams. In facts a connector node synthesize any combination between an incoming arrow and an outgoing arrow. For example here, without the connector node below nodes "B" and "C", we would have had to draw 4 arrows: from "B" to the end node, from "C" to the end node, from "B" to "A" and from "C" to "A", making the drawing more cluttered.

2.3.3 Message Sequence Charts

Message Sequence Chart (MSC) is a trace language for the description of the communication behavior of system components and their environment. It has been developed since 1993 (for its first version) by the International Telecommunications Union (ITU). The official documentation of the standard [22], the latest version of which dates from 2011, presents MSCs via a proposed textual (syntax) and graphical denotation. The description of an informal semantics is also provided in [22]. The MSC standard authorizes more complex structural models in the form of "MSC Documents". As a result, within the MSC standard we can discriminate between several sub-languages among which Basic Message Sequence Chart (BMSC) and High-Level Message Sequence Chart (HMSC) which we have already presented. In addition, the MSC standard allows the definition of sequence charts containing more complex features than those of BMSCs, such as the use of inline operators, references or conditions. There is no specific name given for this extension of BMSC, and we may refer to it simply as MSC. Some of the extended features of MSC are illustrated on the example from Fig.2.10:

- "inline-expression" are operators that can be used to "relate" several parts of an MSC. The "relation" that is defined by an "inline-expression" can correspond to various notions. We have for instance the alternative ("alt") for specifying a choice, the parallel composition ("par") for specifying concurrent execution, or an operator for describing optional behavior ("opt") etc.

- references, which allow the nesting of MSCs by including a reference to another MSC within some part of a parent MSC. In that case, the instances of the parent MSC must match those of the nested MSC (or there must be some correspondence via some renaming mechanism). In addition we can make use of gates so as to link (1) inputs from the environment defined in the nested MSC to outputs produced

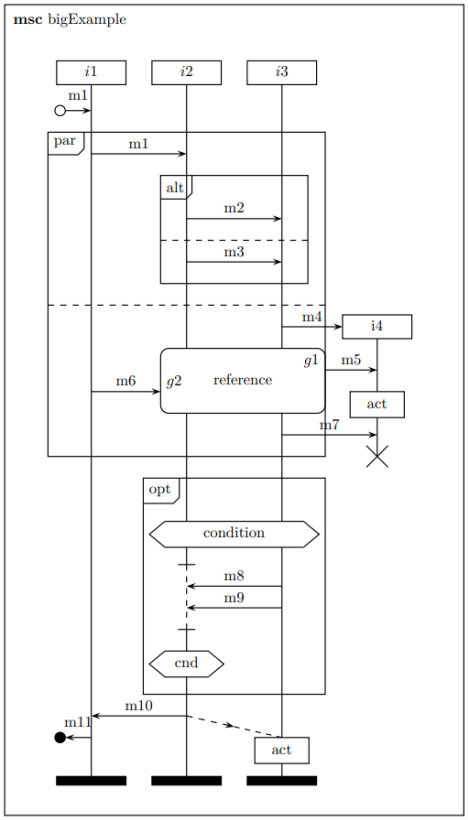


Figure 2.10: Example showcasing the expressivity of the MSC standard

by an instance defined in the parent MSC and (2) outputs towards the environment defined in the nested MSC to inputs received by a given instance defined in the parent MSC

- the evaluation of conditions in some parts of the MSC
- the definition of explicit causal orderings between a-priori independent events (for instance the emission of $m10$ by $i2$ on the example from Fig.2.10 explicitly implies the action act on $i3$)
- etc.

Let us remark however that the MSC language being quite expressive, most formal approaches only deal with some aspects of it.

2.3.4 UML Sequence Diagrams

Universal Modeling Language (UML) is a general-purpose modelling language which aims to standardize and gather, in a coherent ecosystem, various notations that are used for the modelling and design of (predominantly) software systems. UML was adopted in 1997 by the Object Management Group (OMG) and later approved as a standard in 2005 by the International Organization for Standardization (ISO). UML includes in a single ecosystem different types of models which can contain references to one another. Since its 1.1 version was released in 1997, the UML standard has considerably evolved, notably in 2005 with the 2.0 major revision. Given the scope and depth of the UML, there is no formal semantics that is associated

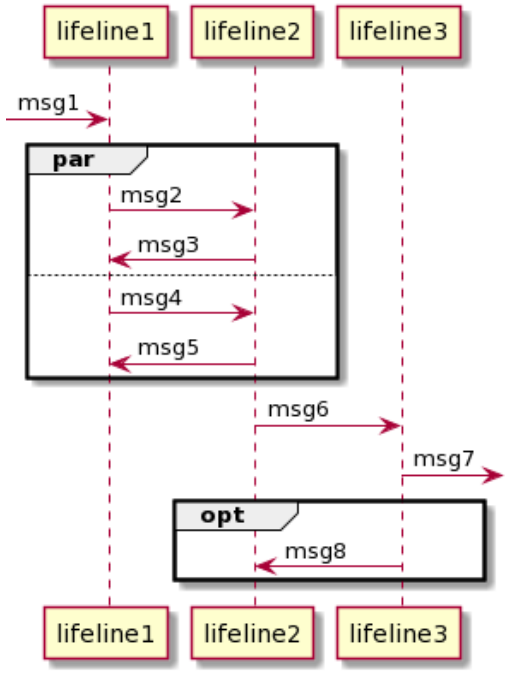


Figure 2.11: Example UML-SD

to it. The official documentation (of the current 2.5.1 version) [24] only describes the intended meaning of models in natural language.

In any case, the scope of UML is far greater than that of more specialized formalisms such as MSC. In the context of this overview, we only focus on a specific type of diagram within UML: "Sequence Diagrams" (abbr. UML-SD), which corresponds to the inclusion of a form of MSC into UML (which dates back from version 2.0).

UML-SDs and MSCs are almost identical when it comes to their visual representation (apart from purely aesthetic considerations). They take the form of a set of vertical lines, each representing a sub-system of the modelled system. Horizontal or diagonal arrows drawn between those vertical lines may represent the exchange of messages between sub-systems. The default interpretation of the vertical axis is that of the passage of time, from top to bottom. As in the case of MSCs with inline-operators, some operators may be used to change this default interpretation. On Fig.2.11 we provide an example UML-SD (drawn using PlantUML [5] / PlantText [4]). In the context of this thesis, we use some vocabulary taken from UML-SDs with, in particular, the word "lifeline" to refer to the vertical lines that may abstract sub-systems of the modelled DS.

Although it is widely used in the industry thanks to its graphical nature and the existence of numerous tools such as Papyrus [3] or Visual Paradigm [18] which allows the creation and drawing of UML models, UML is often criticized in academia for using an overly complex metamodel [120, 56] that even contains some unspecified parts ("semantic variation points"). In practice, it is always some formed of restricted and modified version of UML-SDs that are used in formal approaches, with for instance the extraction of a simplified metamodel [100, 114]. Moreover, some specific operators of UML-SDs such as "*assert*" or "*neg*" are particularly controversial, given that their semantics, as informally described by the norm [24]

can be interpreted in different manners, as explained in [100, 63]. The authors of [111] describe those "*new constructs*" as opening-up "*a veritable pandora's box of expressions whose meaning is obscure*". As a result, most if not all formal approaches to the understanding of UML-SDs use a heavily restricted subset of the language, as in [122, 75, 34, 52, 47, 114] (among others).

2.4 Process Algebras

Process algebras are mathematical constructs used to represent the behavior of a system (in the word "process") and which allow the study of parallel or distributed systems via algebraic means [30, 115] (hence the word "algebra").

Process algebras represent behaviors via what can be observed of them i.e. sequences of observable events. Some process algebras can be used to treat continuous phenomena [115]. However, in the context of this thesis, we only consider discrete events i.e. their occurrences correspond to unique instantaneous points in time. In that case, the authors of [30] describe processes as "discrete event systems".

The treatment of behavior with process algebra is said to be algebraic because a process algebra forms an algebra in which we can perform calculations [38, 30] on objects representing current states of the system via some basic operators. The result of the application of operators can be inferred recursively on the structure of algebraic terms, starting from the application of base axioms.

The study of process algebra has led to the development of use of specific process algebras. Among those formalisms (which constitute formal languages), the most influential or fundamental include Communicating Sequential Process (CSP) [66] for describing concurrent systems, Calculus of Communicating Systems (CCS) [101] for describing in depth communications between 2 participants or Algebra of Communicating Processes (ACP) [38].

In the following we concretize this notion of process algebra by providing a minimalist example of a process algebra. This toy language is in facts the "Basic Process Algebra" from [38]. This toy example will also be used to demonstrate:

- how one can define a denotational-style semantics (inspired from [75])
- how one can define an operational-style semantics (as is classically done since [112])
- how one can use the Coq theorem prover to formally prove the equivalence of those two semantics (in the following section)

2.4.1 Presentation of the toy process algebra

Let us consider a system in which one can observe a variety of events, that we model by a finite set E of events. If the system is intended to express a single event $e \in E$ once, it is represented by the term e of the process algebra. The occurrence of this event then corresponds to the following transformation: $e \xrightarrow{e} \epsilon$, where ϵ denotes the empty process i.e. the process that corresponds to a behavior in which no event is

observed. Terms from E and ϵ constitute the constants of the algebra. In this context, E can be called the "signature" of the process algebra.

In addition to constants, our example process algebra may include more complex terms via the use of binary operators. Let us for instance introduce a "." operator for (strict) sequential composition and a "+" operator for alternative composition (here a non-deterministic choice). Given two distinct events e_1 and e_2 from E:

- the term $e_1.e_2$ denotes a system in which e_1 is expected to occur at first, and then e_2. As a result we expect the following sequence of transformations:

$$e_1.e_2 \xrightarrow{e_1} e_2 \xrightarrow{e_2} \epsilon$$

- the term $e_1 + e_2$ denotes a system in which either e_1 or e_2 is expected to occur (once). As such we may have two possible transformations:

$$\text{either} \quad e_1 + e_2 \xrightarrow{e_1} \epsilon \qquad\qquad \text{or} \quad e_1 + e_2 \xrightarrow{e_2} \epsilon$$

From the constants $E \cup \{\epsilon\}$ and the operators . and +, one can build words of any arbitrary size (in number of events). Terms of this process algebra can then be defined by the following grammar (using notation in the style of Backus–Naur Form (BNF) as in [75]) and we denote by $P(E)$ the set of all such process algebraic terms:

$$P \ ::= \ \epsilon \ \mid \ e \ \mid \ P_1.P_2 \ \mid \ P_1 + P_2$$

2.4.2 Semantic domain

Process algebras are by nature well suited for the definition of operational-style semantics. However we can also define semantics in denotational style for this kind of language. We will do both in the following. However, so as to compare them easily, we use a common semantics domain.

For this toy process algebra, a given execution of the system simply corresponds to a sequence of events from E. We can therefore use E^* as a semantic domain. The semantics of a process $p \in P(E)$ can be defined as a subset of E^*.

Let us note that in the free-monoid $(E^*, .)$:

- we denote the empty word (sequence of length 0) by ϵ, which is not the empty process ϵ
- "." refers to the concatenation operator such that for any two sequences of events t_1 and t_2 in E^*, $t_1.t_2 \in E^*$. It is not the same "." as the sequential composition of $P(E)$.

Additionally, we trivially extend the concatenation "." to sets of sequences of events so that for any T_1 and T_2 in $\mathcal{P}(E^*)$, we have $T_1.T_2 = \{t_1.t_2 \mid t_1 \in T_1,\ t_2 \in T_2\}$.

2.4.3 Denotational-style semantics

For the denotational approach, we can determine which are the accepted sequences of events by using the structure of the process algebraic terms (which is simple given the minimalist nature of this toy example). The following definition highlights the compositional nature of denotational semantics. Let us define $\sigma_d : P(E) \to E^*$ such that:

$$\begin{aligned}
\sigma_d(\epsilon) &= \epsilon \\
\sigma_d(e) &= e \\
\sigma_d(p_1.p_2) &= \sigma_o(p_1).\sigma_o(p_2) \qquad \text{with "." the extension to sets} \\
\sigma_d(p_1 + p_2) &= \sigma_d(p_1) \cup \sigma_d(p_2)
\end{aligned}$$

2.4.4 Operational-style semantics

We now define an operational-style semantics for the same syntax of terms $P(E)$, and the same semantic domain E^*. The operational semantics rely on the definition of transformation rules $p \xrightarrow{e} p'$ where p and p' are terms from $P(E)$ and e is an event in E. We have already seen the axiomatic rules of the form $e \xrightarrow{e} \epsilon$ that characterize the execution of a process that is reduced to a single event. Transformation rules for more complex terms can be inferred inductively. To formalize this, a presentation in the style of Plotkin [112] of structural operational semantics is usually proposed. This kind of semantics consists in inference rules of the form:

$$\frac{p_1 \ , \ \cdots \ , \ p_n}{c} \ f$$

In those rules, the goal is to assess (in the same manner as a mathematical proof) the conclusion c, which corresponds to a certain transformation on a complex term. The goal transformation c can be proved if some conditions on the sub-terms of the complex term are met. Those conditions are represented by $p_1, \cdots, p_n$, which are called "premises". Most oftentimes they correspond to transformations that are already proven on the subterms. f (for "formula") may correspond to additional conditions on the complex term. In the case where there are no premises ($n = 0$) and no additional conditions ($f = \top$), then the conclusion c is always accepted (tautology) and can be considered to be an axiom of the algebra.

For any $e \in E$ and any processes x, x', y and y':

$$\frac{}{\epsilon\downarrow} \qquad \frac{x\downarrow}{x+y\downarrow} \qquad \frac{y\downarrow}{x+y\downarrow} \qquad \frac{x\downarrow \qquad y\downarrow}{x.y\downarrow}$$

$$\frac{}{e \xrightarrow{e} \epsilon} \qquad \frac{x \xrightarrow{e} x'}{x+y \xrightarrow{e} x'} \qquad \frac{y \xrightarrow{e} y'}{x+y \xrightarrow{e} y'} \qquad \frac{x \xrightarrow{e} x'}{x.y \xrightarrow{e} x'.y} \qquad \frac{x\downarrow \qquad y \xrightarrow{e} y'}{x.y \xrightarrow{e} y'}$$

Figure 2.12: Inference rules for our toy process algebra

On Fig.2.12 are given the inference rules for our toy process algebra. Here the predicates $x\downarrow$ signify that the process may immediately terminate successfully. The only constant that may terminate immediately

is the empty process (successful termination) ϵ. However, more complex terms such as $\epsilon + x$ (for any process x) may also immediately terminate successfully. The $x \xrightarrow{e} x'$ predicates are generalisations of the transformations we previously described informally. This corresponds to the fact that an event e can occur within a process x and, after the execution of this event, what remains to be executed corresponds to the process x'.

Taking advantage of the definition of the $p \xrightarrow{e} p'$ transformations, we can formalize a small-step operational semantics as a function $\sigma_o : P(E) \to E^*$ such that:

$$\sigma_o(p) = \left\{ e.s \;\middle|\; \begin{array}{c} \exists\ p' \in P(E)\ s.t. \\ (p \xrightarrow{e} p') \wedge (s \in \sigma_o(p')) \end{array} \right\} \cup \left\{ \begin{array}{ll} \{\epsilon\} & \text{if } p\downarrow \\ \emptyset & \text{if } \neg(p\downarrow) \end{array} \right.$$

We can immediately see that the two approaches are quite different. The denotational approach, which is compositional is often more practical to prove formal properties on the semantics. By contrast, the operational approach is often more easily implementable in functional programming languages for various uses in software programs and FV tools.

2.5 The Coq proof assistant

Coq [39, 1] is a software tool and development environment in which one can: (1) write mathematical definitions, lemmas and theorems, (2) write machine-checked formal proofs of said lemmas and theorems, (3) write executable algorithms, machine-check their correctness against formal properties and theorems and (4) extract from those algorithms certified programs to various programming languages. As a result Coq is often described as a "formal proof management system", a "proof assistant", a "proof development system", an "interactive theorem prover", etc.

Coq has been used with success in both:

- the academic world, to prove important theoretical results such as the four-color theorem [62] or the Gödel-Rosser incompleteness theorem [105]
- and in the context of the software and hardware industry, for instance with the CompCert [81] compiler, which is a formally verified compiler used for programming embedded systems requiring reliability with a variant of the C programming language

Let us now adopt the perspective of a user of Coq. The main features of Coq can be used by writing and compiling ".v" files in which the user can encode the formal definitions of mathematical objects, theorems and programs, as well as proofs of theorems. Within those same files, code can be written using 2 syntactically distinct languages:

- "Gallina", which is used to define objects, declare their properties, state theorems and write algorithms

- another language (unnamed) which can be used in specific sections of the file so as to "machine-proof" theorems and properties that are either stated explicitly in Gallina or derived from programs written in Gallina (the "obligations"). Proving a theorem consists in obtaining a certain "goal" from a list of starting "hypotheses". Goals and hypotheses are logical formulae in intuitionistic propositional logic. Hypotheses and goals can be transformed step by step so as to obtain either a contradiction (in the hypotheses), or a tautology (the goal is also an hypothesis), in which case the proof is complete. To do so, Coq provides a number of "tactics" (instructions), which use is the object of this second unnamed language.

Even though both languages appear distinct from the point of view of the user, there is in facts only one single language that is processed by Coq : Gallina. The use of tactics in the "second language" are in facts instructions to build progressively a Gallina term. Those instructions are simply provided as a more user friendly manner to do so. The Gallina term that is thus obtained is a proof of the theorem to prove. Coq relies on the Curry Howard correspondence which states that a proof is a program, and the formula it proves is the type for the program. As a result, checking that a Gallina term is a proof of a theorem equates to checking its type.

2.5.1 Encoding the toy process algebra in Coq

In the following we use Coq so as to formally proof (in a machine-checked manner) the equivalence of the denotational and operational semantics that we proposed for the toy process algebra from Sec.2.4. Let us start by encoding the definition of the process algebra terms (the syntax) and of sequences of events (the data type of the semantic domain). The corresponding Gallina code is given on Fig.2.13.

```
Parameter E : Set.

Definition Sequence : Type := list E.

Inductive PATerm : Set :=
  empty_pa : PATerm
  | event_pa : E → PATerm
  | seq_pa:PATerm→ PATerm→ PATerm
  | alt_pa:PATerm→ PATerm→ PATerm.
```

Figure 2.13: Definitions of the data types for the syntax and semantics of our toy process algebra in Gallina

At first we declare the set E of events, which constitutes the signature of the process algebra language. To do so, we declare it as a `Parameter` of type `Set`. The declaration of any object in Coq starts with a keyword (here `Parameter`) and ends with a dot (`"."`).

Then we define a data type for sequences of such events with the keyword `Definition`. Let us call it `Sequence`. We declare it to be of type `Type` and equal to a list of elements of E i.e. `list E` (with a dedicated library imported using `Require Import List`).

We then define a data type for the terms of our process algebra. Let us call this type `PATerm`. Given the inductive nature of the process algebra language, we declare it in Coq using the `Inductive` keyword, and,

given that we want to define a set of inductively constructed terms, we declare the type of `PATerm` to be `Set`. Then we must specify under which rules can those inductive terms be constructed. There are 4 rules, which correspond to 4 constructors:

- `empty_pa`, which corresponds to the empty process ϵ
- `event_pa`, which corresponds to processes reduced to a single event $e \in E$. The corresponding constructor is `event_pa` : `E` $\rightarrow$ `PATerm`, meaning that its type is a function which takes an `E` and returns a `PATerm`
- In Coq, all functions can be curried i.e. instead of having a function $f(x, y)$, applied to arguments x and y, we have $f(x)(y)$ i.e. a function f applied to x, which turns into a function $f(x)$ which is then applied to y. In particular, constructors with multiple arguments are declared in this fashion. This is the case for the third constructor: `seq_pa`:`PATerm`$\rightarrow$ `PATerm`$\rightarrow$ `PATerm`, which corresponds to the "." operator of our process algebra, which forms terms of the form "$x.y$", where "x" and "y" are two arguments of type `PATerm`. We named it `seq_pa` to express the fact that it corresponds to a sequential composition.
- likewise, we declare the constructor corresponding to the "+" operator of the process algebra with: `alt_pa`:`PATerm`$\rightarrow$ `PATerm`$\rightarrow$ `PATerm`, its named referring to the alternative choice it models.

The next step is to encode the predicates $\downarrow$ and $\rightarrow$ of the process algebra. We provide the corresponding Gallina code on Fig.2.14.

The $\downarrow$ predicate can be understood as a recursive function that takes as argument a `PATerm` and returns a `Prop` i.e. a proposition in intuitionistic propositional logic (this proposition corresponding to whether or not the given process algebraic term terminates or not). In Gallina, we use the `Fixpoint` keyword to declare the `terminates` function which takes an argument `p` or type `PATerm`. We then use a pattern-matching notation (similar to that of many functional programming language e.g. ML, OCaml, Rust, Haskell, etc.) so as to define the cases of the induction on the term structure of the argument `p`. Here, each case of the induction is introduced by the | symbol, followed by the structure of `p` in that case. Then the symbol $\Rightarrow$ indicates the value the recursive function is supposed to return in that case. Given that the return type of `terminates` is `Prop`, we return:

- `True` for the empty process ϵ i.e. `empty_pa`
- `False` for any process that corresponds to a single event e i.e. (`event_pa e`)
- (`terminates p1`) $\wedge$ (`terminates p2`) for any process that corresponds to a sequence of sub-processes $p_1.p_2$ i.e. (`seq_pa p1 p2`)
- (`terminates p1`) $\vee$ (`terminates p2`) for any process that corresponds to an alternative between sub-processes $p_1 + p_2$ i.e. (`alt_pa p1 p2`)

Let us note that, in Coq, any recursive function must be able to terminate. This requirement is here to ensure the consistency of the logical proof system provided by Coq (otherwise "false statements" could be proven). Behind the scenes, Coq tries to find a measure on the arguments that is strictly decreasing at each recursive call. If no such measure is found, the code does not compile.

```
Fixpoint terminates (p : PATerm) : Prop :=
  match p with
    | empty_pa      ⇒ True
    | (event_pa e)  ⇒ False
    | (seq_pa p1 p2) ⇒ (terminates p1) ∧ (terminates p2)
    | (alt_pa p1 p2) ⇒ (terminates p1) ∨ (terminates p2)
  end.

Inductive is_next_of : PATerm → E → PATerm → Prop :=
|next_event : forall (e:E),
                (is_next_of (event_pa e) e empty_pa)
|next_seq_left : forall (e:E) (p1 p2 p1' : PATerm),
                (is_next_of p1 e p1')
                  → (is_next_of (seq_pa p1 p2) e (seq_pa p1' p2))
|next_seq_right : forall (e:E) (p1 p2 p2' : PATerm),
                ( (is_next_of p2 e p2') ∧ (terminates p1) )
                  → (is_next_of (seq_pa p1 p2) e p2')
|next_alt_left : forall (e:E) (p1 p2 p1' : PATerm),
                (is_next_of p1 e p1')
                  → (is_next_of (alt_pa p1 p2) e p1')
|next_alt_right : forall (e:E) (p1 p2 p2' : PATerm),
                (is_next_of p2 e p2')
                  → (is_next_of (alt_pa p1 p2) e p2').
```

Figure 2.14: The ↓ (`terminates`) and → (`is_next_of`) predicates of the toy process algebra in Gallina

The → predicate of the process algebra can be understood as an inductive predicate i.a. a predicate that can be inferred inductively. A certain $x \xrightarrow{e} y$ is a proposition that can be obtained from an object of type `PATerm → E → PATerm → Prop`. As a result, the Gallina definition that we propose uses the `Inductive` keyword to define a predicate `is_next_of` of type `PATerm → E → PATerm → Prop`.

This definition relies on a number of rules which can be used to infer a certain (`is_next_of x e y`). Those rules correspond to those of the process algebra and their definitions are the exact translation of the mathematical definitions from Sec.2.4:

- `next_event`, i.e. the rule $e \xrightarrow{e} \epsilon$ for any $e \in E$. This is encoded in Gallina as a proposition `forall (e:E), (is_next_of (event_pa e) e empty_pa)`

- `next_seq_left` i.e. the rule $(x \xrightarrow{e} x') \Rightarrow (x.y \xrightarrow{e} x'.y)$. This corresponds (in Gallina on Fig.2.14) to `forall (e:E) (p1 p2 p1' : PATerm), (is_next_of p1 e p1') → (is_next_of (seq_pa p1 p2) e (seq_pa p1' p2))` Let us note here that, in Gallina, the → symbol can both be used to denote a type (of a function) and to denote an implication (as is the case here). This is in line with the Curry-Howard correspondence, which states that an implication $P \to Q$ is in facts the type of functions that can transform proofs of P into proofs of Q. Given that remark, and the previous on currying, implications can also be curried e.g. with $P \to Q \to R$ instead of $(P \wedge Q) \to R$.

- likewise, we have the same kind of encoding for the 3 other rules

2.5.2 The semantics of the toy process algebra in Coq

The operational and denotational semantics proposed in Sec.2.4 can then be formalized in Coq as illustrated on Fig.2.15:

- the operational semantics, by an inductive predicate `sem_op : PATerm → Sequence → Prop`, with two rules `sem_op_empty` and `sem_op_event` corresponding to the base case (whether or not the empty sequence is in the semantics, via the $\downarrow$ predicate) and the small-step of the semantics i.e. the execution of events (via the $\rightarrow$ predicate)

- the denotational semantics, by a recursive function which returns a set of sequences:
  `sem_de (p : PATerm) : (Sequence → Prop)`. The return type corresponds to checking whether or not a given `Sequence` belongs to the set or not (the membership is a proposition). The function is defined inductively w.r.t. the structure of the `p` process in argument. We use pattern-matching to explicit the cases.

```
Inductive sem_op : PATerm → Sequence → Prop :=
 | sem_op_empty : forall (p :PATerm),
                     (terminates p)
                     → (sem_op p nil)
 | sem_op_event : forall (p p':PATerm) (e:E) (s:Sequence),
                     (is_next_of p e p')∧ (sem_op p' s)
                     → (sem_op p (cons e s) ).

Fixpoint sem_de (p : PATerm) : (Sequence → Prop) :=
 match p with
    | empty_pa      ⇒ fun s:Sequence ⇒ s = nil
    | (event_pa e)  ⇒ fun s:Sequence ⇒ s = e :: nil
    | (seq_pa p1 p2) ⇒ fun s:Sequence ⇒ exists (s1 s2 : Sequence),
                                          (sem_de p1 s1) ∧ (sem_de p2 s2) ∧ (s = s1 ++s2)
    | (alt_pa p1 p2) ⇒ fun s:Sequence ⇒ (sem_de p1 s) ∨ (sem_de p2 s)
  end.
```

Figure 2.15: Formalization of the semantics proposed for the toy process algebra in Gallina

2.5.3 Proving the equivalence of both semantics with Coq

Now that we have formalized in Coq both our semantics, let us prove their equivalence. The plan of the proof is represented on Fig.2.16. The theorem that we want to prove corresponds to the blue node on the top. Its formulation is $\forall\ p \in P(E),\ \forall\ s \in E,\ s \in \sigma_o(p) \Leftrightarrow s \in \sigma_d(p)$. It is immediately implied by the two sub-theorems (blue nodes underneath) which correspond to the two directions of the inclusion.

Let us consider the inclusion of σ_o in σ_d, which corresponds to the part of the proof that is on the left on Fig.2.16. This theorem can be proven thanks to 2 intermediate lemmas given in the teal nodes underneath it and which correspond to:

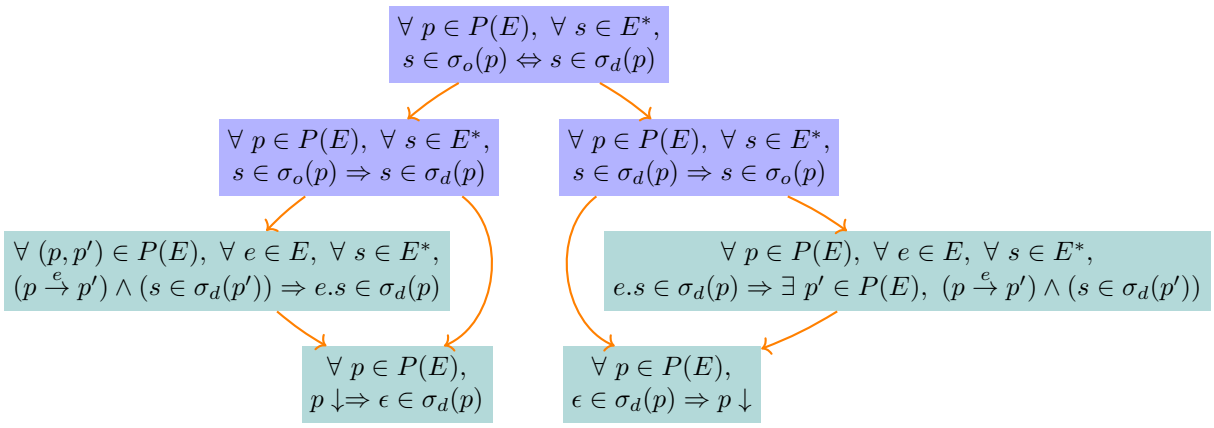


Figure 2.16: Plan of the proof for the equivalence of the semantics of the toy PA

- the fact that if a process p can immediately terminate (the predicate $p \downarrow$) then $\sigma_d(p)$ includes the empty trace ϵ
- the fact that if a process p can execute event e, which turns it into process p', and, that there is a certain trace $s \in \sigma_d(p')$ then $e.s \in \sigma_d(p)$

Let us remark that those two lemmas correspond to the inductive definition of the operational semantics σ_o. In the following, we detail the proof of the first lemma, which is given as Lem.2.1.

Lemma 2.1: p directly terminates implies $\sigma_d(p)$ accepts ϵ

For any process $p \in P(E)$:

$$p \downarrow \Rightarrow \epsilon \in \sigma_d(p)$$

Proof. Let us reason by induction on the term structure of p.

- if $p = \epsilon$ then by definition $p \downarrow$ and $\epsilon \in \sigma_d(p)$
- if $p = e$ for a certain $e \in E$ then we cannot have $p \downarrow$
- if $p = p_1.p_2$ for given $(p_1, p_2) \in P(E)$ then $p \downarrow$ iff $p_1 \downarrow$ and $p_2 \downarrow$ and by the induction hypothesis we have $\epsilon \in \sigma_d(p_1)$ and $\epsilon \in \sigma_d(p_2)$. Then, given that $\sigma_d(p) = \{s_1.s_2 \mid s_1 \in \sigma_d(p_1),\ s_2 \in \sigma_d(p_2)\}$, we have $\epsilon \in \sigma_d(p)$
- if $p = p_1 + p_2$ for given $(p_1, p_2) \in P(E)$ then $p \downarrow$ iff $p_1 \downarrow$ or $p_2 \downarrow$:
  - in the first case, the induction hypothesis gives $\epsilon \in \sigma_d(p_1)$, and given that $\sigma_d(p_1) \subset \sigma_d(p_1 + p_2)$, we have $\epsilon \in \sigma_d(p)$
  - likewise, in the second case the induction hypothesis gives $\epsilon \in \sigma_d(p_2)$, and given that $\sigma_d(p_2) \subset \sigma_d(p_1 + p_2)$, we have $\epsilon \in \sigma_d(p)$

□

The transcription of this lemma and of its proof in Coq is given on Fig.2.17. The definition of the Lemma contains the predicate (`terminates p`) → (`sem_de p nil`) which is the exact translation of the Lemma's mathematical definition, with the curried expression (`sem_de p nil`) representing the inclusion of the empty sequence `nil` into the semantics applied to process `p`. The proof is introduced by the keyword `Proof`. Then, we use a number of tactics to transform the goals and hypotheses and resolve the intermediate sub-goals until the completion of the proof. The first tactic `intros` introduces the left part of the goal formulation as an hypothesis named `H`. The application of this tactic results in the declaration of a hypothesis `H : terminates p`. The second tactic `induction` performs an induction on the term structure of `p`. This changes the proof into 4 sub-proofs, which can be individually solved. The proof of each sub-proof is introduced by a dash.

- in the first sub-proof, we have the hypothesis `H : terminates empty_pa` and the goal `sem_de empty_pa nil`. The tactic `simpl` simplifies the goal (as per the definition of `sem_de`) into `nil = nil`, which can be immediately proved with the tactic `reflexivity`
- in the second sub-proof, we have the hypothesis `H : terminates (event_pa e)`, with a certain `e : E` and the goal `sem_de (event_pa e) nil`. The tactic `simpl` is here used on the hypothesis `H` to reveal a contradiction (it transforms the hypothesis into `H : False`). Then we use the tactic `contradiction` to resolve the sub-proof
- in the third sub-proof, we have several hypotheses:
  - `H : terminates (seq_pa p1 p2)`, which is the hypothesis on the overall term `p`
  - `IHp1 : terminates p1 → sem_de p1 nil`, which is the induction hypothesis on the left sub-term `p1`
  - `IHp2 : terminates p2 → sem_de p2 nil`, which is the induction hypothesis on the right sub-term `p2`

  The goal is `sem_de (seq_pa p1 p2) nil`, which we can simplify with the `simpl` tactic to obtain
  `exists s1 s2 : Sequence, sem_de p1 s1 ∧ sem_de p2 s2 ∧ nil = s1 ++s2`
  Given that we know (from human experience), which are the expected values for the sequences `s1` and `s2`, we can use the `exists` tactic to declare them (both are `nil`), which result in the goal `sem_de p1 nil ∧ sem_de p2 nil ∧ nil = nil ++nil`. We can transform our hypothesis to obtain the 2 clauses on the left. To do so we use `simpl in H`, which turns `H` into `terminates p1 ∧ terminates p2` which we can separate into two hypotheses `H1` and `H2` with the `destruct` tactic. We then use the `apply` tactic to apply the induction hypotheses `IHp1` and `IHp2` on respectively `H1` and `H2`. The third clause of the goal is a tautologic equality, which can be resolved with the `reflexivity` tactic. Therefore, we have all the ingredients to verify our goal. We only need to divide and destroy it. To do so, we use the `split` tactic, and treat each case as a sub-goal. The `assumption` tactic is used to prove a goal that is already given as an hypothesis.
- in the fourth sub-proof, we have the hypotheses:
  - `H : terminates (alt_pa p1 p2)`, which is the hypothesis on the overall term `p`

- `IHp1 : terminates p1 → sem_de p1 nil`, which is the induction hypothesis on the left sub-term `p1`
- `IHp2 : terminates p2 → sem_de p2 nil`, which is the induction hypothesis on the right sub-term `p2`

The goal is `sem_de (alt_pa p1 p2) nil`, which we can simplify with the `simpl` tactic to obtain

`sem_de p1 nil ∨ sem_de p2 nil`

Likewise, we can simplify the `H` hypothesis to obtain `terminates p1 ∨ terminates p2`. We have here a disjunction as an hypothesis. We can use the `destruct` tactic on `H` so as to treat the two cases (either `terminates p1` or `terminate p2`) as two sub-proofs (with different sets of hypotheses but the same goal).

- in the first one, we have `H : terminates p1`. We use our human intuition to select the clause of the goal (a disjunction) that we want to prove. It is the left clause and we can use the `left` tactic to select it. By applying the induction hypothesis `IHp1` the result is trivial.
- the other case is similar, except we choose the `right` clause and `apply IHp2`

```
Lemma terminates_implies_de_accept_empty (p : PATerm) :
  (terminates p) → (sem_de p nil).
Proof.
intros H.
induction p.
- simpl. reflexivity.
- simpl in H. contradiction.
- simpl. exists nil. exists nil.
  simpl in H. destruct H as (H1,H2).
  apply IHp1 in H1. apply IHp2 in H2.
  split.
  + assumption.
  + split.
    * assumption.
    * simpl. reflexivity.
- simpl. simpl in H. destruct H.
  + left. apply IHp1. assumption.
  + right. apply IHp2. assumption.
Qed.
```

Figure 2.17: Proof of $\forall\ p \in P(E),\ p \downarrow\Rightarrow \epsilon \in \sigma_d(p)$ with Coq

Now that we have proved all our goals (Coq notices the user of that with `No more subgoals.`), we can close the `Proof` with the `Qed` keyword. The proof is correct by construction, given that, behind the scenes, Coq uses the tactics to construct a functional program in Gallina, which constitutes a proof of the proposition, as per the Curry-Howard correspondence. The `terminates_implies_de_accept_empty` lemma is now an established fact in the remainder of the ".v" Coq file, and we will be able to use it (for instance with `apply terminates_implies_de_accept_empty` in other proofs).

The whole proof for the equivalence of σ_o and σ_d on this toy process algebra is available in [89]. In it, we use additional tactics, including:

- `dependent induction`, which is an extension of the `induction` tactic which makes possible to generalize some variables of which the induction hypothesis found in the resulting sub-goals depend. With the

`generalizing` keyword, we can specify which variables we want to generalize (although it is not always possible to generalize a variable)

- `inversion`, which, applied to an inductive predicate derives all the possible conditions (may transforms the initial set of hypothesis into several others, each corresponding to a new sub-proof) for it to be true according to its constructors.

An overview / a documentation of the tactics available in Coq's standard library is available in [2].

By detailing a Coq proof for this toy process algebra, we have only scratched the surface of can be done with Coq. However it is an introduction that should suffice to understand the more complex proofs proposed in the thesis and which were encoded in Coq. In particular this proof of semantic equivalence for the toy process algebra [89] is a miniature version of the proof available on github in [88] in which we prove the equivalence of three semantics (that from Chap.4, Chap.5 and Chap.6) defined over the Interaction Language which we define in Chap.4 of this thesis. Another proof, pertaining to the correctness of the multi-trace analysis algorithm from Chap.9 is available on github in [87].

Conclusion

In this chapter we have presented some elements of context for the works of this thesis. In particular:

- Sec.2.1 and Sec.2.2 pertaining to formal languages and term algebras will be particularly useful in Chap.4 in which we define our IL, its denotational-style trace semantics and a process to normalize interaction terms and in Chap.7 where we extend the notion of the semantics of interactions to multi-traces with various multi-trace semantics.
- Sec.2.3 serves as an introduction to Interaction Languages which is one of the main focus of the entire thesis
- Sec.2.4, by introducing process calculus is a prelude to Chap.5 where we define a structural operational semantics for our IL
- finally, with the presentation of Coq and of a miniature proof for semantic equivalence in Sec.2.5 we foreshadow the proof from Sec.5.2 as well as various other proofs presented in this thesis (Sec.6.2, Sec.9.2) which are encoded in Coq ([88, 87])

In the following chapter, so as to anchor the present thesis in the state of the art, we present a quick survey of formal semantics for ILs that are found in the literature.

Chapter 3

On the semantics of Interaction Languages

Contents

In this chapter we discuss the state of the art on the definition of formal semantics for Interaction Languages.

The plan of this chapter is as follows:

- in Sec.3.1 we review a short selection of papers and we discuss how they relate to our work,
- in Sec.3.2 we give a broader and shallower survey on the formal semantics of ILs,
- finally, in Sec.3.3 we give some concluding remark and position the works of this thesis with regards to the state of the art.

3.1 A discussion on a selection of papers

In this section we discuss a selection of papers from the literature pertaining to the formalisation of the semantics of some ILs. We have chosen those papers to represent a wide variety of formalisms and different approaches to the definition of their corresponding semantics.

3.1.1 An algebraic semantics of BMSCs

In [97] an algebraic semantics of BMSCs is proposed. It relies on corresponding the syntax of BMSCs to process algebraic terms and then simplifying those terms, using a dedicated "state operator" [31] λ_M to ensure the respect of the emission-reception causality. In [97] the semantics of BMSC is a "free merger" (i.e. an interleaving) of that of its constituant instances (lifelines). Some transformations, constrained by λ_M can be applied so as to progressively transform those interleavings into sequences and alternatives. The end result is a process algebraic term which only consists of alternatives and sequences and can therefore be directly mapped to a set of accepted sequences.

The syntax of BMSCs in [97] is quite different from that of our own IL given that each instance is described independently as a list of atomic events and it is then the state operator λ_M which (by remembering which messages have been send and not yet received) links reception events to corresponding emission events. The manner in which this λ_M operator is used causes some limitations to the formalism. In particular, it is not allowed that two distinct outputs or inputs have the same message name and message instance name. By contrast, our approach handles the issue of the emission-reception causality via the use of a strict-sequencing operator which can schedule the executions of two sub-interactions so that the first occurs strictly before the second. By applying this operator to an atomic emission (on the left) and an atomic reception of the same message (on the right), we can in effect implement message passing, with no restrictions on the number of occurrences of said message. This remark is also linked to the fact that atomic emissions in [97] must specify which is the target instance, and atomic receptions must specify which is the source instance. By contrast, in our formalism, this information do not appear in individual events (as should be the case in all generality, when we consider communications in a distributed context). When a message is observed i.e. logged somewhere in the network, the only information that is available is the data it carries (and the local

time of the reception event). It is not a given that it carries information pertaining to its original sender, and, even if it is the case, it could be false (e.g. man-in-the-middle attack).

[97] also details a proposed semantics for a "create" and "stop" atomic events, which correspond to the creation and deletion of temporary instances. This is handled similarly to the passing of messages, with a dedicated λ^L operator. λ^L keeps track of which instances have been created (and later deleted) and does not allow the execution of events that concern an instance that does not exist. The semantics is likewise extended to include timers. By contrast, in our formalism we only consider a core language and we mainly focus on the different scheduling operators that can be defined to specify exchanges of labelled message within a statically defined distributed system (no creation or deletion of sub-systems). Our aim is to use this core language as a platform to experiment on the definition of different semantics. Those semantics can then be compared and their equivalences proven. Moreover, the addition of new primitives (creation/deletion, timers, etc.) could be the object of further works.

The approach proposed in [97] could easily be implemented into a FV tool given the nature of its models, which are finite terms and describe a finite set of finite executions (no repetitions). To obtain the semantics as a set of sequences, it suffices to process the simplified process algebraic terms and eliminate the alternative operators. To go further, the difficulty lies in having models which allow behaviors to be repeated, which may result in arbitrarily long behaviors. In this thesis, we tackle this with dedicated loop operators.

3.1.2 High-level message sequence charts

In [98], a formal syntax and semantics is proposed for HMSC. This approach is an extension of [97] which includes additional operators for "weak sequencing" and "delayed choice" so as to include the increased expressivity of HMSC w.r.t. BMSC.

The weak sequencing of two BMSCs "A" and "B" specifies a partially parallel executions of the behaviors specified by each. In any such execution any event e from "B" can only occur if it is permitted by "A" by a "permission relation". This permission relation refers to the ability of "A" of not expressing actions on the same instance (lifeline) on which e occurs. As a result, weak sequencing allows to compose BMSCs with more flexibility than the classical (strict) sequencing operator. As in [97], in [98] HMSCs are associated to process algebraic terms. In particular, the definition of the permission relation is done inductively on those terms. The delayed choice operator is quite similar to the classical alternative operator, except that the choice of the alternative is delayed as long as possible. This means that whenever the same event can occur in both branches of an alternative then it is executed in both and the choice of which alternative has been taken is delayed.

The semantics of HMSC in [98] is defined in two parts. At first a semantics is defined for finite paths within the graph-like structures that a HMSC constitutes. This semantics is defined similarly as that of [97] with the inclusion of weak sequencing and delayed choice.

The semantics of a HMSC, as proposed in [98], is then defined as follows:

- if the HMSC is reduced to a single BMSC then its semantics is defined like that of [97]. Let us note that, in an individual BMSC, the emission-reception causality is enforced by a dedicated state-operator (as per [97]). As a result, within a HMSC, a reception in a given node cannot be interpreted as corresponding to an emission from another node.

- else it is given as a denotation consisting of a set of equations in which variables correspond to the semantics of individual nodes and symbols correspond to interleaving (between child nodes in a fork of the graph), weak sequencing (between two successive nodes in the graph) or delayed choice (between child nodes in a split of the graph).

Discussion w.r.t. the thesis

The structure of the HMSC models studied in [98] is quite different from that of our models. HMSCs are graph structures whereas our IL is an inductive language which terms can be understood as finite binary trees.

Moreover, the semantics proposed in [98] consists in translating a HMSC into a set of equations on process algebra terms. The semantics is then obtained from those terms. By contrast, in our operational semantics, the small-steps operate directly on the syntactic terms that constitute our models (as a by-product, this enables the dynamic animation of models during execution).

In [98], process algebra terms can contain references. Those references incidentally serve as a means to represent repeatable behaviors. In other words repetition is handled by process algebraic terms referencing themselves directly or indirectly. By contrast, in our IL, repetition is handled by dedicated loop operators. As a result, the syntactic structure of a term which defines repeatable behaviors in our language remains a finite and self-contained (no references) binary tree.

Although both formalisms are quite different, we can find many correspondences between the notions found in [98] and those found in our thesis. The basic sequence-diagram-like terms are:

- the "empty process" ϵ from [98] corresponds to our empty interaction $\varnothing$

- however, we do not have anything similar to the "deadlock" δ from [98]. In our approach, when analyzing the conformity of a trace against an interaction model, we can "be deadlocked" when the next event to analyze cannot be executed by the model (is not found within the model's "frontier of execution").

- atomic models which correspond to atomic events are emissions and receptions of messages in both formalisms. However, [98] specifies which is the target instance in emission events, and which is the source instance in reception events. By contrast, in our formalism, this information do not appear in individual events. In all generality, when we consider communications in a distributed context it is the case, otherwise

- the "permission relation" defined in [98] fulfills the same purpose as our evasion predicate (for determining whether or not an event is executable) and our pruning relation (for constructing a remaining "follow-up" term), which are defined as part of our operational semantics in Chap.5

In the small-step operational semantics that we propose in this thesis, we take advantage of having models as finite binary trees so as to be able to compute, each time an event is observed, what is the "next" model, which specifies "what remains to be executed" (which is another, rewritten finite binary tree). This small-step approach in turns allowed us to implement a tool for the animation of interaction models. In the approach proposed by [98], HMSCs are translated into systems of equations between process algebraic terms. However, no method is detailed as how to process those equations.

3.1.3 Operational Semantics for MSC

In [99], an operational semantics is defined for a subset of MSC. This subset of MSC includes the exchange of messages, actions on individual lifelines, the inline-expressions for parallel and alternative composition, co-regions and explicit causal ordering. It does not however include references (and gates), conditions, the "optional" and "exception" expressions and any loop operator.

In [99], although no formal syntax is provided for encoding MSCs, two manners of doing so are described:

- an "instance-oriented" textual representation, which is similar to that of [97] and which consists in describing one-by-one the instances of the MSC (which is quite different from the syntax of our IL).

- an "event-oriented" textual representation, which lists all the building blocks (events, inline-expressions) that constitute the MSC in an order which is that of the top to bottom order of the corresponding diagram (which is somewhat more similar to the syntax of our IL).

The first step in the approach from [99] consists in transforming an MSC described via an "event-oriented" textual representation into a process algebra term. This process is not described in [99]. Instead, the process algebra is directly described and its operational semantics given. Let us note that [99] is written by the same authors as [97] and [98] and that the process algebra that is provided is based on that of [98].

Instead of relying on a state operator as in [97] to enforce an order between the occurrences of events, [99] proposes a more generalizable approach. It consists in the attribution of some ordering requirements to the operators themselves. Those requirements concern the subterms on which the operators are applied. In more practical terms, this consists in attaching mappings to operators which keep track of constraints that must be enforced in relation to the precedence of events. Those mappings may then be checked whenever one needs to verify whether or not an event is executable and may be updated during execution. Indeed, during execution an event that is executed disappears from the term and must be removed from the mappings in which it appears as a precondition to the execution of some other events.

Discussion w.r.t. the thesis

The operational semantics which we define in Chap.5 for our IL shares some similarities with that of [99] which include:

- the general principle of the language with basic building blocks being emission and reception events and constructors being used to express sequencing, interleaving and choice
- the use of a pruning relation which, in [99] is called a permission relation

However we have some major distinctions concerning:

(i) which constructors are defined: we propose the addition of the four loop constructors while there were no loops in [99] and, in addition, there is no *strict* constructor in [99] (this also relates to point (iii))

(ii) how the alternative and the parallel composition are handled: in [99] they use a delayed choice constructor which allows to delay the choice between two alternative branches whenever an action can be executed in both branches. In that case, the follow-up term is the alternative composition of the follow-ups of the two sub-interactions. This also effects parallel composition given that whenever an action can be executed in both parallel branches we can define the follow-up interaction as a choice between interpreting the action as having been executed in either the left or the right branch of the parallel. In our IL and our semantics however we have not proposed this mechanism for three reasons:

- in Chap.6 we will see how we have "algorithmicized" this operational semantics for it to be easily and efficiently implemented and further used in Formal Verification (FV) techniques. This algorithmization relies on separating concerns between the determination of which actions are executable and the execution of said actions. Immediately executable actions are uniquely identified by their positions in the term structure. The execution function then corresponds to the execution of a specific action at a specific position. If we used delayed choice we would need to determine which identical actions at which positions can safely be executed simultaneously and propose an execution function to execute several actions at the same time in the model.
- in Chap.14, we will see how we have, in our tool implementation HIBOUX, added data in interaction models and how we complemented the operational semantics to include symbolic execution in order to treat the modification and exchange of typed data. Without delayed choice, the execution of an action is uniquely defined and uniquely modifies the symbolic interpretation of variables and the symbolic path condition of the system. Each intermediate interaction i is associated to a unique interpretation η and a unique path condition π to form a state (i, η, π) of the system. Given that several instances of a same

action within the model can be associated to different statements (e.g. assignment of variables), simultaneously executing them in the model would yield the same interaction term i but different symbolic states $(\eta_1, \pi_1), \cdots, (\eta_n, \pi_n)$. As a result, if we had added delayed choice this would have required to treat more complex states.

- delayed choice can be used to minimize the exploration of different paths of executions in the model. Indeed, roughly speaking, instead of exploring $alt(a_1, seq(a_1, a_2)) \xrightarrow{a_1} \varnothing$ and $alt(a_1, seq(a_1, a_2)) \xrightarrow{a_1} a_2$, we simply explore $alt(a_1, seq(a_1, a_2)) \xrightarrow{a_1} alt(\varnothing, a_2)$. This could be useful in FV techniques to diminish overhead when exploring the semantics of models. However we will see in Chap.10 that, with "local frontiers", we can provide mechanisms that offset this disadvantage of not using delayed choice.

(iii) how strict sequencing and the emission-reception causality is handled. Indeed, we use a dedicated *strict* constructor to generalize causality between actions occurring on different lifelines while [99] use mappings between causally-related events that are updated at each step of execution.

(iv) how the weak sequencing is handled: indeed, due to the differences in how causation is handled (with maps that are updated) and because of the use of delayed choice the operational rules corresponding to weak sequencing that are proposed in [99] are quite different from our own.

3.1.4 UML Interactions Meet State Machines – An Institutional Approach

In [75], a simplified version of UML-SDs (interactions) is formalised as an institution. In this institution, the signature of an interaction is defined by a set of lifelines and a set of messages. Interactions defined over this signature are then sentences build by composition from basic building blocks which may consist of the empty interaction or communication events (4 kinds: emissions with and without explicit targets and receptions with and without explicit sources) using some operators which are strict and weal sequencing, interleaving and alternative composition.

The semantics of such interactions is then formulated in denotational style in the form of sets of event traces i.e. sequences of events. The semantics of a complex interaction is build by composition from those of its subterms using some algebraic operators on sets of event traces. In particular, the operator for weak sequencing uses a notion of conflict so as to determine which interleavings are allowed and which are not.

Discussion w.r.t. the thesis

The syntax of our IL and the denotational formulation of its corresponding trace semantics in Chap.4 is inspired by and extends that of [75] with the addition of loops.

3.1.5 Global and Local Testing from Message Sequence Charts

In [83], a semantics for a subclass of HMSC is proposed. This subclass of HMSC is Message Sequence Graph (MSG). It consists in HMSC models where every node must contain a BMSC (i.e. it cannot contain another nested HMSC). Moreover only closed systems are considered, i.e. messages cannot be sent to or received from the environment; all exchanges have to occur between the sub-systems of the distributed system.

[83] proposes a denotational semantics for MSG. Its general principle is to correspond individual BMSCs to partial orders sets (posets) which synthesize precedence relations between events. Every sequence of events that the BMSC specifies in intention can be obtained via a linearization of a partial order from the BMSC's corresponding poset. By linearization we mean to consider sequences of events such that each appears exactly once in the partial order and such that their relative order of occurrence, as specified by the partial order is respected. The semantics of a BMSC can then be understood as the set of all such linearizations. Then, in order to define the semantics of MSGs, it suffices to consider all finite paths within the graph, all of which being associated to a BMSC which is the "concatenation" (using weak sequencing) of all the BMSC nodes encountered in the path.

In addition to providing a denotational-style semantics for MSGs, [83] also discusses testing. This paper proposes a notion of global conformance between a system and its MSG specification. A system is conform to a MSG iff all the traces (sequences of events) it can express correspond to a prefix of a linearization of the MSG (i.e. the "semantics" we referred to). [83] then proposes a method for testing specific words (traces) via the definition of an automaton which execution is concomitant to the execution of the test and terminates in either a "pass", "fail" or "inc" (inconclusive) state. Those automata can be used to test well-founded words i.e. words in which each reception event is preceded by a corresponding emission event (it is not sufficient to be well-founded, words must be well-founded and conform to the MSG specification).

A test set, which includes tests for all words diverging from prefixes of linearizations of the MSG is then proposed and proven to suffice to imply the conformance relation. Finally, a technique to factorize such tests is informally described. This technique allows the obtention of a single automaton to test the system. This automaton has a tree-like structure and allows the off-line analysis of execution traces. Elements from the trace can be analyzed via the execution of this automata, each transition corresponding to the consumption of an event in the trace.

In [83], no formal methodology is given for the realisation of those automata. Moreover, the provided example only concerns a single BMSC and not a MSG. Among other things, this raises the question of what happens to the automaton when dealing with a cyclic MSG. Given that some behaviors can be repeated any number of times within a cyclic MSG, this would result in an automaton with infinitely many states.

[83] also discusses some considerations about local testing i.e. testing the system from the point of view of every local process. This discussion relies on two important notions, which are defined on MSG specifications:

- "local-synchronicity", which roughly means that there cannot be arbitrarily long one-sided exchanges between two processes (for instance in a MSG which loops back around a BMSC consisting of a message

passing, there can be infinitely many emissions of the message before any corresponding reception, so the MSG is not locally-synchronised).

- "k-testability", which is defined for any $k \in [1, n]$ with n the number of processes in the modelled DS. A MSG model is 1-testable if it is not possible to reconstruct wrong BMSCs from independent local observations of correct BMSCs obtained from paths within the graph. The notion of "k-testability" is an extension of "1-testability" where, instead of projecting the model on single processes, we project them on subsets of processes of size k.

Different notions of local testing are proposed:

- "Local testing with Local Observations", in which each sub-system produces its own local trace which is then analyzed locally w.r.t. a projection of the specification (this local test translates into an automaton as in the global test seen earlier). Information is not shared between local testers so this conformance relation only denotes a conjunction of local tests (all must pass), but this does not guarantees that local traces can be gathered are reordered globally into a trace that satisfies the global conformance. Such a guarantee can only be ensured when the specification respects a "closure condition", which is very restricting. This condition is that the MSG specification is locally-synchronised and "1-testable".

- "Local Testing with Gathered Observations", in which the local traces produced by several sub-systems are gathered and analyzed together w.r.t. the specification. According to the number $k \in [1, n]$ of such local traces that are gathered, the paper defines a notion of "k-local conformance". [83] then generalizes results obtained for "1-local conformance" to "k-local conformance" and concludes that "k-local conformance" is equivalent to the global conformance iff the MSG is locally synchronized and "k-testable".

Local conformance is most generally a weaker property than global conformance. However, this last result of [83] implies that for $k = n$ (with n the number of processes), and for locally-synchronized distributed systems, local testing with gathered observations is equivalent to global testing. Indeed, n-testability is always true for an MSG.

Discussion w.r.t. the thesis

In the second part of the thesis, we define various multi-trace analysis algorithms. Those in fact perform a form of off-line testing which can be related to that found in [83]. In [83], the analysis of a trace corresponds to having it being recognized by a test automaton generated from the MSG specification. This is somewhat similar to the analysis graphs that are central to the definition of our multi-trace analysis algorithms. The main differences between those two approaches being that:

- [83] presupposes that the trace to analyze is well-founded (every reception is preceded by a corresponding emission) and that events are uniquely defined (no degeneracy), while our approach requires no such hypothesis on the nature of the trace (it can be any sequence of events).

- the nature of our analysis graph is quite different, with each vertex containing an intermediate multi-trace which is the multi-trace which "remains to be analyzed" at that point in the analysis and an intermediate interaction model which is the continuation of the executable interaction specification

- our analysis graphs can be computed on-the-fly given that, from any given vertex, the information to compute neighboring vertices is entirely self-contained (the current interaction model and what remains of the multi-trace to analyze). This allows in our approach to analyse arbitrarily long multi-traces against models which allows repeatable behaviors (with loops).

In [83], there are 3 different verdicts: "pass", which is returned when the trace is recognized to be a prefix of a trace specified by the MSG, "fail", which is returned when there is a deviation from said specification, and "inc" otherwise. With the various algorithms defined in this thesis in Chap.9, Chap.10 and Chap.11, we allow the identification and discrimination between multi-traces which might correspond to exactly accepted multi-traces, projections of prefixes of accepted traces, prefixes (in the sense of multi-traces) of accepted multi-traces or slices (i.e. "sub-words" in the sense of multi-traces) of accepted multi-traces.

The analysis algorithms defined in this thesis may take as input generalized multi-traces i.e. sets of sequences of events, each occurring on a subset of lifelines called a co-localization. This positions our algorithms in the case of "Local Testing with Gathered Observations" described in [83] with $k = n$, given that in our concept of "multi-trace", all the sub-systems are represented by a trace "component" of the multi-trace. Another remark is that our notion of "multi-trace" is more general than the notion of "k-observation from [83] given that we allow trace components of the multi-trace to represent several sub-systems (components may represent co-localizations non reduced to singletons i.e. a set of several sub-systems).

3.2 A broader and shallower survey

In this section, we propose a broad and shallow survey on the formal semantics of Interaction Languages that can be found in the literature and of their potential uses in Formal Verification (FV).

From studying the state of the art, we have found that most approaches to the definition of formal semantics for ILs can be categorized into three groups depending on the type of formulation of the semantics that is proposed. Those correspond to translational semantics, denotational semantics and operational semantics. We have remarked however that the first two groups are dominant, with translational semantics being quasi-exclusively used for applications in FV. By contrast, fewer direct operational semantics (that do not rely on translations) of ILs are found in the literature.

In any case the reader can refer to the surveys [100] and [85] for a different perspective on the state of the art.

3.2.1 Translational approaches

Many formal approaches to the semantics of ILs rely on translations that map concepts of the given IL into a target formal framework, most often based on automata [55, 34, 124, 83] or Petri nets [52, 47, 54]. Depending on which is the target formal framework, those translations may have the advantage of allowing the use of Formal Verification tools designed for the target framework. As indicated in this survey [28] (among others), there exist a wide array of tools for applying formal methods to the verification of communicating processes. Those tools however mostly rely on input models in the form of communicating automata or Petri nets (UPPAAL [79], DIVERSITY [67], CPNTools [69] etc.).

However, relying on translations to capture semantics leads to reasoning on foreign concepts and it often leads to having some limitations:

- many translations are incomplete (not all concepts of ILs are mapped), for instance in [55], the UML-SDs that are used can only contain synchronous complete (i.e. end-to-end) message exchanges and only a form of strict sequencing (which can be repeated using a loop operator) is handled.
- in some approaches, restrictions are imposed on how the original IL can be used and interpreted. For instance:
  - in [124], there is a need to synchronize at specific points in the interaction; here when entering or exiting some operators. This implies that some interleavings of actions (some actions that are inside the operator w.r.t. others that are not in it) may not be taken into consideration.
  - in [47], the rule for translating loops (Loop Combined Fragment of UML) states that there must be a unique "Deciding Lifeline" that "decides the number of iterations to be used by all other lifelines" and that "Deciding Lifeline" is the one which executes the "first event" of the loop. However nothing in UML-SDs says that such a lifeline should or must exist.

In this section we give a broad overview of translational approaches.

Translations into automata-like formalisms

In [55], UML-SD are translated into timed automata, which are then verified with the UPPAAL tool [79] (this process is applied to a case study concerning a communication protocol of audio/video components). The translation mechanisms that are proposed only concern models with synchronous communications and are only partially automated; many design choices for the UPPAAL model being left to the designer. An "observer automaton" has to be designed so as to intercept communications between automata, make them observable, and enter an error state if other events are observed. The notion of "visual order" of the diagram is similar to our notion of "ordering". However, the process to build the observer automaton from the "visual order" is only detailed in the case where this order is total (i.e. no "par" or "alt"...). This approach still has the advantage of benefiting from UPPAAL's temporal logic model checker, allowing the verification of

predicates such as $\forall\Box P$ (in all reachable states predicate P holds) or $\exists \diamond P$ (a state satisfying predicate P is reachable).

In [34], each lifeline is translated into an associated Timed Input Output Symbolic Transition System (TIOSTS) and message passing relies on some synchronous product. In order to cope with the high level of asynchronism between executions associated to different lifelines, FIFO based communication schema have been introduced to ensure the consistency of executions on different lifelines. Also, given that distinct automata may take different paths when reaching brancing choices specified by *alt* or *loop* operators, so as to maintain consistency, the authors of [34] introduced dedicated variables to keep track of those choices. This approach allows translating UML-SD specifications written with the Papyrus visual modelling tool [61] into automata-like models (TIOSTS), complete with symbolic variables, guards and operations, that can then be formally analyzed via the DIVERSITY tool [67].

In [124], a symbolic automaton is built from UML-SD specifications. [124] is interested in the problem of "trace analysis". Here, test traces can be checked against the automaton and provided a verdict (valid, invalid or inconclusive). The proposed TERMOS framework has limitations e.g. entering and exiting combined fragments are considered to be synchronization points between all lifelines; however it allows lifeline mobility, different nodes being able to appear, disappear and get in or out of range of one another.

Translations into Petri nets

In [52], UML-SD specifications are translated into multivalued nets (M-nets). The translation is compositional, entry and exit places of the M-nets corresponding to subinteractions being connected differently according to the parent combined fragment. However this process is complicated by the tracking of actions that are completely unordered w.r.t. one another. Indeed, so as not to overspecify the translation, "maximal independent sets" are computed and glued together when reconstructing the M-net. As is the case for our approach, goes with the assumption that all behaviour is explicitly specified in the diagrams. There is therefore no use for *neg*, *assert*, *consider* or *ignore* combined fragments. However, like [124], [52] imposes synchronization points (each combined fragment is prefixed and postfixed sequentially). [52] treats data in the form of variables, message parameters and guards but does not deal with time.

In [47], the authors propose an approach to automatically translate UML-SDs designed with the Papyrus tool [61] to Coloured Petri Nets (CPNs) in a format compatible with CPNTools [69] for execution / simulation / testing. CPNs come with an execution semantics that is particularly adapted for the description and analysis of distributed and concurrent systems. A CPN is indeed a network of "places" owning "tokens" and being connected by "arcs". Each such place can behave independently and fire an outgoing transition ("arc") if it fulfills certain requirements (owning specific tokens, etc.). When an "arc" is fired, corresponding tokens are passed between the origin and target "places", signifying the progression of the executed behavior. In [47], the translation revolves around a list of 11 rules with different priorities and which apply to translate different concepts (lifelines, message occurences, combined fragments, etc.) while iterating sequentially through the UML-SD's elements (model-to-model transformation enabled by EMF).

In [54] a set of UML-SDs specifying partial behaviors of a SUT are translated into Extended Petri Nets. Input execution traces can then be checked against the EPNs. The advantage provided by EPNs is that they combine the token of Colored Petri Nets allowing the handling of data values, and the port event places of Event Driven Petri Nets so as to treat exchanges with the environment. This toolset can be applied (for SUTs that are Java programs) to automatically generate and run JUnit testcases.

3.2.2 Denotational approaches

In [122], the author proposes a denotational semantics based on partial order sets (as in [83] which we reviewed in Sec.3.1.5) defined on the set of UML event occurrences. [122] also evokes the issues caused by the *assert* and *negate* operators being taken as operators and not as modalities.

[65] proposes a semantics that is a detailed version of the one from [122]. [65] evacuate the modality issue by only considering the *negate* operator and by having it characterize traces absolutely (as opposed to relatively).

In [42], an institutional approach, likened to that of [75] (which we reviewed in Sec.3.1.4) is proposed. However it includes a *loops* operator (which corresponds to our $loop_W$) and deals with modality by including the *neg* and *assert* operators and separating the semantics in sets of accepted and refused traces.

3.2.3 Operational approaches

In addition to [99], which we have reviewed in Sec.3.1.3, the literature contains few attempts at defining operational semantics for ILs.

The contribution in [84] presents an operational semantics that is somewhat similar to our own in the sense that it builds traces from transformations of the form $i \xrightarrow{act} i'$. However, in addition to the interaction syntax, it relies on a so-called communication medium to define its semantics as the output of a combination of two transitions systems: an execution system which keeps track of the state of the communication medium, and a projection system which selects the next action to execute and provide the interaction resulting from the execution. Communication models, as explained in [53], are tasked with keeping track of which messages have been sent and which are pending receptions. They often take the form of a set of dedicated buffers (e.g. FIFO). Our approach has the advantage of making such communication models implicit as they are implied by the mechanism that construct continuation interactions.

3.2.4 On the analysis of logs

We have seen with [83], which we reviewed in Sec.3.1.5, that it is possible to test MSGs (and more generally ILs) both when the system can be observed globally and when it cannot (local observability). In this section, we discuss some more on the state of the art related to the question of trace analysis w.r.t. interaction models.

Part of our work is indeed related to the general problem of the automatic analysis and debugging of DSs based on local logging of traces [103, 37, 104, 86, 27]. We are positioned at the intersection of two main

issues: (1) that of tracking the causality of actions in traces [103, 86] based on the happened-before relation of Lamport [77] and (2) that of checking multi-traces against formal properties [27] or models [104, 37].

Interactions have been extensively used to validate DSs using Test Case generation [49, 34, 83]. Much effort is spent on the generation of local test cases to mitigate the following problems: (1) "observability", i.e. the difficulty in inferring global executions from partial visions of message exchanges and (2) "controllability", i.e. the difficulty in determining when to apply stimuli in order to realize a targeted global execution. In this present thesis we do not address Test Case generation. However, our work confronts the issue of "observability". With the multi-trace analysis algorithms presented in Chap.9, Chap.10 and Chap.11 we provide methods so as to reconstruct global executions from partial observations (which are partial both in terms of a lack of global reordering and in terms of having missing events) of distibuted executions.

With multi-trace analysis, our work rather falls within the domain of Passive Testing [27, 104] (in which testers are only observers). This notably relates to the Test Oracle Problem [50] which is that of determining expected outputs w.r.t. given stimuli. Such a problem also falls into the domain of offline approaches to Runtime Verification [119, 36]. In [27] and [104], authors have proposed approaches to check a set of local logs recorded in Service Oriented Systems. Authors of [27] propose a methodology to verify the conservation of invariants during the execution of the system. Both local and global invariants can be checked, although the latter is more costly in terms of computations. Our approach is different in that the reference for the analysis is not a correctness property but a model of interaction as in [104, 50].

Logics such as Linear Temporal Logic (LTL), are widely used in runtime verification to specify and verify requirements as logical properties. For DSs, either local properties are considered for synthesizing verifiers (as in the centralized case) in which case verification at a global level is difficult to reason about, or a global property is considered. In the latter case, either the property is transformed into decentralized verifiers and can lose meaning in the process, or all verifiers use the same global property, but they must be informed of other's local states [119]. There remains the possibility of coming back to the centralized case, but the accuracy of the global ordering of events using timestamping requires keeping the remote clocks synchronised [36]. In this perspective, models of interactions are well-suited to be used as a reference for correctness when analyzing DS executions. This is all the more relevant in cases where the temporal ordering of remote events is not feasible.

[104] discusses passive testing against models of interactions expressed in the Chor [113] language. It differs from our approach in so far as: (1) Chor is less expressive than the IL we propose (particularly w.r.t. the absence of weak sequencing and the nature of loops), (2) [104] only handles synchronous communication between services, which cannot always describe accurately concrete implementations and (3) the local logs are not directly checked against the model but first pass through a synthesis step in which a global log is reconstituted based on timestamp information, and then this global log is checked. In [104], putting logs together is facilitated by assuming synchronized clocks, which is not a prerequisite to applying our analysis approach.

Authors in [50] investigate the computational cost of log analysis w.r.t. graphs of MSCs. This cost is

compared in different cases according to the quality of observations (local or tester observability i.e. whether one has a set of independent local logs or a globally ordered log) and the expressivity of the MSC graphs (presence of choice, loop or parallelism). The work echoes results for "MSC Membership" [26, 60] which state that this problem is NP-complete. The main factor of the cost blow-up lies in the fact that distributed actions can be equally re-ordered in multiple ways. In this thesis, we address this issue of the complexity of multi-trace analysis in Chap.9.

3.3 Conclusion and position of the thesis

ILs include a variety of specific formalisms, some being particularly expressive in terms of the intended meanings (informal semantics) of the models they can define. However, most formal approaches are restricted to specific sub-languages and, in the case of applications to FV, mostly rely on translations towards other formalisms.

With our thesis, we propose an expressive Interaction Language (IL) and a framework for FV based on a structural operational semantics which directly manipulates interaction terms (guaranteeing traceability and allowing the graphical animation of the execution of interaction models) and which is backed-up by (i.e. proven equivalent to) a denotational semantics which serves as a mathematical foundation. We use this framework to implement multi-trace analysis algorithms that provide solutions to the membership problems associated to various multi-trace semantics (that of accepted multi-traces and various notions of prefixes, reflecting partially observed behavior).

Part I

The Interaction Language

Chapter 4

Syntax & Denotation

Contents

In the previous chapters, we have introduced the context in which this thesis falls (Chap.2) and a state of the art on the semantics of Interaction Languages (Chap.3) that is formal languages for specifying the behaviors of distributed systems in sequence-diagram-like models.

In this chapter, we present our own take on a formal Interaction Language (IL). Our interactions are terms of a term algebra. Those terms are then associated to a semantics, which represents the behaviors that are specified by the model. The semantics of an interaction takes the form of a set of traces. Each trace - a sequence of atomic events (emission or reception of a message) - represents a behavior that the modelled system can express. The trace semantics that we propose in this section is a denotational-style semantics that uses some operators on sets of traces (the semantic domain) to construct the semantics of interactions by composition.

As a result, the layout of this chapter is the following:

- in Sec.4.1 we formally define the semantic domain and some algebraic operators on this domain,
- in Sec.4.2 we introduce the IL and its associated denotational-style semantics,
- in Sec.4.3 we discuss equivalent interaction terms and define a process to compute normal forms of interaction terms.

The formalization and all the results presented in Sec.4.1 and Sec.4.2 of this chapter have been encoded and proven using the Coq proof assistant in [88].

4.1 Semantic domain

In this section, we will define the semantic domain of our IL. This domain is that of sets of traces. Traces are sequences of observable events that characterize specific executions of Distributed System (DS)s. Given that a system may express several distinct behaviors, their models (interactions) can be associated to semantics in the form of sets of traces.

4.1.1 Traces

As in [75], that we reviewed in Chap.3, interactions and the behaviors they can express are defined up to a given signature $\Omega = (L, M)$ where:

- L is a set of lifelines. Lifelines are abstractions which can be used to represent sub-systems, or groups of sub-systems of a DS.
- M is a set of messages. Messages are abstractions which can be used to represent what is exchanged between several sub-systems of the DS, or between a sub-system and the external environment of the DS.

The atomic events that can be defined to represent the exchange of messages, which characterize the execution of the DS are communications actions. Indeed, the principle of interactions is to represent DSs through the perspective of their internal and external communications. A communication action is:

- either the emission of a message $m \in M$ from a lifeline $l \in L$, noted $l!m$
- or the reception of a message $m \in M$ by a lifeline $l \in L$, noted $l?m$

We note $\mathbb{A}_\Omega$ the set of such actions, defined up to the signature Ω. For any such action a, the notation $\theta(a)$ denotes the lifeline on which a occurs.

Definition 4.1: Communication Action

For any signature $\Omega = (L, M)$, we define the set of communication actions:

$$\mathbb{A}_\Omega = \{l\Delta m \mid (l \in L) \ \wedge \ (\Delta \in \{!, ?\}) \ \wedge \ (m \in M)\}$$

For any action $a \in \mathbb{A}_\Omega$ of the form $l\Delta m$, $\theta(a)$ denotes the lifeline l.

As in [75], an execution of a DS defined over a signature $\Omega = (L, M)$ can be characterized by a sequence of actions from $\mathbb{A}_\Omega$ that occurred globally. Such a sequence is called a trace and describes an execution from a global perspective. There is a total order between actions disregarding the lifelines on which they occur. We define traces over Ω as words in $\mathbb{A}_\Omega^*$, with "." as concatenation law and ϵ the empty trace. The set of traces is denoted by $\mathbb{T}_\Omega$.

Definition 4.2: Traces of Execution

For any signature $\Omega = (L, M)$, we define the set of traces of execution:

$$\mathbb{T}_\Omega = \mathbb{A}_\Omega^*$$

For any $t \in \mathbb{T}_\Omega$, we denote by $|t|$ the length of t i.e. the number of actions in the sequence, which is defined by:

- $|\epsilon| = 0$
- $|a.t| = 1 + |t|$ for any action $a \in \mathbb{A}_\Omega$ and trace $t \in \mathbb{T}_\Omega$

For example, $t = l_1!m_1.l_2?m_1.l_2!m_2$ is a trace on the signature $\Omega = (L, M)$ with $L = \{l_1, l_2\}$ and $M = \{m_1, m_2\}$ and we have $|t| = 3$.

In the following, we will introduce some operators which can be applied to elements of $\mathbb{T}_\Omega$, and state some of their properties, as preliminaries to later definitions and proofs. Those operators are then extended to sets of traces as in [75].

4.1.2 Interleaving operator

The interleaving operator, denoted by $||$ corresponds to the shuffling of elements from two traces t_1 and t_2 such that the local orders of elements within t_1 and t_2 are respected, but there is no such order that is enforced when it comes to ordering elements from t_1 w.r.t. those of t_2.

Interleaving operator on traces

Let us at first define the $||$ operator on traces in Def.4.3.

Definition 4.3: Interleaving on traces

For any signature Ω, we define $|| : \mathbb{T}_\Omega \times \mathbb{T}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ the function s.t. for any t_1 and t_2 in $\mathbb{T}_\Omega$ and any a_1 and a_2 in $\mathbb{A}_\Omega$:

$$\begin{array}{lcl} \epsilon||t_2 & = & \{t_2\} \\ t_1||\epsilon & = & \{t_1\} \\ (a_1.t_1)||(a_2.t_2) & = & \{a_1.t \mid t \in t_1||(a_2.t_2)\} \cup \{a_2.t \mid t \in (a_1.t_1)||t_2\} \end{array}$$

For example, we have:

$$a_1.a_2 \;||\; a_4 = \left\{ \begin{array}{l} a_4.a_1.a_2, \\ a_1.a_4.a_2, \\ a_1.a_2.a_4 \end{array} \right\}$$

In the following, we will state and prove some properties w.r.t. the interleaving operator.

Lem.4.1 states that for any two traces t_1 and t_2, the set of their interleavings $(t_1||t_2)$ contains at least one trace.

Lemma 4.1: Existence of at least one interleaving

For any t_1 and t_2 in $\mathbb{T}_\Omega$, we have:

$$(t_1||t_2) \neq \emptyset$$

Proof. Let us reason by induction on trace t_1.

- If $t_1 = \epsilon$ then $\epsilon||t_2 = \{t_2\} \neq \emptyset$
- Let us suppose that $t_1||t_2 \neq \emptyset$. Then, given that $\{a.t \mid t \in (t_1||t_2)\} \subset (a.t_1||t_2)$, it immediately follows that $(a.t_1||t_2) \neq \emptyset$

□

Lem.4.2 is another property related to the existence of a certain decomposition given that a non-empty trace of the form $a.t$ is an interleaving of two traces t_1 and t_2.

Lemma 4.2: Implications of having a non-empty interleaving

For any t_1, t_2 and t in $\mathbb{T}_\Omega$ and for any $a \in \mathbb{A}_\Omega$ we have:

$$(a.t \in (t_1||t_2)) \Rightarrow \left(\exists\ t_3 \in \mathbb{T}_\Omega,\ \left(\begin{array}{l} \left((t_1 = a.t_3) \wedge (t \in (t_3||t_2)) \right) \\ \vee \left((t_2 = a.t_3) \wedge (t \in (t_1||t_3)) \right) \end{array} \right) \right)$$

Proof. This property is an immediate consequence of the definition of the $||$ operator. □

In Lem.4.3 we consider some properties of the interleaving operator w.r.t. the empty trace ϵ.

Lemma 4.3: Some properties of the interleaving operator w.r.t. ϵ

For any t_1, t_2 and t in $\mathbb{T}_\Omega$, we have:

(i) $(\exists\ t \in (t_1||t_2),\ t \neq \epsilon) \Leftrightarrow ((t_1 \neq \epsilon) \vee (t_2 \neq \epsilon))$

(ii) $(\epsilon \in (t_1||t_2)) \Leftrightarrow ((t_1 = \epsilon) \wedge (t_2 = \epsilon))$

(iii) $(t \in (\epsilon||t_2)) \Rightarrow (t = t_2)$

(iv) $(t \in (t_1||\epsilon)) \Rightarrow (t = t_1)$

Proof. ◯ Let us prove the first property (i):

⇒ We suppose the existence of t. The fact that $t \neq \epsilon$ implies the existence of action a and trace t' such that $t = a.t'$. Then, as per Lem.4.2, $a.t' \in (t_1||t_2)$ implies that either $t_1 = a.t'_1$ and $t' \in (t'_1||t_2)$ or $t_2 = a.t'_2$ and $t' \in (t_1||t'_2)$. In any case this implies that at least one of either t_1 or t_2 is non-empty.

⇐ Let us suppose that $t_1 \neq \epsilon$ (the other case is similar). Then there exist action a and trace t'_1 such that $t_1 = a.t'_1$. Let us also remark that $(t'_1||t_2) \neq \emptyset$ as per Lem.4.1. Let us then consider $t_0 \in (t'_1||t_2)$. By definition we therefore have $a.t_0 \in (a.t'_1||t_2) = (t_1|t_2)$ and $a.t_0 \neq \epsilon$.

◯ The other points are immediate given the definition of the $||$ operator. □

Finally, in Lem.4.4, we consider some algebraic properties of the $||$ operator on traces. We can remark that this operator is symmetric and associative.

Lemma 4.4: Algebraic properties of the interleaving operator on traces

For any t_1, t_2, t_3 and t in $\mathbb{T}_\Omega$, we have:

symmetry	$(t \in (t_1||t_2)) \Leftrightarrow (t \in (t_2||t_1))$
associativity	$t \in ((t_1||t_2)||t_3) \Leftrightarrow t \in (t_1||(t_2||t_3))$

Proof. ◯ Let us start by the symmetry. Let us suppose the existence of $t \in (t_1||t_2)$ and prove that $t \in (t_2||t_1)$ (the other direction is the same). Let us reason by induction on (the size of) t.

- If $t = \epsilon$, then, as per Lem.4.3, we have $t_1 = t_2 = \epsilon$ so $\epsilon \in (\epsilon||\epsilon)$ all the same.
- If $t = a.t'$, then, as per Lem.4.2, we have:
  - either $t_1 = a.t'_1$ and $t' \in (t'_1||t_2)$. In that case, t' being smaller that t, we can apply the induction hypothesis, which implies that $t' \in (t_2||t'_1)$ and then by definition of the $||$ operator, $t = a.t' \in (t_2||a.t'_1) = (t_2||t_1)$.
  - or $t_2 = a.t'_2$ and $t' \in (t_1||t'_2)$. In that case, t' being smaller that t, we can apply the induction hypothesis, which implies that $t' \in (t'_2||t_1)$ and then by definition of the $||$ operator, $t = a.t' \in (a.t'_2||t_1) = (t_2||t_1)$.

◯ Let us now consider the associativity.

⇒ Let us consider $t \in ((t_1||t_2)||t_3)$. This means that there exists a certain t_{12} s.t. $t_{12} \in (t_1||t_2)$ and $t \in (t_{12}||t_3)$. Let us then reason by induction on t.

- If $t = \epsilon$ then, as per Lem.4.3, this implies that $t_{12} = t_3 = \epsilon$, and the fact that $t_{12} \in (t_1||t_2)$ implies that $t_1 = t_2 = \epsilon$. Then $\epsilon \in (\epsilon||(\epsilon||\epsilon))$ is given.
- If $t = a.t'$, then, as per Lem.4.2, we have:
  - ∗ either $t_{12} = a.t'_{12}$ and $t' \in (t'_{12}||t_3)$. Then, given that $a.t'_{12} \in (t_1||t_2)$ by applying Lem.4.2 again we have:
    - · either $t_1 = a.t'_1$ and $t'_{12} \in (t'_1||t_2)$. In that case we have $t' \in ((t'_1||t_2)||t_3)$. Then, we can apply the induction hypothesis on t', which implies that $t' \in (t'_1||(t_2||t_3))$. By definition of the $||$ operator, $t = a.t' \in (a.t'_1||(t_2||t_3)) = (t_1||(t_2||t_3))$
    - · or $t_2 = a.t'_2$ and $t'_{12} \in (t_1||t'_2)$. In that case we have $t' \in ((t_1||t'_2)||t_3)$. Let us then consider $t'_{23} \in (t'_2||t_3)$ s.t. $t' \in (t_1||t'_{23})$. By definition of the $||$ operator, $t = a.t' \in (t_1||a.t'_{23})$ and $a.t'_{23} \in (a.t'_2||t_3) = (t_2||t_3)$. Therefore $t \in (t_1||(t_2||t_3))$
  - ∗ or $t_3 = a.t'_3$ and $t' \in (t_{12}||t'_3) \subset ((t_1||t_2)||t'_3)$. Then, we can apply the induction hypothesis on t', which implies that $t' \in (t_1||(t_2||t'_3))$. Let us then consider $t'_{23} \in (t_2||t'_3)$ s.t. $t' \in (t_1||t'_{23})$. By definition of the $||$ operator, $t = a.t' \in (t_1||a.t'_{23})$ and $a.t'_{23} \in (t_2||a.t'_3) = (t_2||t_3)$. Therefore $t \in (t_1||(t_2||t_3))$

⇐ The other direction can be proven in a similar fashion.

□

Interleaving operator on sets of traces

We then extend the definition of the interleaving operator to sets of trace in Def.4.4. We will use the same overloaded notation $||$ which is not problematic given that for any two traces t_1 and t_2 we have $t_1||t_2 = \{t_1\}||\{t_2\}$.

Definition 4.4: Interleaving on sets of traces

We extend $||$ to sets of traces with $|| : \mathcal{P}(\mathbb{T}_\Omega) \times \mathcal{P}(\mathbb{T}_\Omega) \to \mathcal{P}(\mathbb{T}_\Omega)$ s.t. for any sets of traces T_1 and T_2:

$$T_1||T_2 = \bigcup_{\substack{t_1 \in T_1 \\ t_2 \in T_2}} (t_1||t_2)$$

For example, we have:

$$\left\{ \begin{array}{l} a_1.a_2, \\ a_3 \end{array} \right\} || \left\{ a_4 \right\} = \left\{ \begin{array}{l} a_4.a_1.a_2, \\ a_1.a_4.a_2, \\ a_1.a_2.a_4, \\ a_4.a_3, \\ a_3.a_4 \end{array} \right\}$$

In Lem.4.5 we state some algebraic properties of this extended operator.

Lemma 4.5: Algebraic properties of the interleaving operator on sets of traces

For any sets T, T_1, T_2 and T_3 in $\mathcal{P}(\mathbb{T}_\Omega)$:

neutral element	$T\|\|\{\epsilon\} = T = \{\epsilon\}\|\|T$
commutativity	$T_1\|\|T_2 = T_2\|\|T_1$
associativity	$(T_1\|\|T_2)\|\|T_3 = T_1\|\|(T_2\|\|T_3)$
left-distributivity	$T_1\|\|(T_2 \cup T_3) = (T_1\|\|T_2) \cup (T_1\|\|T_3)$
right-distributivity	$(T_1 \cup T_2)\|\|T_3 = (T_1\|\|T_3) \cup (T_2\|\|T_3)$

Proof. The fact that $\{\epsilon\}$ is a neutral element is implied by definition and by Lem.4.3. The commutativity and associativity are implied by the respective properties from Lem.4.4. The distributivity is immediate. □

4.1.3 Strict sequencing operator

Strict sequencing operator on traces

The strict sequencing operator, denoted by ";" corresponds to the concatenation of traces.

Definition 4.5: Strict Sequencing on traces

For any signature Ω, we define $;: \mathbb{T}_\Omega \times \mathbb{T}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ the function s.t. for any t_1 and t_2 in $\mathbb{T}_\Omega$ and any a in $\mathbb{A}_\Omega$:

$$\epsilon; t_2 = \{t_2\} \qquad \text{and} \qquad (a.t_1); t_2 = \{a.t \mid t \in t_1; t_2\}$$

In fact, the strict sequencing of two traces t_1 and t_2, denoted by $t_1; t_2$ is the singleton containing the concatenation $t_1.t_2$ of both traces. This formulation in terms of a set of trace can be explained by a will to have the same formulation as that of the other operators. This notation has notably been used in [75].

For example, we have:

$$a_1.a_2 \ ; \ a_4.a_5 = \left\{ \ a_1.a_2.a_4.a_5 \ \right\}$$

As what we did for the interleaving operator, let us consider analogous properties for the strict sequencing operator concerning the existence of strict sequences. In Lem.4.6 we state the existence and the unicity of the strict sequencing of two traces t_1 and t_2.

Lemma 4.6: Existence of exactly one strict sequence

For any t_1 and t_2 in $\mathbb{T}_\Omega$, we have:

$$(t_1; t_2) = \{t_1.t_2\}$$

Proof. Immediate. □

Lem.4.7 is analogous to Lem.4.2. It states the the existence of a certain decomposition given that a non empty trace of the form $a.t$ is a strict sequence of two traces t_1 and t_2.

Lemma 4.7: Implications of having a non-empty strict sequence

For any t_1, t_2 and t in $\mathbb{T}_\Omega$ and for any $a \in \mathbb{A}_\Omega$ we have:

$$(a.t \in (t_1; t_2)) \Rightarrow \left(\exists\, t_3 \in \mathbb{T}_\Omega, \left(\begin{array}{l} \left(\ (t_1 = a.t_3) \wedge (t \in (t_3; t_2)) \ \right) \\ \vee \ \left(\ (t_1 = \epsilon) \wedge (t_2 = a.t) \ \right) \end{array} \right) \right)$$

Proof. Immediate. □

In Lem.4.8, we consider some properties of the strict sequencing operator w.r.t. the empty trace ϵ.

Lemma 4.8: Some properties of the strict sequencing operator w.r.t. ϵ

For any t_1, t_2, t_3 and t in $\mathbb{T}_\Omega$, for any $l \in L$, and any $a \in \mathbb{A}_\Omega$ we have:

(i) $(\exists\, t \in (t_1; t_2),\ t \neq \epsilon) \Leftrightarrow ((t_1 \neq \epsilon) \vee (t_2 \neq \epsilon))$

(ii) $(\epsilon \in (t_1; t_2)) \Leftrightarrow ((t_1 = \epsilon) \wedge (t_2 = \epsilon))$

(iii) $(t \in (\epsilon; t_2)) \Rightarrow (t = t_2)$

(iv) $(t \in (t_1; \epsilon)) \Rightarrow (t = t_1)$

Proof. Immediate. □

Finally, in Lem.4.9, we consider some algebraic properties of the ; operator. We can remark that this operator is associative.

Lemma 4.9: Algebraic properties of the strict sequencing operator on traces

For any t_1, t_2 and t_3 in $\mathbb{T}_\Omega$ we have:

$$\text{associativity} \qquad t \in ((t_1; t_2); t_3) \Leftrightarrow t \in (t_1; (t_2; t_3))$$

Proof. Immediate. □

Strict sequencing operator on sets of traces

Similarly to what we did for the interleaving operator, we extend, in Def.4.6 the strict sequencing operator to sets of traces. We will likewise use the same overloaded notation ; which is not problematic given that for any two traces t_1 and t_2 we have $t_1; t_2 = \{t_1\}; \{t_2\}$.

Definition 4.6: Strict Sequencing on sets of traces

We extend ; to sets of traces with $;: \mathcal{P}(\mathbb{T}_\Omega) \times \mathcal{P}(\mathbb{T}_\Omega) \to \mathcal{P}(\mathbb{T}_\Omega)$ s.t. for any sets of traces T_1 and T_2:

$$T_1; T_2 = \bigcup_{\substack{t_1 \in T_1 \\ t_2 \in T_2}} (t_1; t_2)$$

For example, we have:

$$\left\{ \begin{array}{l} a_1.a_2, \\ a_3 \end{array} \right\}; \left\{ a_4.a_5 \right\} = \left\{ \begin{array}{l} a_1.a_2.a_4.a_5, \\ a_3.a_4.a_5 \end{array} \right\}$$

In Lem.4.10 we state some algebraic properties of this extended operator.

Lemma 4.10: Algebraic properties of the strict sequencing operator on sets of traces

For any sets T, T_1, T_2 and T_3 in $\mathcal{P}(\mathbb{T}_\Omega)$:

$$\begin{aligned}
&\text{neutral element} && T;\{\epsilon\} = T = \{\epsilon\};T \\
&\text{associativity} && (T_1;T_2);T_3 = T_1;(T_2;T_3) \\
&\text{left-distributivity} && T_1;(T_2 \cup T_3) = (T_1;T_2) \cup (T_1;T_3) \\
&\text{right-distributivity} && (T_1 \cup T_2);T_3 = (T_1;T_3) \cup (T_2;T_3)
\end{aligned}$$

Proof. The fact that $\{\epsilon\}$ is a neutral element is implied by definition and by Lem.4.8. The associativity is implied by the respective property from Lem.4.9. The distributivity is immediate. □

4.1.4 Weak sequencing operator

The weak sequencing operator, denoted by $;_\divideontimes$ corresponds to the reordering of events from traces such that events from the right-hand side trace can only be placed when all the events occurring on the same lifeline from the left-hand side trace have already been placed.

Conflict

The definition of this operator relies on a notion of conflict, denoted by the $\divideontimes$ function, which is formally defined in Def.4.7. For any lifeline l and any trace t, $t \divideontimes l$ denotes the occurrence of a conflict between l and t, i.e. that there is at least one action in t that occurs on lifeline l.

Definition 4.7: Conflict

For any signature Ω, we define $\divideontimes : \mathbb{T}_\Omega \times L \to \{\top, \bot\}$ s.t. for any lifeline l and any trace t:

$$\begin{aligned}
\epsilon \divideontimes l &= \bot \\
(a.t) \divideontimes l &= (\theta(a) = l) \vee (t \divideontimes l)
\end{aligned}$$

We can extend $\divideontimes$ to sets of traces as $\divideontimes : \mathcal{P}(\mathbb{T}_\Omega) \times L \to \{\top, \bot\}$ s.t. for any $T \in \mathcal{P}(\mathbb{T}_\Omega)$ and $l \in L$:

$$T \divideontimes l \Leftrightarrow (\exists\ t \in T \text{ s.t. } t \divideontimes l)$$

Weak sequencing operator on traces

Weak sequencing then functions in a similar fashion as interleaving but with the added constraint that events occurring on the right-hand side trace are only added if they have no conflict w.r.t. what remains to be added in the left-hand side trace. Its formal definition is given in Def.4.8.

Definition 4.8: Weak Sequencing on traces

For any signature Ω, we define $;_{\ltimes} : \mathbb{T}_\Omega \times \mathbb{T}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ the function s.t. for any t_1 and t_2 in $\mathbb{T}_\Omega$ and any a_1 and a_2 in $\mathbb{A}_\Omega$:

$$\begin{array}{lcl} \epsilon ;_{\ltimes} t_2 & = & \{t_2\} \\ t_1 ;_{\ltimes} \epsilon & = & \{t_1\} \\ (a_1.t_1) ;_{\ltimes} (a_2.t_2) & = & \{a_1.t \mid t \in t_1 ;_{\ltimes} (a_2.t_2)\} \cup \left\{ a_2.t \;\middle|\; \begin{array}{c} (t \in (a_1.t_1) ;_{\ltimes} t_2) \\ \wedge (\neg (a_1.t_1 \ltimes \theta(a_2))) \end{array} \right\} \end{array}$$

Let us remark that, although it may be due to a typographical error or an oversight in [75], we provide a different definition than that of [75]. Indeed, in [75] it suffices that the first element a_1 of the left-hand side $a_1.t_1$ do not enter into conflict with the first element a_2 of the right-hand side $a_2.t_2$ for a_2 to be added at the beginning of the reconstructed trace (it is the condition $\neg(a_1 \ltimes a_2)$ in [75], with the definition of $\ltimes$ being that of a conflict between two actions).

For example, we have:

$$l_1!m_1.l_2!m_2 \;;_{\ltimes}\; l_1!m_3 = \left\{ \begin{array}{l} l_1!m_1.l_1!m_3.l_2!m_2, \\ l_1!m_1.l_2!m_2.l_1!m_3 \end{array} \right\}$$

In the following, we will state some properties of the weak sequencing operator as we have done for the two previous operators.

In Lem.4.11, we state that for any two traces t_1 and t_2, the set of their weak sequences $(t_1 ;_{\ltimes} t_2)$ contains at least one trace.

Lemma 4.11: Existence of at least one weak sequence

For any t_1 and t_2 in $\mathbb{T}_\Omega$, we have:

$$(t_1 ;_{\ltimes} t_2) \neq \emptyset$$

Proof. Analogous too that of Lem.4.1. □

Lem.4.12 is another property related to the existence of a certain decomposition given that a non empty trace of the form $a.t$ is a weak sequence of two traces t_1 and t_2.

Lemma 4.12: Implications of having a non-empty weak sequence

For any t_1, t_2 and t in $\mathbb{T}_\Omega$ and for any $a \in \mathbb{A}_\Omega$ we have:

$$(a.t \in (t_1 ;_\ltimes t_2)) \Rightarrow \left(\exists\, t_3 \in \mathbb{T}_\Omega,\ \left(\begin{array}{l} \left(\ (t_1 = a.t_3) \wedge (t \in (t_3 ;_\ltimes t_2)) \ \right) \\ \vee \left(\ (\neg(t_1 \ltimes \theta(a))) \wedge (t_2 = a.t_3) \wedge (t \in (t_1 ;_\ltimes t_3)) \ \right) \end{array} \right) \right)$$

Proof. Analogous too that of Lem.4.2, except for the added condition $\neg(t_1 \ltimes \theta(a))$ in the case where the action a is found in the right trace t_2, and which stems from the fact that it wouldn't be otherwise possible, if there were some conflicts between $\theta(a)$ and t_1. □

In Lem.4.13, we consider some properties of the weak sequencing operator w.r.t. the empty trace ϵ.

Lemma 4.13: Some properties of the weak sequencing operator w.r.t. ϵ

For any t_1, t_2 and t in $\mathbb{T}_\Omega$, we have:

(i) $(\exists\, t \in (t_1 ;_\ltimes t_2),\ t \neq \epsilon) \Leftrightarrow ((t_1 \neq \epsilon) \vee (t_2 \neq \epsilon))$

(ii) $(\epsilon \in (t_1 ;_\ltimes t_2)) \Leftrightarrow ((t_1 = \epsilon) \wedge (t_2 = \epsilon))$

(iii) $(t \in (\epsilon ;_\ltimes t_2)) \Rightarrow (t = t_2)$

(iv) $(t \in (t_1 ;_\ltimes \epsilon)) \Rightarrow (t = t_1)$

Proof. Analogous too that of Lem.4.3. □

Before introducing the algebraic properties of the weak sequencing operator (its associativity), let us quickly digress on the relationships of the $\ltimes$ "conflict" function w.r.t. the three previously defined operators (strict and weak sequencing and interleaving). We remark some distributive properties of $\ltimes$ w.r.t. the three operators. This property of distributivity will be required to prove the associativity of $;_\ltimes$.

Lemma 4.14: Distributive properties of $\ltimes$

Let us consider $\diamond \in \{;,\ ;_\ltimes,\ ||\}$. Then, for any traces t_1 and t_2 and any lifeline l we have:

$$(\exists\, t \in (t_1 \diamond t_2),\ \neg(t \ltimes l)) \quad \Leftrightarrow \quad ((\neg(t_1 \ltimes l)) \wedge (\neg(t_2 \ltimes l))) \quad \Leftrightarrow \quad (\forall\, t \in (t_1 \diamond t_2),\ \neg(t \ltimes l))$$

And for any non-empty sets of traces T_1 and T_2 we have:

$$\begin{aligned} (T_1 \diamond T_2) \ltimes l \quad &\Leftrightarrow \quad (T_1 \ltimes l) \vee (T_2 \ltimes l) \\ (T_1 \cup T_2) \ltimes l \quad &\Leftrightarrow \quad (T_1 \ltimes l) \vee (T_2 \ltimes l) \end{aligned}$$

Proof. Let us discuss both properties:

◯ For the first property, the sketch of the proof is that the absence of conflict for any recomposed trace t w.r.t. a lifeline l means that all of its actions do not occur on l. Given that the set of the actions that occur within this recomposed trace is exactly the union of those that occur in either t_1 and/or t_2, then this absence of conflict is equivalent to having an absence of conflict for both t_1 and t_2 w.r.t. l.

◯ For the second property, the sketch of the proof is that the presence of a conflict w.r.t lifeline l for a certain recomposed trace $t \in (T_1 \diamond T_2)$ means that it contains an action occurring on l. Then this action must either come from a trace from T_1 or from a trace from T_2. Reciprocally any action occurring on l found in a trace from either T_1 or T_2 will also be found on a least one trace from T (given non-emptiness). □

Let us finally consider the algebraic properties of the $;_{\divideontimes}$ operator. We can remark that this operator is associative, as stated by Lem.4.15.

Lemma 4.15: Algebraic properties of the weak sequencing operator

For any t_1, t_2, t_3 and t in $\mathbb{T}_\Omega$, we have:

$$\text{associativity} \qquad t \in ((t_1 ;_{\divideontimes} t_2) ;_{\divideontimes} t_3) \Leftrightarrow t \in (t_1 ;_{\divideontimes} (t_2 ;_{\divideontimes} t_3))$$

Proof. The proof is analogous to that of the associativity of $||$ from Lem.4.4. The main difference occurs in the induction, when considering a trace of the form $t = a.t'$. If the action a is to be found in t_2 then there must be no conflict between t_1 and $\theta(a)$. If a is found in t_3 then there must be no conflict between t_1 and $\theta(a)$ and no conflict either between t_2 and $\theta(a)$. When recomposing the traces t_{23} this must also be taken into account. To do that we need the distributivity of $\divideontimes$ over $;_{\divideontimes}$, which was introduced in Lem.4.14 (hence the digression). □

Weak sequencing operator on sets of traces

Similarly to what we did for the two previous operators, we extend, in Def.4.9 the weak sequencing operator to sets of traces. We will likewise use the same overloaded notation $;_{\divideontimes}$ which is not problematic given that for any two traces t_1 and t_2 we have $t_1 ;_{\divideontimes} t_2 = \{t_1\} ;_{\divideontimes} \{t_2\}$.

Definition 4.9: Weak Sequencing on sets of traces

We extend $;_{\divideontimes}$ to sets of traces with $;_{\divideontimes} : \mathcal{P}(\mathbb{T}_\Omega) \times \mathcal{P}(\mathbb{T}_\Omega) \to \mathcal{P}(\mathbb{T}_\Omega)$ s.t. for any sets of traces T_1 and T_2:

$$T_1 ;_{\divideontimes} T_2 = \bigcup_{\substack{t_1 \in T_1 \\ t_2 \in T_2}} (t_1 ;_{\divideontimes} t_2)$$

For example, we have:

$$\left\{ \begin{array}{l} l_1!m_1.l_2!m_2, \\ l_2?m_1 \end{array} \right\} ;_* \left\{ \; l_1!m_3 \; \right\} = \left\{ \begin{array}{l} l_1!m_1.l_1!m_3.l_2!m_2, \\ l_1!m_1.l_2!m_2.l_1!m_3, \\ l_1!m_3.l_2?m_1, \\ l_2?m_1.l_1!m_3 \end{array} \right\}$$

In Lem.4.16 we state some algebraic properties of this extended operator.

Lemma 4.16: Algebraic properties of the weak sequencing operator on sets of traces

For any sets T, T_1, T_2 and T_3 in $\mathcal{P}(\mathbb{T}_\Omega)$:

neutral element	$T ;_* \{\epsilon\} = T = \{\epsilon\} ;_* T$
associativity	$(T_1 ;_* T_2) ;_* T_3 = T_1 ;_* (T_2 ;_* T_3)$
left-distributivity	$T_1 ;_* (T_2 \cup T_3) = (T_1 ;_* T_2) \cup (T_1 ;_* T_3)$
right-distributivity	$(T_1 \cup T_2) ;_* T_3 = (T_1 ;_* T_3) \cup (T_2 ;_* T_3)$

Proof. The fact that $\{\epsilon\}$ is a neutral element is implied by definition and by Lem.4.13. The associativity is implied by the respective property from Lem.4.15. The distributivity is immediate. □

4.1.5 Comparing the scheduling operators

We can remark that compositions enabled by the strict sequencing operator are also enabled by the weak sequencing operator, and, in turn, compositions enabled by the weak sequencing operator are also enabled by the interleaving operator. As a result, we can order the operators as follows: (1) the interleaving operator is the weakest of the three, given that it is less strict w.r.t. which compositions it allows, and (2) the strict sequencing operator is the strongest.

Lemma 4.17: Comparing scheduling operators on traces

For any traces t_1 and t_2 we have:

$$(t_1 ; t_2) \subset (t_1 ;_* t_2) \quad \text{and} \quad (t_1 ;_* t_2) \subset (t_1 || t_2)$$

Proof. ◯ For the first predicate, let us proceed by induction on t_1.

- If $t_1 = \epsilon$ then $(\epsilon ; t_2) = \{t_2\}$ and $(\epsilon ;_* t_2) = \{t_2\}$ so the property holds.
- If $t_1 = a.t_1'$ then $(a.t_1' ; t_2) = \{a.t \mid t \in t_1' ; t_2\}$ and given that, by the induction hypothesis $(t_1' ; t_2) \subset (t_1' ;_* t_2)$, this is included in $(a.t_1' ;_* t_2)$.

◯ We can proceed in a similar fashion for the second predicate i.e. the inclusion of $(t_1 ;_* t_2)$ into $(t_1 || t_2)$ □

This property can be immediately extended to sets of traces as is done in Lem.4.18.

Lemma 4.18: Comparing scheduling operators

For any sets of traces T_1 and T_2 we have:

$$(T_1;T_2) \subset (T_1;_* T_2) \quad \text{and} \quad (T_1;_* T_2) \subset (T_1||T_2)$$

Proof. Immediately implied by Lem.4.17. □

4.1.6 Kleene closures

Definition & basic properties

We have seen in the lemmas Lem.4.5, Lem.4.10 and Lem.4.16, that all three scheduling operators are associative. This allows us to define the Kleene closures of those operators in Def.4.10.

Definition 4.10: Kleene closures

For any scheduling operator $\diamond \in \{;,\ ;_*,\ ||\}$, we define, for any set of traces $T \in \mathcal{P}(\mathbb{T}_\Omega)$:

$$\begin{aligned} T^{\diamond 0} &= \{\epsilon\} \\ T^{\diamond j} &= T \diamond T^{\diamond(j-1)} \quad \textbf{for } j > 0 \\ T^{\diamond *} &= \bigcup_{j\in\mathbb{N}} T^{\diamond j} \end{aligned}$$

The three Kleene closures defined in Def.4.10 may be called as follows:

- $;^*$ is the strict Kleene closure,
- $;_*{}^*$ is the weak Kleene closure,
- $||^*$ is the interleaving Kleene closure.

Within the Kleene closure $T^{\diamond *}$ we can find traces obtained from the repetition (using the specific scheduling operator $\diamond$) of any number of traces of T.

Let us consider the following example. In this example we have a set T of traces in which there are two traces $l_1!m_1.l_2!m_2$ in orange and $l_2?m_1$ in red. We consider here the weak Kleene closure of T i.e. $T^{;_*{}^*}$. We display a subset of $T^{;_*{}^*}$ (the rest being indicated by the $\cdots$) which consists of $T^{;_*{}^0} \cup T^{;_*{}^1} \cup T^{;_*{}^2}$ i.e. the first 3 powersets of T.

$$\left\{ \begin{array}{l} l_1!m_1.l_2!m_2, \\ l_2?m_1 \end{array} \right\}^{;\divideontimes *} = \left\{ \begin{array}{ll} \epsilon, & l_1!m_1.l_2!m_2.l_1!m_1.l_2!m_2, \\ l_1!m_1.l_2!m_2, & l_1!m_1.l_1!m_1.l_2!m_2.l_2!m_2, \\ l_2?m_1, & l_1!m_1.l_2!m_2.l_2?m_1, \\ & l_2?m_1.l_2?m_1, \\ & l_2?m_1.l_1!m_1.l_2!m_2, \\ & l_1!m_1.l_2?m_1.l_2!m_2, \end{array} \quad \ldots \right\}$$

In the following we state some properties of the Kleene closures.

Following what has already been done for the three scheduling operators, we remark in Lem.4.19 the distributivity of $\divideontimes$ w.r.t. the Kleene closures.

Lemma 4.19: Distributivity of $\divideontimes$ over the Kleene closures

For any $\diamond \in \{;,\ ;_{\divideontimes},\ ||\}$, for any set of traces $T \in \mathcal{P}(\mathbb{T}_\Omega)$ and any lifeline $l \in L$, we have:

$$\left(\exists\ t \in (T^{\diamond *} \setminus \{\epsilon\}),\ \neg(t \divideontimes l) \right) \Leftrightarrow \left(\exists\ t \in (T \setminus \{\epsilon\}),\ \neg(t \divideontimes l) \right)$$
$$(T^{\diamond *} \divideontimes l) \Leftrightarrow (T \divideontimes l)$$

Proof. As for Lem.4.14 the sketch of the proof relies on discussing the implications of the presence or absence of conflicts.

◯ In the first case there exists a non empty recomposed trace in $T^{\diamond *}$ with no conflicts w.r.t. l. This trace has been recomposed from at least one non empty trace from T in which there are no conflicts w.r.t. l.

◯ In the second case we simply remark that the presence of an action occurring on l in T implies its presence in $T^{\diamond *}$, and, reciprocally, if it is found in $T^{\diamond *}$ then it must be found in T. □

In Lem.4.20 we state that the set $\{\epsilon\}$ containing only the empty trace is a fixed-point w.r.t. any of the three Kleene closures.

Lemma 4.20: A fixed-point for the Kleene closures

For any $\diamond \in \{;,\ ;_{\divideontimes},\ ||\}$, we have:

$$\{\epsilon\}^{\diamond *} = \{\epsilon\}$$

Proof. Trivial. □

Comparing closures & idempotence properties

In Lem.4.18 from before, we have seen that the three scheduling operators can be characterized w.r.t. each other by inclusion. Given that the three Kleene closures are based on those operators, those properties of inclusions can be extended. We formalize this observation in Lem.4.21.

Lemma 4.21: Comparing the three Kleene closures

For any set of traces T, we have:

$$T^{;*} \subset T^{;\circledast *} \quad \text{and} \quad T^{;\circledast *} \subset T^{||*}$$

Proof. The Kleene closures are obtained from unions of the powersets of T. It then suffices to prove that, for any index $j \in \mathbb{N}$ we have:

$$T^{;j} \subset T^{;\circledast j} \quad \text{and} \quad T^{;\circledast j} \subset T^{||j}$$

Let us therefore reason by induction on j:

- we have $T^{;0} = T^{;\circledast 0} = T^{||0} = \{\epsilon\}$ therefore the property holds for $j = 0$
- we have $T^{;0} = T^{;\circledast 0} = T^{||0} = T$ therefore the property holds for $j = 1$
- let us suppose that $T^{;j} \subset T^{;\circledast j} \subset T^{||j}$ we then have:

$$\begin{array}{rcll} T^{;(j+1)} & = & T; T^{;j} & \text{by definition} \\ & \subset & T; T^{;\circledast j} & \text{by induction hypothesis} \\ & \subset & T;_{\circledast} T^{;\circledast j} & \text{by Lem.4.18} \\ & \subset & T^{;\circledast (j+1)} & \text{by definition} \end{array}$$

In the same manner, we have:

$$\begin{array}{rcll} T^{;\circledast (j+1)} & = & T;_{\circledast} T^{;\circledast j} & \text{by definition} \\ & \subset & T;_{\circledast} T^{||j} & \text{by induction hypothesis} \\ & \subset & T||T^{||j} & \text{by Lem.4.18} \\ & \subset & T^{||(j+1)} & \text{by definition} \end{array}$$

Hence the property holds.

□

A consequence of those properties of inclusions is given in Lem.4.22. It pertains to the composition of several Kleene closures. Those properties are akin to properties of idempotence.

Lemma 4.22: Idempotence properties of the Kleene closures

For any set of traces T, we have:

(i)	$(T^{;*})^{;*} = T^{;*}$	(iv)	$(T^{;\circledast *})^{;*} = T^{;\circledast *}$	(vii)	$(T^{||*})^{;*} = T^{||*}$
(ii)	$(T^{;*})^{;\circledast *} = T^{;\circledast *}$	(v)	$(T^{;\circledast *})^{;\circledast *} = T^{;\circledast *}$	(viii)	$(T^{||*})^{;\circledast *} = T^{||*}$
(iii)	$(T^{;*})^{||*} = T^{||*}$	(vi)	$(T^{;\circledast *})^{||*} = T^{||*}$	(ix)	$(T^{||*})^{||*} = T^{||*}$

Proof. Let us prove each predicate one by one.

(i) Let us prove both inclusions of the equality:

- $\subset$ Let us consider $t \in (T^{;*})^{;*}$. This implies the existence of $j_1 \geq 0$ s.t. $t \in (T^{;*})^{;j_1}$. This trace t is obtained from recomposing a finite number j_1 of traces from $T^{;*}$. As a result there exists an index j_2 such that all of those trace are included in $T^{;j_2}$. Therefore we have $t \in (T^{;j_2})^{;j_1}$ and as a result, given that ; is associative as per Lem.4.10, we have $t \in T^{;(j_1+j_2)}$ and therefore $t \in T^{;*}$
- $\supset$ Let us consider $t \in T^{;*}$. We immediately have that $t \in (T^{;*})^{;1}$ and therefore $t \in (T^{;*})^{;*}$

(ii) Let us prove both inclusions of the equality:

- $\subset$ Let us consider $t \in (T^{;*})^{;\circledast *}$. This implies the existence of $j_1 \geq 0$ s.t. $t \in (T^{;*})^{;\circledast j_1}$. This trace t is obtained from recomposing a finite number j_1 of traces from $T^{;*}$. As a result there exists an index j_2 such that all of those trace are included in $T^{;j_2}$. Then, as per Lem.4.21, this implies that those traces are also included in $T^{;\circledast j_2}$. Therefore we have $t \in (T^{;\circledast j_2})^{;\circledast j_1}$. As a result, given that $;_{\circledast}$ is associative as per Lem.4.16, we have $t \in T^{;\circledast (j_1+j_2)}$ and therefore $t \in T^{;\circledast *}$
- $\supset$ Let us consider $t \in T^{;\circledast *}$. We immediately have that $t \in (T^{;1})^{;\circledast *}$ and therefore $t \in (T^{;*})^{;\circledast *}$

(*) all the other points can be proven similarly

□

4.1.7 Operational-style characterizations of Kleene closures

In this section we will provide an operational-style characterization for the strict Kleene closure $^{;*}$ and the interleaving Kleene closure $^{||*}$. We will then show, using a counter-example that this characterization does not hold for the weak Kleene closure $^{;\circledast *}$.

Head-First closure

For any scheduling operator $\diamond \in \{;, \ ;_{\circledast}, \ ||\}$ whenever $a.t \in T_1 \diamond T_2$ (with a and t any action and trace and T_1 and T_2 any sets of traces), we may "take" the action a either from a trace of T_1 or from a trace of T_2.

The restricted scheduling $\diamond^{\neg}$ that we define in Def.4.11 correspond to a narrower definition of the corresponding scheduling operators, that do not allow the recomposition of traces which start with an action taken from a right-hand side set.

Definition 4.11: Restricted scheduling operators

For any $\diamond \in \{;,\ ;_*,\ ||\}$, we define $\diamond^{\curvearrowleft} : \mathcal{P}(\mathbb{T}_\Omega) \times \mathcal{P}(\mathbb{T}_\Omega) \to \mathcal{P}(\mathbb{T}_\Omega)$ such that for any sets of traces T_1 and T_2 we have:

$$T_1 \diamond^{\curvearrowleft} T_2 = \left\{ t \in T_1 \diamond T_2 \;\middle|\; (t = a.t') \Rightarrow \left(\exists\, t_1 \in \mathbb{T}_\Omega,\ \text{s.t.} \quad \begin{array}{l} (a.t_1 \in T_1) \\ \wedge\ (t' \in \{t_1\} \diamond T_2) \end{array} \right) \right\}$$

Let us consider the following example:

$$T_1 ;_* T_2 \quad \text{with} \quad T_1 = \left\{ \begin{array}{l} l_1!m_1, \\ l_2!m_3 \end{array} \right\} \quad \text{and} \quad T_2 = \left\{ \begin{array}{l} l_1!m_1, \\ l_1!m_2 \end{array} \right\}$$

Let us remark that $T_1 ;_* T_2 \neq T_2 ;_* T_1$. Indeed, we have for example that $l_1!m_1.l_1!m_2 \in T_1 ;_* T_2$ but it is not in $T_2 ;_* T_1$. Likewise, we have that $l_1!m_2.l_1!m_1 \in T_2 ;_* T_1$ but it is not in $T_1 ;_* T_2$.

By composition, we can compute the resulting set of traces as follows:

$$T_1 ;_* T_2 = \left(\begin{array}{l} (l_1!m_1 ;_* l_1!m_1) \\ \cup (l_1!m_1 ;_* l_1!m_2) \\ \cup (l_2!m_3 ;_* l_1!m_1) \\ \cup (l_2!m_3 ;_* l_1!m_2) \end{array} \right) = \left(\begin{array}{l} \left\{ l_1!m_1.l_1!m_1 \right\} \\ \cup \left\{ l_1!m_1.l_1!m_2 \right\} \\ \cup \left\{ \begin{array}{l} l_2!m_3.l_1!m_1, \\ l_1!m_1.l_2!m_3 \end{array} \right\} \\ \cup \left\{ \begin{array}{l} l_2!m_3.l_1!m_2, \\ l_1!m_2.l_2!m_3 \end{array} \right\} \end{array} \right) = \left\{ \begin{array}{l} l_1!m_1.l_1!m_1, \\ l_1!m_1.l_1!m_2, \\ l_2!m_3.l_1!m_1, \\ l_1!m_1.l_2!m_3, \\ l_2!m_3.l_1!m_2, \\ l_1!m_2.l_2!m_3 \end{array} \right\}$$

However, if we use $;_*^{\curvearrowleft}$ instead of $;_*$ we obtain the following:

$$T_1 ;_*^{\curvearrowleft} T_2 = \left(\begin{array}{l} \left\{ l_1!m_1.l_1!m_1 \right\} \\ \cup \left\{ l_1!m_1.l_1!m_2 \right\} \\ \cup \left\{ l_2!m_3.l_1!m_1 \right\} \\ \cup \left\{ l_2!m_3.l_1!m_2 \right\} \end{array} \right) = \left\{ \begin{array}{l} l_1!m_1.l_1!m_1, \\ l_1!m_1.l_1!m_2, \\ l_2!m_3.l_1!m_1, \\ l_2!m_3.l_1!m_2 \end{array} \right\}$$

We can then define head-first closures of scheduling operators using the notion of restricted scheduling and of Kleene closure as is done in Def.4.12.

Definition 4.12: Head-first closures

For any $\diamond \in \{;,\ ;_*,\ ||\}$, we define the Head-First closure of $\diamond$ as $\diamond^{\curvearrowleft *}$ i.e. the Kleene closure of the restricted $\diamond^{\curvearrowleft}$ operator.

The restricted operators that we have just defined can trivially be characterized w.r.t. their unrestricted counterparts (Lem.4.23).

Lemma 4.23: Relating the restricted operators to their unrestricted counterparts

For any $\diamond \in \{;,\ ;_{\ast},\ ||\}$, for any sets of traces T_1, T_2 and T we have:

$$(T_1 \diamond^{\curvearrowleft} T_2) \subset (T_1 \diamond T_2) \quad \text{and} \quad T^{\diamond^{\curvearrowleft}*} \subset T^{\diamond *}$$

Proof. The first point is immediate. The second is implied by the first. □

In a similar fashion to what we did in Lem.4.18 and Lem.4.21, we compare, in Lem.4.24, the different restricted operator we have just defined.

Lemma 4.24: Comparing the restricted operators

For any sets of traces T_1, T_2 and T we have:

$$(T_1 ;^{\curvearrowleft} T_2) \subset (T_1 ;_{\ast}^{\curvearrowleft} T_2) \quad \text{and} \quad (T_1 ;_{\ast}^{\curvearrowleft} T_2) \subset (T_1 ||^{\curvearrowleft} T_2)$$

$$\text{and} \quad T^{;^{\curvearrowleft}*} \subset T^{;_{\ast}^{\curvearrowleft}*} \quad \text{and} \quad T^{;_{\ast}^{\curvearrowleft}*} \subset T^{||^{\curvearrowleft}*}$$

Proof. The two first points are immediately implied by Lem.4.18 and by the definition of the restriction of scheduling operators (Def.4.11). The two last points are implied by the first two points, by the definition of the Head-First closures (Def.4.12) and by Lem.4.21. □

This notion of Head-First closure constitutes an operational-style counterpart to the notion of Kleene closure. Indeed, it provides a means to ensure that the occurrence of the first action in a trace $a.t$ comes from the expression of a behavior that is found in the first instance of the repeatable specification T.

In the following we will show that this notion of Head-First closure is equivalent to that of the Kleene closure for the strict sequencing ; and the interleaving || operators. However, as we will later see, those two notions are not equivalent whenever $\diamond = ;_{\ast}$ i.e. for the weak sequencing operator.

Characterizations for the strict and interleaving Kleene closures

In Lem.4.25 we show that, whenever $\diamond \in \{;,\ ||\}$, we can characterize the presence of a non empty trace $a.t$ in $T^{\diamond *}$ as the existence of a trace t' such that $a.t' \in T$ and $t \in \{t'\} \diamond T^{\diamond *}$.

Lemma 4.25: Operational characterization of $;^*$ and $||^*$ on traces

For any $\diamond \in \{;, ||\}$, for any set of traces $T \in \mathcal{P}(\mathbb{T}_\Omega)$, for any trace t in $\mathbb{T}_\Omega$ and for any action $a \in \mathbb{A}_\Omega$ we have:

$$(a.t \in T^{\diamond *}) \Rightarrow \left(\exists\, t' \in \mathbb{T}_\Omega \text{ s.t. } \begin{cases} (a.t' \in T) \\ \wedge\ (t \in (\{t'\} \diamond T^{\diamond *})) \end{cases} \right)$$

Proof. By definition of the Kleene closure, $a.t \in T^{\diamond *}$ implies the existence of $j \geq 0$ s.t. $a.t \in T^{\diamond j}$. We can then reason by induction on the power j:

- we cannot have $j = 0$ because $T^{\diamond 0} = \{\epsilon\}$
- if $j = 1$ then $a.t \in T^{\diamond 1} = T$ and $t \in \{t\} \diamond \{\epsilon\} \subset \{t\} \diamond T^{\diamond *}$ therefore the property holds
- if $j > 1$ the fact that $a.t \in T^{\diamond j} = T \diamond T^{\diamond (j-1)}$ implies the existence of $t'' \in T$ s.t. $a.t \in \{t''\} \diamond T^{\diamond (j-1)}$ then:
  - if $\diamond = ;$ we have, as per Lem.4.7 that:
    - either t'' is of the form $a.t'$ and $t \in \{t'\}; T^{;(j-1)}$ and therefore $t \in \{t'\}; T^{;*}$ and hence the property holds
    - or $t'' = \epsilon$ and $a.t \in T^{;(j-1)}$ In this case we can use the induction hypothesis so that the property holds
  - if $\diamond = ||$ we have, as per Lem.4.2 that:
    - either t'' is of the form $a.t'$ and $t \in \{t'\}||T^{||(j-1)}$ and therefore $t \in \{t'\}||T^{||*}$ and hence the property holds
    - or there exists a certain t''' s.t. we have $a.t''' \in T^{||(j-1)}$ and $t \in \{t''\}||\{t'''\}$. Let us then remark that given that we have $a.t''' \in T^{||(j-1)}$ we can apply the induction hypothesis to reveal t' such that $a.t' \in T$ and $t''' \in \{t'\}||T^{||*}$. We can then use the associativity and commutativity of $||$ as follows:

$$\begin{array}{rcll} t \in \{t''\}||(\{t'\}||T^{||*}) & \Rightarrow & t \in \{t''\}||(T^{||*}||\{t'\}) & \text{commutativity} \\ & \Rightarrow & t \in (\{t''\}||T^{||*})||\{t'\} & \text{associativity} \\ & \Rightarrow & t \in T^{||*}||\{t'\} & \text{property of Kleene closure} \\ & \Rightarrow & t \in \{t'\}||T^{||*} & \text{commutativity} \end{array}$$

  As a result, we have found t' such that the property holds.

□

From Lem.4.25 we can then immediately conclude that:

- the head-first closure $^{;^\curvearrowleft *}$ and the Kleene closure $^{;*}$ of the strict sequencing ; are equivalent
- the head-first closure $^{||^\curvearrowleft *}$ and the Kleene closure $^{||*}$ of the interleaving sequencing $||$ are equivalent

Lemma 4.26: Equivalence of HF & K closures for strict sequencing & interleaving

For any set of traces T we have:

$$T^{;^\curvearrowleft *} = T^{;*} \qquad \text{and} \qquad T^{||^\curvearrowleft *} = T^{||*}$$

Proof. As per Lem.4.23, we already have $T^{;^\curvearrowleft *} \subset T^{;*}$ and $T^{||^\curvearrowleft *} \subset T^{||*}$. There remains to prove the other inclusion. Let us then consider $\diamond \in \{;, \ ||\}$ and reason by induction on a member trace:

- for $t = \epsilon$ we have, by definition $\epsilon \in T^{\diamond^\curvearrowleft *}$ and $\epsilon \in T^{\diamond *}$ therefore both properties holds for an empty trace
- if the trace is of the form $a.t$ then if $a.t \in T^{\diamond *}$, then, as per Lem.4.25 there exists a trace t' such that $a.t' \in T$ and $t \in \{t'\} \diamond T^{\diamond *}$. This in turn implies that:
  - given that action a is taken from $a.t'$, we have $a.t \in \{a.t'\} \diamond^\curvearrowleft T^{\diamond *}$
  - and there exists $t'' \in T^{\diamond *}$ such that $t \in \{t'\} \diamond \{t''\}$. Given that t'' is strictly smaller than $a.t$, we can apply the induction hypothesis to obtain that $t'' \in T^{\diamond^\curvearrowleft *}$.

  Therefore we have that $a.t \in \{a.t'\} \diamond^\curvearrowleft T^{\diamond^\curvearrowleft *}$, and hence $a.t \in T^{\diamond^\curvearrowleft *}$

□

Counter-example for the weak Kleene closure

In the following we will show, using a counter example that the weak Kleene closure $^{;\circledast *}$ and the weak Head-First closure $^{;^\curvearrowleft_\circledast *}$ are not equivalent.

Let us consider the following example set of traces T:

$$T = \left\{ \begin{array}{l} l_1!m_1.l_2?m_1, \\ l_2!m_2 \end{array} \right\}$$

We will then show that $T^{;\circledast *} \notin T^{;^\curvearrowleft_\circledast *}$. To do that let us consider the powerset $T^{;\circledast 2}$ of T:

$$T^{;_\otimes 2} = \left\{ \begin{array}{l} l_1!m_1.l_2?m_1, \\ l_2!m_2 \end{array} \right\} ;_\otimes \left\{ \begin{array}{l} l_1!m_1.l_2?m_1, \\ l_2!m_2 \end{array} \right\} = \left(\begin{array}{l} (l_1!m_1.l_2?m_1 ;_\otimes l_1!m_1.l_2?m_1) \\ \cup (l_1!m_1.l_2?m_1 ;_\otimes l_2!m_2) \\ \cup (l_2!m_2 ;_\otimes l_1!m_1.l_2?m_1) \\ \cup (l_2!m_2 ;_\otimes l_2!m_2) \end{array} \right)$$

$$= \left(\begin{array}{l} \left\{ \begin{array}{l} l_1!m_1.l_2?m_1.l_1!m_1.l_2?m_1, \\ l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1 \end{array} \right\} \\ \cup \left\{ l_1!m_1.l_2?m_1.l_2!m_2 \right\} \\ \cup \left\{ \begin{array}{l} l_2!m_2.l_1!m_1.l_2?m_1, \\ l_1!m_1.l_2!m_2.l_2?m_1 \end{array} \right\} \\ \cup \left\{ l_2!m_2.l_2!m_2 \right\} \end{array} \right) = \left\{ \begin{array}{l} l_1!m_1.l_2?m_1.l_1!m_1.l_2?m_1, \\ l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1, \\ l_1!m_1.l_2?m_1.l_2!m_2, \\ l_2!m_2.l_1!m_1.l_2?m_1, \\ \underline{l_1!m_1.l_2!m_2.l_2?m_1}, \\ l_2!m_2.l_2!m_2 \end{array} \right\}$$

We have underlined the fact that $l_1!m_1.l_2!m_2.l_2?m_1 \in T^{;_\otimes 2}$ and therefore this trace is in $T^{;_\otimes *}$. Moreover, given that it contains one instance of each of the actions $l_1!m_1$, $l_2!m_2$ and $l_2?m_1$, and given that $\epsilon \notin T$, this means that this trace is obtained from repeating exactly two instances of T.

As a result, if this trace was in $T^{;_\otimes^\curvearrowleft *}$, it would be in $T ;_\otimes^\curvearrowleft T$. However, we have:

$$T ;_\otimes^\curvearrowleft T = \left(\begin{array}{l} \left\{ \begin{array}{l} l_1!m_1.l_2?m_1.l_1!m_1.l_2?m_1, \\ l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1 \end{array} \right\} \\ \cup \left\{ l_1!m_1.l_2?m_1.l_2!m_2 \right\} \\ \cup \left\{ \begin{array}{l} l_2!m_2.l_1!m_1.l_2?m_1 \end{array} \right\} \\ \cup \left\{ l_2!m_2.l_2!m_2 \right\} \end{array} \right) = \left\{ \begin{array}{l} l_1!m_1.l_2?m_1.l_1!m_1.l_2?m_1, \\ l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1, \\ l_1!m_1.l_2?m_1.l_2!m_2, \\ l_2!m_2.l_1!m_1.l_2?m_1, \\ l_2!m_2.l_2!m_2 \end{array} \right\}$$

Therefore, we have shown that $T^{;_\otimes *} \notin T^{;_\otimes^\curvearrowleft *}$. For a more detailed discussion on this topic, we can study the properties of the objects that correspond to the notions of weak Kleene closure and weak Head-First closure in the world of interaction terms. This will notably be addressed in Chap.5 in the context of the operational-style formulation of the trace semantics (more particularly one can refer to Sec.5.1.13 for details on the same counter-example).

Inclusions and properties of closures

Given that we have shown that $;_\otimes^\curvearrowleft *$ and $;_\otimes *$ are different operators, we can take an interest in how they relate to each-other, and, how $;_\otimes^\curvearrowleft *$ relates to $;*$ and $||*$.

In Lem.4.27, we precise the results from Lem.4.21 on the inclusions of the different closures.

Lemma 4.27: Comparing the four closures

For any set of traces T, we have:

$$T^{;*} \subset T^{;\overset{\curlywedge}{*}*} \subset T^{;*} \subset T^{||*}$$

Proof. Given the results from Lem.4.23 and Lem.4.21 there only remains to prove that $T^{;*} \subset T^{;\overset{\curlywedge}{*}*}$.

◯ Let us consider $t \in T^{;*}$. As per Lem.4.26, $t \in T^{;*}$ is equivalent to having $t \in T^{;\overset{\curlywedge}{*}*}$. Then, as per Lem.4.24, this implies that $t \in T^{;\overset{\curlywedge}{*}*}$ □

As an extension to the properties listed in Lem.4.22 we propose some additional idempotence properties related to the $;\overset{\curlywedge}{*}*$ operator in Lem.4.28

Lemma 4.28: Idempotence properties of the weak Head-First closure

For any set of traces T, we have:

(i) $(T^{;\overset{\curlywedge}{*}*})^{;\overset{\curlywedge}{*}*} = T^{;\overset{\curlywedge}{*}*}$ (ii) $(T^{;*})^{;\overset{\curlywedge}{*}*} = T^{;\overset{\curlywedge}{*}*}$ (v) $(T^{;\overset{\curlywedge}{*}*})^{;*} = T^{;\overset{\curlywedge}{*}*}$

(iii) $(T^{;*})^{;\overset{\curlywedge}{*}*} = T^{;*}$ (vi) $(T^{;\overset{\curlywedge}{*}*})^{;\overset{\curlywedge}{*}*} = T^{;*}$

(iv) $(T^{||*})^{;\overset{\curlywedge}{*}*} = T^{||*}$ (vii) $(T^{;\overset{\curlywedge}{*}*})^{||*} = T^{||*}$

Proof. Similar proof to that of Lem.4.22. □

4.2 Syntax & Denotational Semantics

In the previous section, we have introduced the semantic domain $\mathcal{P}(\mathbb{T}_\Omega)$ (defined up to a signature Ω), its main operators and their algebraic properties. In this section, we introduce the syntax of our Interaction Language (IL) and its associated trace semantics:

- in Sec.4.2.1 we progressively introduce the different elements that constitute the IL, from its basic building blocs to the constructors that allow the definition of more complex interactions.
- in Sec.4.2.2 we formalize the syntax of the IL as a term algebra
- in Sec.4.2.3, we define a denotational-style trace semantics for the IL.

The syntax and denotational-style semantics presented in this section is partly inspired from [75].

4.2.1 Informal description of the syntax

Interactions are defined up to a given signature $\Omega = (L, M)$. Interaction terms are built by composition from a set of basic building blocks and using some constructors to define more complex terms. We use here the word "constructor" (in the domain of the syntax) so as not to confuse the reader with the usage of the word "operator" in the semantic domain.

Any interaction i constitutes a specification for a certain set of behaviors that it may express. During the execution of an interaction it must express a behavior that is found in that set, but it may express any one of those (non-deterministically).

Given that behaviors are modelled by traces from the semantic domain $\mathbb{T}_\Omega$, the set of behaviors, or "semantics", that is associated to an interaction i is a non-empty subset of $\mathbb{T}_\Omega$. This means, that the semantic domain of that language is $\mathcal{P}(\mathbb{T}_\Omega)$, which we have described in the previous section.

Basic building blocks

Communication actions (from $\mathbb{A}_\Omega$) constitute basic building blocks for both the traces of execution (i.e. $\mathbb{T}_\Omega$) and the interaction terms (which constitute the syntax), with the addition of a special empty interaction $\varnothing$.

	empty	emission	reception
diagram			
syntax	$\varnothing$	$l!m$	$l?m$
traces	$\{\epsilon\}$	$\{l!m\}$	$\{l?m\}$

Figure 4.1: Basic building blocks of interactions

As illustrated on Fig.4.1, we consider 3 possible kinds of atomic interactions:

- the empty interaction, noted $\varnothing$, and represented by an empty sequence diagram, models the empty behavior i.e. the behavior in which no observable event occurs. As a result, its semantic, i.e. the set of traces that it may express is the singleton $\{\epsilon\}$, where ϵ is the empty trace.
- the other building blocks are all the possible atomic reception and emission actions (defined up to the signature Ω). The semantics of an atomic action $a \in \mathbb{A}_\Omega$ (when considered as an atomic interaction) is the singleton $\{a\}$, i.e. it contains a single trace representing the occurrence of action a once globally. When it comes to drawing such atomic interactions, we distinguish between:
  - atomic emissions of the form $l!m$. Those can be drawn in a sequence diagram as an horizontal arrow exiting lifeline l and carrying message m, with the arrowhead being drawn in an empty space, meaning the message is send to the external environment of the distributed system.
  - atomic receptions of the form $l?m$. Those can be drawn in a sequence diagram as an horizontal arrow entering lifeline l and carrying message m, with the tail of the arrow being in empty space - meaning the message comes from the external environment of the distributed system - and the arrowhead touching lifeline l.

In the following, we introduce the different constructors that can be defined so as to build more complex interactions.

Scheduling constructors

Scheduling constructors are binary constructors, and as such, specify interactions of the form $f(i_1, i_2)$ where f is the given scheduling constructors, i_1 is an interaction called the left sub-interaction, and i_2 is another interaction called the right sub-interaction. A behavior expressed by such a $f(i_1, i_2)$ interaction, is a certain composition of a behavior expressed by i_1 and of another behavior expressed by i_2. Given that those behaviors are traces, one can define three scheduling constructors corresponding to the three scheduling operators ";", ";$_*$" and "||" from the previous section.

Given that the semantic domain is $\mathcal{P}(\mathbb{T}_\Omega)$ all the behaviors specified by i_1 and i_2 correspond to two sets of traces T_1 and T_2. The semantics of $f(i_1, i_2)$ can simply be obtained by composition as $T_1 \diamond T_2$ for a certain scheduling operator $\diamond$.

In more details, we distinguish between the three scheduling constructors:

- *strict* for "strict sequencing" such that the semantics of $strict(i_1, i_2)$ is $T_1; T_2$ whenever T_1 and T_2 respectively are the semantics of i_1 and i_2
- *seq* for "weak sequencing" such that the semantics of $seq(i_1, i_2)$ is $T_1;_* T_2$ whenever T_1 and T_2 respectively are the semantics of i_1 and i_2
- *par* for "parallelization" such that the semantics of $par(i_1, i_2)$ is $T_1||T_2$ whenever T_1 and T_2 respectively are the semantics of i_1 and i_2

Fig.4.2 illustrates the use of those constructors on some examples:

- in $strict(i_1, i_2)$, for any trace expressed by sub-interaction i_1, all the actions from this trace must occur before any action from any trace expressed by i_2 may occur. In other terms, sub-interaction i_1 must be entirely executed before sub-interaction i_2 may start to be executed. In the column on the left, two examples are provided for the use of *strict*:
  - in $strict(l!m_1, l!m_2)$ (the example at the top of the left column), action $l!m_1$ must occur before action $l!m_2$. This results, in the semantics $\{l!m_1.l!m_2\}$, which only contains one trace, for the example interaction.
  - in $strict(l_1!m_1, l_2!m_2$ (the example at the bottom of the left column) we have likewise a single trace that is specified i.e. $l_1!m_1.l_2!m_2$.

  In both examples, there is a single possible total order between the actions that is authorized, resulting in a single accepted trace. As illustrated on Fig.4.2, the representation of a *strict* constructor in the drawn sequence diagram is that of a square with two subsections divided horizontally by a line. In the top one is drawn the left sub-interaction, and on the bottom one is drawn the right sub-interaction of the interaction term. We draw a `strict` label on the top left corner of the square to indicate the use of the *strict* constructor, so as not to confuse the drawing with that of other constructors, which also use the square representation.

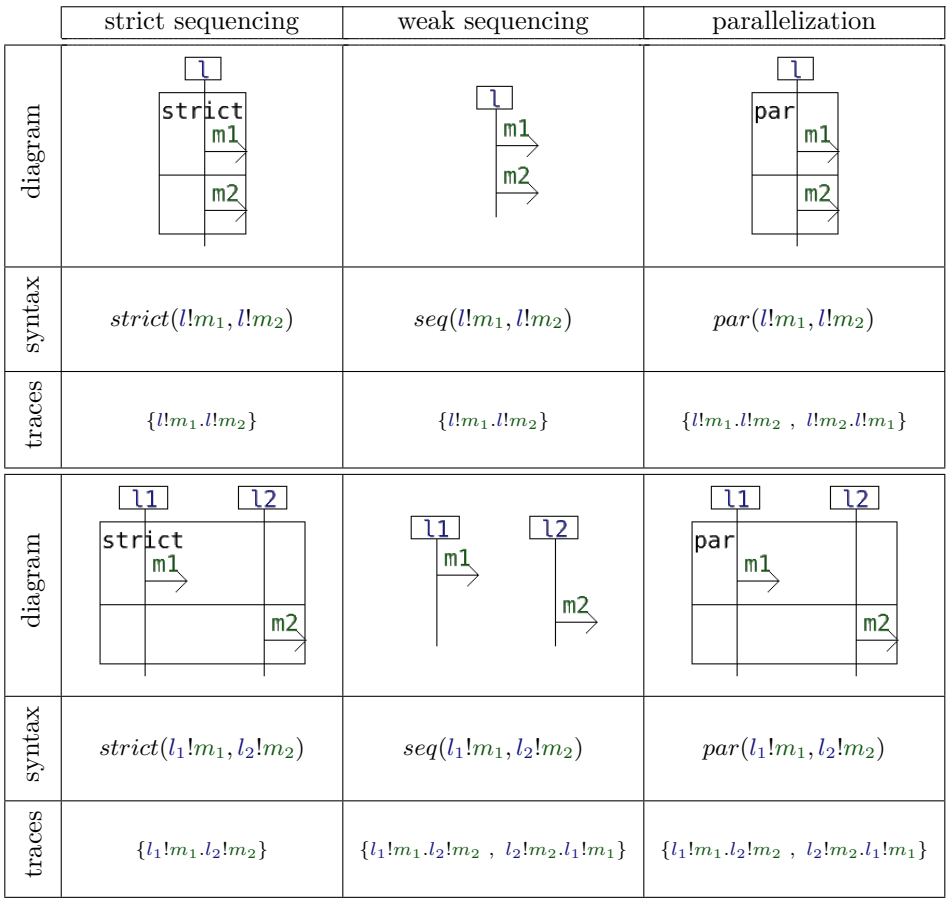


Figure 4.2: Scheduling constructors

- in $seq(i_1, i_2)$ sequentiality is only enforced between actions that occur on the same lifeline. This "weak" sequentiality is particularly apt at representing globally the behaviors of the locally defined sub-systems of a distributed system. Indeed, given that they are independent and run concurrently w.r.t. one another, there is a priori no reason why there should exist a strict order between events occurring on different sub-systems. The execution of actions on each lifeline can occur (from top to bottom in the drawn sequence diagram representations) independently. In the column on the middle of Fig.4.2, two examples are provided for the use of *seq*:

  – in $seq(l!m_1, l!m_2)$ (the example at the top of the middle column), action $l!m_1$ must occur before action $l!m_2$. Here, using *seq* equates to using *strict* i.e. we obtain the same semantics $\{l!m_1.l!m_2\}$. This is because both actions occur on the same lifeline l.

  – however, in $seq(l_1!m_1, l_2!m_2)$, actions $l_1!m_1$ and $l_2!m_2$ occur on distinct lifelines l_1 and l_2. As a result they may occur in any order w.r.t. one another. Consequently, the semantics of that interaction contains two traces: $\{l_1!m_1.l_2!m_2, \ l_2!m_2.l_1!m_1\}$.

  As illustrated on Fig.4.2, in contrast with *strict*, the representation of a *seq* constructor in the drawn sequence diagram does not include the square that surrounds the sub-interactions. The left sub-interaction is simply drawn on top of the right one. This coincides with the fact that, in the graphical format of sequence diagrams, the top to bottom direction is usually associated with the passing of time.

The implicit sequencing of the top to bottom direction (i.e. what is drawn higher occurs "before", and what is drawn underneath occurs "after"), that is often only described in natural language, exactly corresponds to the *seq* weak sequencing constructor.

- in $par(i_1, i_2)$, any interleaving between the executions of i_1 and i_2 is allowed. In the column on the right of Fig.4.2, two examples are provided for the use of *par*:
  - in $par(l!m_1, l!m_2)$ (the example at the top of the right column), actions $l!m_1$ and $l!m_2$ can occur in any order. We therefore obtain the semantics $\{l!m_1.l!m_2,\ l!m_2.l!m_1\}$.
  - in $par(l_1!m_1, l_2!m_2)$, we have likewise that actions $l_1!m_1$ and $l_2!m_2$ can occur in any order. We then have the semantics $\{l_1!m_1.l_2!m_2,\ l_2!m_2.l_1!m_1\}$, which is the same as what we would have obtained if we used the *seq* constructor here, because both actions occur on different lifelines.

  With the examples from Fig.4.2, we can see that for both examples, we can have either the left action being executed at first and then the right action being executed afterwards, or the opposite. The *par* constructors represents, in facts, the parallel execution of two distinct and independent sub-interactions. As for *strict*, the *par* constructor is represented graphically by using the square notation, with a `par` label on the top left corner.

Within the scheduling constructors, *seq* can be considered to be a hybrid, at the crossroads between *strict* and *par*. Indeed, it behaves like *strict* when we consider local behaviors (i.e. the orders of execution that are enforced between the actions occurring on a same localization) but when we consider global behaviors, it is more akin to *par*.

Message passing and broadcasts

We can use scheduling constructors to model specific patterns that are characteristic of the exchange of messages within a distributed system. Two of those are illustrated on Fig.4.3.

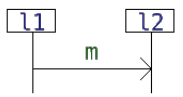
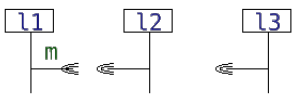
	message passing	broadcast
diagram		
syntax	$strict(l_1!m, l_2?m)$	$strict(l_1!m, seq(l_2?m, l_3?m))$
traces	$\{l_1!m.l_2?m\}$	$\{l_1!m.l_2?m.l_3?m\ ,\ l_1!m.l_3?m.l_2?m\}$

Figure 4.3: Message passing & broadcast

In Fig.4.3 we have:

- on the example on the left, the illustration of the passing of a message from an origin lifeline (emitter) to a target lifeline (receiver). This passing of message can be modelled by the pattern $strict(l_1!m, l_2?m)$

i.e. we use the *strict* operator to schedule the reception of message m on the receiver lifeline l_2 w.r.t. its previous emission from the emitter lifeline l_1.

- on the example on the right, the illustration of the broadcast of a message from an origin lifeline (emitter) to a set of target lifelines (receivers).

Let us note that, of course, given the nature of this IL, there can be several distinct manners to model a broadcast, in addition to the one presented in Fig.4.3. We could have used a *par* constructor instead of the *seq* (i.e. $strict(l_1!m, par(l_2?m, l_3?m))$ or we could have inverted the order of $l_2?m$ and $l_3?m$ in the term (i.e. $strict(l_1!m, seq(l_3?m, l_2?m))$. Those changes would not have made any differences as for the semantics (sets of accepted traces).

If recognized within an interaction term, those specific patterns can be drawn in a dedicated manner. For the passing of a message m from lifeline l_1 to lifeline l_2, as in Fig.4.3, we can draw an horizontal arrow carrying m from the representation of lifeline l_1 to that of lifeline l_2. For broadcasts, we can draw, as in Fig.4.3, on the same alignment horizontally, a number of small arrows with recognizable heads, one for each of the involved lifelines. The emitter lifeline has an arrow which head goes outwards (i.e. towards the environment) while all receiver lifelines have their arrowheads going inwards (i.e. towards the lifeline).

Choice constructors

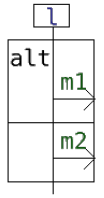
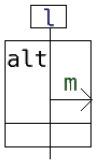
diagram		
syntax	$alt(l!m_1, l!m_2)$	$alt(l!m, \varnothing)$
traces	$\{l!m_1\ ,\ l!m_2\}$	$\{l!m\ ,\ \epsilon\}$

Figure 4.4: Alternative constructor

In addition to the scheduling constructors, one can also define choice constructors, which, instead of scheduling the behaviors expressed by sub-interactions, select them. In our IL we propose a single choice constructor which is the non-deterministic alternative "*alt*" constructor.

alt is a binary constructor which specifies an exclusive alternative between 2 sub-interactions. In $alt(i_1, i_2)$, any behavior specified by either the left sub-interaction i_1 or by the right sub-interaction i_2 is acceptable. In Fig.4.4, the use of the *alt* constructor is illustrated on 2 examples:

- in $alt(l!m_1, l!m_2)$ (the example on the left), if $l!m_1$ occurs then $l!m_2$ does not occur and vice-versa. This leads to a semantics containing two traces: $\{l!m_1,\ l!m_2\}$.

- in $alt(l!m, \varnothing)$ (the example on the right), action $l!m$ may occur, or may not occur, because the empty interaction is selected. This likewise lead to the acceptance of two traces: $\{l!m,\ \epsilon\}$.

In effect, the semantics of an interaction of the form $alt(i_1, i_2)$ is an union of that of i_1 and i_2 i.e. it is $T_1 \cup T_2$ whenever T_1 and T_2 respectively are the semantics of i_1 and i_2.

Similarly to *strict* and *par*, the *alt* constructor is represented graphically by using the square notation, with a `alt` label on the top left corner.

Repetition constructors

Repetition constructors, also called loops, specify the possible repetition (any number of times) of a given sub-interaction. As such, an interaction $loop_k(i)$ (with k being the "kind" of the loop of which we define four) synthesizes into a finite interaction term the specification of any arbitrary repetitions of behaviors specified by the sub-interaction i.

As the reader may expect, those loop constructors are associated to the three Kleene closures $^{;*}$, $^{;\circledast *}$ and $^{||*}$ and the weak Head-First closure $^{\overset{\curvearrowright}{;}*}$ operators from the previous section. Therefore we define four loop constructors such that, whenever T is the semantics of an interaction i then the semantics of:

- $loop_S(i)$ is $T^{;*}$ and we may call $loop_S$ the strict loop constructor
- $loop_W(i)$ is $T^{;\circledast *}$ and we may call $loop_W$ the weak loop constructor
- $loop_P(i)$ is $T^{||*}$ and we may call $loop_P$ the parallel or interleaving loop constructor
- $loop_H(i)$ is $T^{\overset{\curvearrowright}{;}*}$ and we may call $loop_H$ the head loop constructor

We can describe those operators as loops or repetition operators because they enable the repetition, any number of times, and according to a certain scheduling operator, of behaviors:

- $loop_S$ uses strict sequencing as a scheduler
- $loop_P$ uses interleaving as a scheduler
- both $loop_W$ and $loop_H$ use weak sequencing as a scheduler. However, $loop_H$ restrict accepted traces to those in which one take the first action (the head) in the first instance of the sub-behavior while $loop_W$ has no such restriction.
- As a side note, let us remark that there is no use in defining a "$loop_{alt}$" constructor i.e. using the *alt* constructor as a scheduler for the repetition of the behavior. Indeed, if we consider $loop_{alt}(i)$, such a loop is simply equivalent to $alt(i, \varnothing)$. If we choose to repeat 0 times the sub-interaction i then we have the $\varnothing$ alternative. If we choose to repeat it once, we get i and if twice, we get $alt(i, i)$ which equates having i.

The pertinence of defining four loops can be intuitively understood given that the three scheduling operators indeed have different semantics and, as a result, repeating their use should also result in different

semantics. As for the $loop_H$ constructor, we can link it to the unique $\overset{\circlearrowleft}{;}{}^*$ operator which we have shown to be distinct from $\overset{}{;}{}^*$. In the following we illustrate the distinctness (and hence pertinence) of the four loops on simple examples.

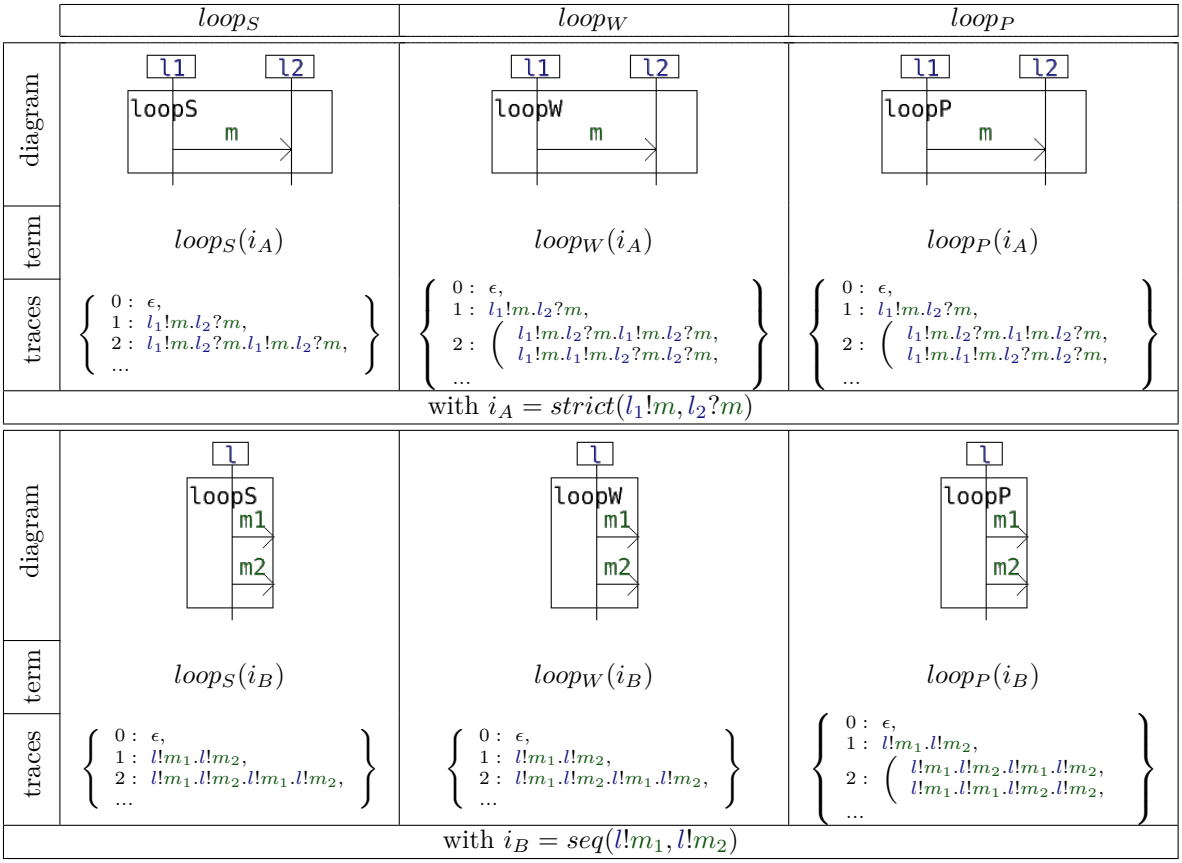


Figure 4.5: Comparing the $loop_S$, $loop_W$ & $loop_P$ repetition constructors

On Fig.4.5, we use $loop_S$, $loop_W$ and $loop_P$ to repeat two example interactions which are $i_A = strict(l_1!m, l_2?m)$ and $i_B = seq(l!m_1, l!m_2)$. In each of the six cells of the table from Fig.4.5, we have on top the diagram representation of an interaction, in the middle the corresponding term and on the bottom a partial representation of its associated semantics. The three top cells correspond to $loop_S(i_A)$, $loop_W(i_A)$ and $loop_P(i_A)$ while the three bottom cells correspond to $loop_S(i_B)$, $loop_W(i_B)$ and $loop_P(i_B)$.

We can remark that the graphical representation of a loop is that of a square, containing the representation of the nested sub-interaction and decorated with a label on the top left corner, which is either `loopS`, `loopH`, `loopW` or `loopP` depending on the type of loop.

For each example interaction of Fig.4.5, its semantics is, as mentioned before, partially represented i.e. a subset of the semantics is detailed:

- in all six cases the empty trace ϵ belongs to the semantics. It indeed corresponds to the repetition zero times of the sub-interaction (either i_A or i_B)
- when the sub-interaction is repeated once, we get, for the three cases from the top line (i.e. corresponding to i_A) the subset $\{l_1!m.l_2?m\}$, and, for the three cases from the bottom line (i.e. corresponding to i_B) the subset $\{l!m_1.l!m_2\}$

- when the sub-interaction is repeated twice we can observe that:
  - in the case where the sub-interaction that is repeated is i_A (top row), the semantics of $loop_W(i_A)$ and $loop_P(i_A)$ remain the same, but the semantics of $loop_S(i_A)$ is different from the others, only allowing strictly less traces
  - in the case where the sub-interaction that is repeated is i_B (bottom row), the semantics of $loop_S(i_B)$ and $loop_W(i_B)$ remain the same, but the semantics of $loop_P(i_B)$ is different from the others, allowing more traces

As for the head loop operator $loop_H$ it mostly behaves as $loop_W$ except on certain cases due to the restriction imposed on the first action that is taken. $loop_H$ is indeed associated to the weak Head-First closure operator $\overset{\curvearrowleft}{;}{}^*$.

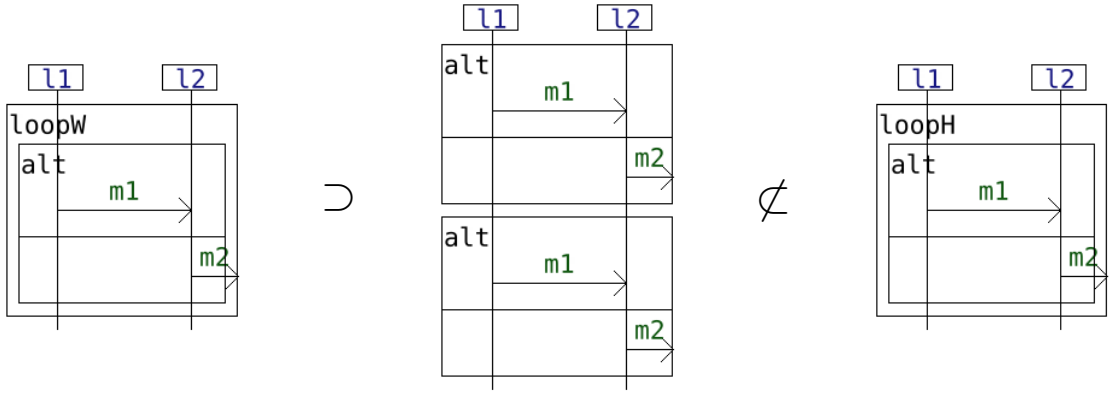


Figure 4.6: Illustrating the difference between the $loop_W$ & $loop_H$ constructors

On page 92, we have introduced an example in the semantic domain $\mathcal{P}(\mathbb{T}_\Omega)$ demonstrating the distinctness of $\overset{\curvearrowleft}{;}{}^*$ and $;^*$. On Fig.4.6 is illustrated its counterpart in the world of interaction terms. On the left, we have an interaction $i_A = loop_W(i_1)$ where $i_1 = alt(strict(l_1!m_1, l_2?m_1), l_2!m_2)$, in the middle we have an interaction $i_B = seq(i_1, i_1)$ and on the right we have $i_C = loop_H(i_1)$. We can remark that the trace $l_1!m_1.l_2!m_2.l_2?m_1$ belongs to the semantics of i_B given that we can choose to execute at first the occurrence of $l_1!m_1$ from the second instance of i_1 i.e. the one underneath. By definition, this trace also belongs to the semantics of i_A. However it does not belong to that of i_C because it corresponds to the case of taking the first action $l_1!m$ from the second instance of behavior i_1.

Even though the $loop_H$ is not associated to a Kleene closure contrary to the three other loops, we have decided to include it for two reasons:

- because it is indeed a unique artifact, distinct from the three other loops. Indeed, as we have seen in Sec.4.1.7 while the HF-closures for the strict sequencing and the interleaving operators are equivalent to the K-closures of the same operators (Lem.4.26) this is not the case for weak sequencing.
- and also because it corresponds to a notion of "loop" which may appear more intuitive

As a result of those observations, one can conclude that the four constructors $loop_S$, $loop_H$, $loop_W$ and $loop_P$ are indeed different. From a system designer perspective, using either loop is motivated by different goals:

- With $loop_S(i)$, each existing instance of a repeatable behavior (specified by i) must be executed entirely before any other can start. This implies that, at any given moment there can only exists zero or a single instance. $loop_S$ can therefore be used to specify some critical repeatable behavior of which there can only exist one instance at a time.
- With $loop_P(i)$, all existing instances can be executed concurrently w.r.t. one another, and, at any given moment, new instances can be created. $loop_P$ can therefore be used to specify protocols in which any number of new sessions can be created and run in parallel.
- With $loop_W(i)$, new instances can be created whenever the action triggering the instantiation occurs on a lifeline which is not occulted by previous instances. This can be roughly explained as follows: (1) on each individual lifeline only one instance can be active and (2) given that, for any such instance, a lifeline might "have finished" before the others then it is allowed to start another instance. $loop_W$ can therefore be used to specify repeatable behaviors that are sequential but that have no enforced synchronization mechanisms.
- The head loop $loop_H$ is associated to the weak HF-closure operator $\overset{\circlearrowleft}{;}{}^*$ which is an ad-hoc algebraic artifact which might correspond to a more intuitive understanding of sequential loops than $loop_W$.

4.2.2 Formal syntax

Now that we have informally introduced all the building blocks and constructors that constitute the syntax of our Interaction Language (IL), let us provide a full formalization.

Definition 4.13: Interaction Language

The set $\mathbb{I}_\Omega$ of interactions defined over the signature $\Omega = (L, M)$ is the set of ground terms $\mathcal{T}_\mathcal{F}$ associated to the term algebra of signature $\mathcal{F}$ such that:

$$\mathcal{F}_0 = \mathbb{A}_\Omega \cup \{\varnothing\} \qquad \mathcal{F}_2 = \{strict,\ seq,\ par,\ alt\}$$
$$\mathcal{F}_1 = \{loop_S,\ loop_H,\ loop_W,\ loop_P\} \qquad \mathcal{F}_j = \emptyset \text{ for any } j > 2$$

In Def.4.13 we define our IL as that of the ground terms of a term algebra. This allows us to use notations and benefit from results from the fields of equational logic and term rewriting which we introduced in Sec.2.2.

A first benefit of this presentation is that of structural induction. In order to prove a given property ψ on all interactions terms, it suffices to prove it for the empty interaction $\varnothing$ and for any action $a \in \mathbb{A}_\Omega$ and then to prove that for any interaction of the form $i = f(i_1, i_2)$ and $i = loop_k(i_1)$ (given $f \in \{strict, seq, par, alt\}$ and $k \in \{X, H, S, P\}$) if i_1 and i_2 satisfy ψ then i satisfies ψ.

A second benefit is the ability to represent interactions as trees. Given that, in $\mathbb{I}_\Omega$, the arity of the operation symbols is at most 2, words describing positions within interaction terms are in $\{1,2\}^*$. Using notations from Sec.2.2, we may denote by $pos(i)$ the set of positions associated to an interaction term i.

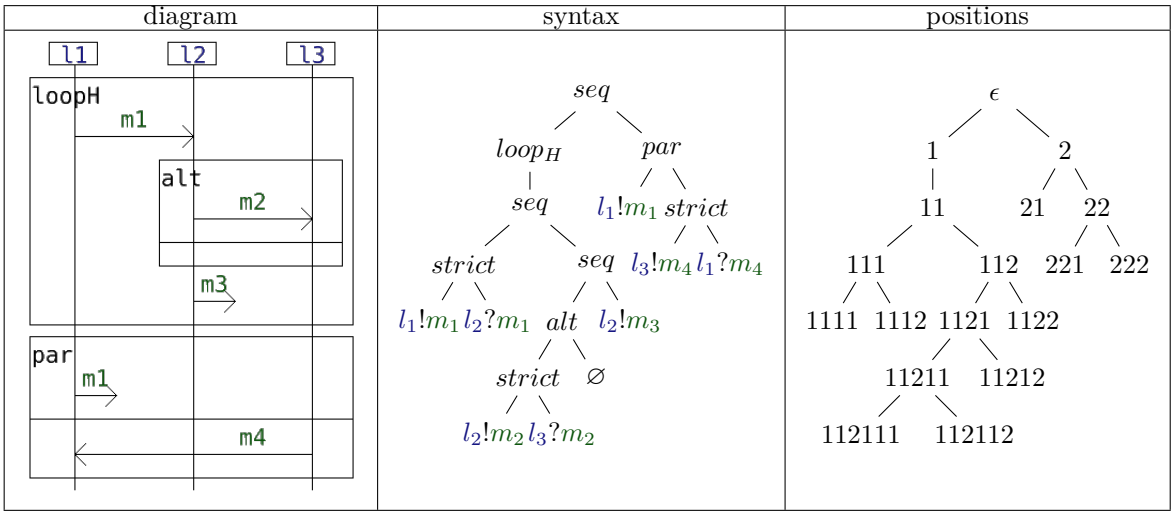


Figure 4.7: Describing interactions as trees

Fig.4.7 illustrates the principle of describing interactions as trees on an example. On the left is drawn a sequence diagram. In the middle is a syntactic term in our interaction language which corresponds to this diagram and which is represented as a tree. On the right is represented a tree structure corresponding to that of the syntactic term, with each node being replaced by its position within the tree:

- the empty position ϵ is that of the root node of the interaction term / tree
- whenever a node at position $p \in \{1,2\}^*$ has a left child or a unique child (in the case of loop constructors), then this child is at position $p.1$
- whenever a node at position $p \in \{1,2\}^*$ has a right child, then this child is at position $p.2$

As a result, and roughly speaking, starting from the empty position ϵ, adding "ones" at the end corresponds to going to the left and adding "twos" at the end corresponds to going to the right. In the example from Fig.4.7, the root node of the interaction hosts a *seq* constructor and is at position ϵ. The node at its left hosts a $loop_H$ constructor and is at position 1 while the node at its right hosts a *par* constructor and is at position 2.

Those positions allow the unambiguous designation of any sub-tree in the tree structure and therefore, we can pinpoint any sub-terms of an interaction. Considering the inductive nature of the language, any such sub-term is in itself a fully-fledged interaction, which we may call a "sub-interaction".

Using the notations from Sec.2.2, for any interaction i and any position $p \in pos(i)$, $i_{|p}$ refer to the sub-interaction of i which root node is at position p within i. Fig.4.8 illustrates the use of the $i_{|p}$ notation by highlighting three different sub-interactions both in the syntactic term on the right and on the diagram representation on the left. Those three sub-interactions are:

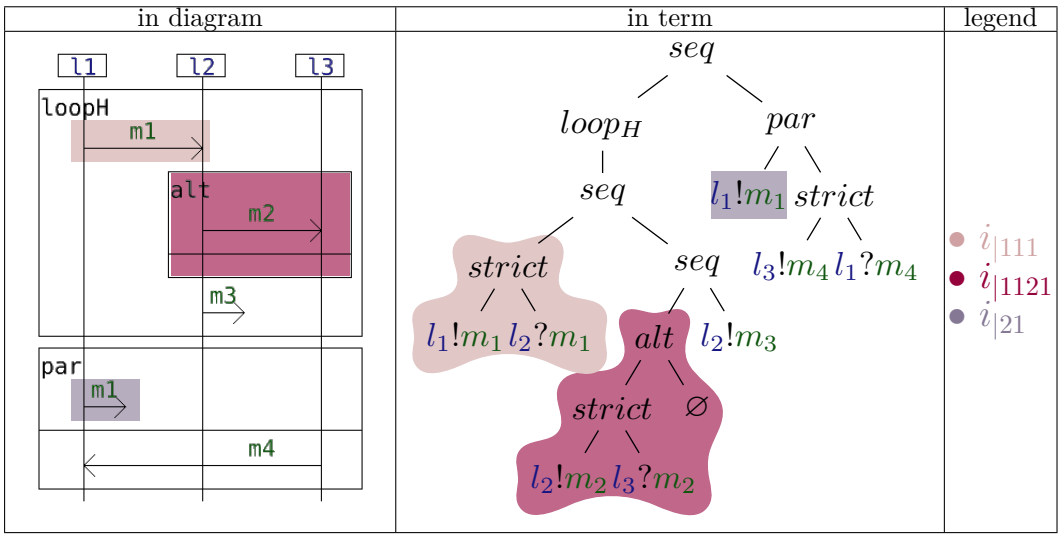


Figure 4.8: Illustrating sub-interactions within an interaction

- $i_{|111}$ which root node (a *strict*) is at position $p = 111$ w.r.t. the overall interaction i
- $i_{|1121}$ which root node (an *alt*) is at position $p = 1121$ w.r.t. i
- $i_{|21}$ which root node (an atomic action) is at position $p = 21$ w.r.t. i

4.2.3 Denotational semantics

As the reader may expect, we can define a denotational semantics of interactions as a homomorphism between the semantic domain $\mathcal{P}(\mathbb{T}_\Omega)$, which admits the structure of a $\mathcal{F}$-algebra and the term algebra $\mathbb{I}_\Omega$ of interaction terms.

Let us indeed consider the carrier $\mathbf{A} = \mathcal{P}(\mathbb{T}_\Omega)$ i.e. the set of sets of traces. On this carrier $\mathbf{A} = \mathcal{P}(\mathbb{T}_\Omega)$ we have the following operators:

the union $\cup$	the strict K-closure $^{:*}$
the strict sequencing ;	the weak HF-closure $^{;\overset{\frown}{*}*}$
the weak sequencing $;_*$	the weak K-closure $^{;*}$
the interleaving $\|\|$	the interleaving K-closure $^{\|\|*}$

We can see that those operators can be described as interpretations of the constructors from $\mathcal{F}$ the signature of the term algebra $\mathcal{T}_\mathcal{F} = \mathbb{I}_\Omega$ of interaction terms.

In addition, for the basic building blocks $\varnothing$ (the empty interaction) and actions $a \in \mathbb{A}_\Omega$, which are constants of $\mathcal{F}_0$, we can match them respectively with $\{\epsilon\}$, meaning that the semantics of $\varnothing$ only contains a single trace which is the empty trace (observation of no action) and with $\{a\}$, meaning that the semantics of a (interaction reduced to a single action) only contains a single trace which is a (observation of a single action).

In Def.4.14 we formalize in this manner the $\mathcal{F}$-algebra of sets of traces.

Definition 4.14: Semantic domain $\mathcal{P}(\mathbb{T}_\Omega)$ as a $\mathcal{F}$-algebra

$\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega),\ \mathcal{F}^{\mathcal{A}})$ (with $\mathcal{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathcal{F}\}$) is the $\mathcal{F}$-algebra defined by its carrier $\mathcal{P}(\mathbb{T}_\Omega)$ and the following interpretations of the operation symbols in $\mathcal{F}$:

$$
\begin{array}{llllll}
 & & strict^{\mathcal{A}} = ; & & loop_S^{\mathcal{A}} = & ;^* \\
\varnothing^{\mathcal{A}} = \{\epsilon\} & & seq^{\mathcal{A}} = ;_\otimes & & loop_H^{\mathcal{A}} = & ;_\otimes^{\circlearrowleft *} \\
a^{\mathcal{A}} = \{a\} & & par^{\mathcal{A}} = \| & & loop_W^{\mathcal{A}} = & ;_\otimes^* \\
 & & alt^{\mathcal{A}} = \cup & & loop_P^{\mathcal{A}} = & \|^*
\end{array}
$$

We then define in Def.4.15 a denotational-style semantics σ_d of interactions as the unique homomorphism between $\mathcal{T}_\mathcal{F} = \mathbb{I}_\Omega$ and $\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega),\ \mathcal{F}^{\mathcal{A}})$. Fig.4.9 illustrates this definition by representing σ_d as a morphism which preserves algebraic structures.

Definition 4.15: Denotational semantics of interactions

The denotational semantics σ_d of $\mathbb{I}_\Omega$ is the unique $\mathcal{F}$-homomorphism $\sigma_d : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ between $\mathcal{T}_\mathcal{F} = \mathbb{I}_\Omega$ and $\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega),\ \mathcal{F}^{\mathcal{A}})$

This denotational-style semantics constitutes the mathematical foundation for some other contributions of this present thesis, notably:

- we use it as a reference to justify the correctness of another semantics, defined in operational-style, that will be introduced in Chap.5. A proof for the equivalence of both semantics is presented in Sec.5.2.
- it serves as the basis to characterize an equivalence relation between interaction terms and to compute normal forms via rewriting. This will be covered in the next section.

$$
\left(\mathbb{I}_\Omega, \left\{ \begin{array}{l} \varnothing,\ a \in \mathbb{A}_\Omega, \\ alt,\ strict,\ seq,\ par \\ loop_S,\ loop_H,\ loop_W,\ loop_P \end{array} \right\}\right)
$$

$$
\text{homomorphism } \sigma_d \Big\downarrow \text{(denotational semantics)}
$$

$$
\left(\mathcal{P}(\mathbb{T}_\Omega), \left\{ \begin{array}{l} \{\epsilon\},\ \{a\}, \\ \cup,\ ;,\ ;_\otimes,\ \|, \\ ;^*,\ ;_\otimes^{\circlearrowleft *},\ ;_\otimes^*,\ \|^* \end{array} \right\}\right)
$$

Figure 4.9: The denotational semantics as a homomorphism

4.3 Normal forms of interactions

In this section we use results from the domain of (1) equational theory [25, 71] so as to define classes of interactions which have the same semantics, and (2) term rewriting [51, 68] so as to define normal forms of

interactions.

Those two points are related in so far as whenever two interactions have the same normal form (defined up to a certain rewrite system) then they are equivalent (up to a certain equational theory).

Working with normal forms instead of general interactions can be useful in various ways. It can notably help reduce the complexity of some problems in which one can explore the space of normal forms instead of exploring the larger space of all interaction terms.

This section is organised as follows:

- in Sec.4.3.1 we define an axiom system on $\mathbf{I}_\Omega$ that is sound w.r.t. $\mathcal{A}$,
- in Sec.4.3.2 we define a process to normalize interactions modulo Associativity-Commutativity,
- in Sec.4.3.3 we present automated proofs for the convergence of this process i.e. that it yields unique AC classes for any interaction,
- in Sec.4.3.4 we present a total rewrite order on interactions that can be used to select unique representatives of AC classes,
- finally, in Sec.4.3.5 we present an implementation of the whole process and demonstrate its use on some examples.

4.3.1 A sound axiom system for interactions

As explained in Sec.2.2, a $\mathcal{F}$-algebra $\mathcal{A}$ induces a congruence relation $=_\mathcal{A}$ on a corresponding term algebra $\mathcal{T}_\mathcal{F}(\mathcal{X})$. This also holds in the case of interactions and their semantics, and, in that case, $=_\mathcal{A}$ refers to semantic equality i.e. for any two interactions i_1 and i_2 we have $(i_1 =_\mathcal{A} i_2) \Leftrightarrow (\sigma_d(i_1) = \sigma_d(i_2))$.

Finding an equational base that can generate $=_\mathcal{A}$ is outside the scope of this thesis. However, using algebraic properties of the operators on sets of traces defined in Sec.4.1, we define in this section a sound axiom system w.r.t. $=_\mathcal{A}$.

Let us consider the axiom system $E_\mathbb{I}$ defined in Def.4.16 on the set of interaction terms with variables. Those 39 equations can be informally summarized as follows, given variables x, y and z:

$$
\begin{array}{lll}
\forall\ f \in \{strict, seq, par\}, & f(\varnothing, y)\ \approx\ y & \text{(Simpl-Left)} \\
\forall\ f \in \{strict, seq, par\}, & f(x, \varnothing)\ \approx\ x & \text{(Simpl-Right)} \\
\forall\ f \in \{strict, seq, par, alt\}, & f(f(x, y), z)\ \approx\ f(x, f(f, z)) & \text{(Flush)} \\
\forall\ f \in \{par, alt\}, & f(x, y)\ \approx\ f(y, x) & \text{(Invert)} \\
 & alt(x, x)\ \approx\ x & \text{(Duplicate)} \\
\forall\ f \in \{strict, seq, par\}, & f(x, alt(y, z))\ \approx\ alt(f(x, y), f(x, z)) & \text{(Factorize-Left)} \\
\forall\ f \in \{strict, seq, par\}, & f(alt(x, y), z)\ \approx\ alt(f(x, z), f(y, z)) & \text{(Factorize-Right)} \\
\forall\ k \in \{S, H, W, P\}, & loop_k(\varnothing)\ \approx\ \varnothing & \text{(Loop-Simpl)} \\
\forall\ (k_1, k_2) \in \{S, H, W, P\}^2, & loop_{k_1}(loop_{k_2}(x))\ \approx\ loop_{min(k_1,k_2)}(x) & \text{(Loop-Unnest)}
\end{array}
$$

We have here gathered the 39 equations from Def.4.16 under 9 named transformations. Those names informally describe the intended effect of the transformation they describe. It is then trivial to show that this axiom system $E_{\mathbb{I}}$ is sound w.r.t. the $\mathcal{F}$-algebra $\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega), \mathcal{F}^{\mathcal{A}})$ which is the semantic domain of sets of traces. Indeed, all those equations correspond to algebraic properties of the operators on $\mathcal{P}(\mathbb{T}_\Omega)$ which are the counterparts $f^{\mathcal{A}}$ of the operation symbols $f \in \mathcal{F}$ that are involved.

Indeed, we have that:

- "Simpl-Left" and "Simpl-Right" respectively describes the simplification by removal of an empty interaction $\varnothing$ on the left and on the right of a scheduling constructor. The 6 equations related to those transformations are sound because $\{\epsilon\}$ is a neutral element for ;, $;_*$ and $||$ as per Lem.4.10, Lem.4.16 and Lem.4.5 and because $\varnothing^{\mathcal{A}} = \{\epsilon\}$

- "Flush" describes transformations related the associativity of the ; (Lem.4.10), $;_*$ (Lem.4.16), $||$ (Lem.4.5), and $\cup$ operators. The 4 equations related to "Flush" are therefore sound.

- "Invert" describes the inversion of the left and right sub-interactions underneath a *par* or *alt* constructor. The 2 equations related to this transformation are sound because they correspond to the commutativity of the $||$ (Lem.4.5) and $\cup$ operators.

- "Duplicate" describes the duplication or simplification of an *alt* whenever its two branches are identical. The equation related to this transformation is sound because it corresponds to the idempotence of the union $\cup$ operator.

- "Factorize-Left" and "Factorize-Right" respectively describes the factorization of a prefix/suffix x whenever two branches of an alternative "start/end" (given a scheduling constructor) with x. The 6 equations related to those transformation are sound because they correspond to the left and right distributivity of the ; (Lem.4.10), $;_*$ (Lem.4.16) and $||$ (Lem.4.5) operators w.r.t. $\cup$.

- "Loop-Simpl" describe the simplification of a loop constructor that specify the repetition of the empty behavior $\varnothing$. The 4 equations related to this transformation are sound because $\{\epsilon\}$ is a neutral element for $^{;*}$, $^{;^{\rightarrow}_*}$, $^{;**}$ and $^{||*}$ as per Lem.4.20.

- "Loop-Unnest" describe the simplification of nested loops whenever they specify the repetition of a repetition, which can be brought back to a simple repetition. Here, $min : \{S, H, W, P\}^2 \to \{S, H, W, P\}$ is the minimum according to the total order $P < W < H < S$. The 16 equations related to this transformation are sound because they correspond to the idempotency properties of the Kleene closures and Head-First closures that we have described in Lem.4.22 and Lem.4.28.

Definition 4.16: An axiom system on interaction terms

Given the signature $\mathcal{F} = \{\epsilon\} \cup \mathbb{A}_\Omega \cup \{strict, seq, par, alt, loop_S, loop_H, loop_W, loop_P\}$ and a set of variables $\mathcal{X}$ which includes $\{x, y, z\}$, let us consider the following axiom system $E_{\mathbb{I}}$ on the set $\mathcal{T}_{\mathcal{F}}(\mathcal{X})$ of interaction terms with variables, which contains the following equations:

$$strict(x, \varnothing) \approx x \qquad seq(x, \varnothing) \approx x \qquad par(x, \varnothing) \approx x$$
$$strict(\varnothing, x) \approx x \qquad seq(\varnothing, x) \approx x \qquad par(\varnothing, x) \approx x$$
$$strict(strict(x, y), z) \approx strict(x, strict(y, z)) \qquad seq(seq(x, y), z) \approx seq(x, seq(y, z))$$
$$par(par(x, y), z) \approx par(x, par(y, z)) \qquad alt(alt(x, y), z) \approx alt(x, alt(y, z))$$
$$par(x, y) \approx par(y, x) \qquad alt(x, y) \approx alt(y, x) \qquad alt(x, x) \approx x$$
$$strict(x, alt(y, z)) \approx alt(strict(x, y), strict(x, z)) \qquad seq(x, alt(y, z)) \approx alt(seq(x, y), seq(x, z))$$
$$par(x, alt(y, z)) \approx alt(par(x, y), par(x, z)) \qquad strict(alt(x, y), z) \approx alt(strict(x, z), strict(y, z))$$
$$seq(alt(x, y), z) \approx alt(seq(x, z), seq(y, z)) \qquad par(alt(x, y), z) \approx alt(par(x, z), par(y, z))$$
$$loop_S(\varnothing) \approx \varnothing \quad loop_H(\varnothing) \approx \varnothing \quad loop_W(\varnothing) \approx \varnothing \quad loop_P(\varnothing) \approx \varnothing$$
$$loop_S(loop_S(x)) \approx loop_S(x) \qquad loop_S(loop_H(x)) \approx loop_H(x)$$
$$loop_S(loop_W(x)) \approx loop_W(x) \qquad loop_S(loop_P(x)) \approx loop_P(x)$$
$$loop_H(loop_S(x)) \approx loop_H(x) \qquad loop_H(loop_H(x)) \approx loop_H(x)$$
$$loop_H(loop_W(x)) \approx loop_W(x) \qquad loop_H(loop_P(x)) \approx loop_P(x)$$
$$loop_W(loop_S(x)) \approx loop_W(x) \qquad loop_W(loop_H(x)) \approx loop_W(x)$$
$$loop_W(loop_W(x)) \approx loop_W(x) \qquad loop_W(loop_P(x)) \approx loop_P(x)$$
$$loop_P(loop_S(x)) \approx loop_P(x) \qquad loop_P(loop_H(x)) \approx loop_P(x)$$
$$loop_P(loop_W(x)) \approx loop_P(x) \qquad loop_P(loop_P(x)) \approx loop_P(x)$$

Lemma 4.29: Soundness of $E_{\mathbb{I}}$

$E_{\mathbb{I}}$ is sound w.r.t. the $\mathcal{F}$-algebra $\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega), \mathcal{F}^{\mathcal{A}})$

Proof. Trivial. □

$E_{\mathbb{I}}$ is a sound axiom system w.r.t. $\mathcal{A}$ but we can however remark that it is not complete. Indeed, we have for instance that $seq(l!m_1, l!m_2)$ and $strict(l!m_1, l!m_2)$ have the same image through any environment $\overline{\rho}$ even though they cannot be related via $\approx_{E_{\mathbb{I}}}$ i.e. we have $seq(l!m_1, l!m_2) =_{\mathcal{A}} strict(l!m_1, l!m_2)$ but we also have that $seq(l!m_1, l!m_2) \not\approx_{E_{\mathbb{I}}} strict(l!m_1, l!m_2)$ and therefore $=_{\mathcal{A}} \not\subseteq \approx_{E_{\mathbb{I}}}$.

Given that the denotational semantics σ_d of interactions is defined as the initial homomorphism between the ground term algebra $\mathbb{I}_\Omega$ and $\mathcal{P}(\mathbb{T}_\Omega)$ we have in particular that, for any two interactions i and i', if $i \approx_{E_{\mathbb{I}}} i'$ then we have that $\sigma_d(i) = \sigma_d(i')$.

Lemma 4.30: $\approx_{E_{\mathbb{I}}}$ preserves σ_d

For any two interactions i and i' we have that:

$$(i \approx_{E_{\mathbb{I}}} i') \Rightarrow (\sigma_d(i) = \sigma_d(i'))$$

Proof. Implied by Lem.4.29. □

We have therefore identified a syntactic relation on interaction terms that relate (some but not all) semantically equivalent interactions. This relation $\approx_{E_{\mathbb{I}}}$ naturally partitions the set of interaction terms into classes $[\,]_{\approx_{E_{\mathbb{I}}}}$ of semantically equivalent terms.

4.3.2 A process to normalize interaction terms

In Sec.2.2, we have seen that Associative-Commutative Rewriting (AC-R) can be used to define terminating rewrite systems on languages in which some operators may be associative or commutative. Given that this is the case for our IL, we use AC-R in this section so as define a process to normalize interaction terms.

Let us consider the following equational theory $T_{\mathbb{I}}$, which gathers all equations from $E_{\mathbb{I}}$ related to the associativity and commutativity of constructors:

$$T_{\mathbb{I}} = \left\{ \begin{array}{cc} strict(strict(x,y),z) \approx strict(x,strict(y,z)) & seq(seq(x,y),z) \approx seq(x,seq(y,z)) \\ par(par(x,y),z) \approx par(x,par(y,z)) & alt(alt(x,y),z) \approx alt(x,alt(y,z)) \\ par(x,y) \approx par(y,x) & alt(x,y) \approx alt(y,x) \end{array} \right\}$$

$T_{\mathbb{I}}$ contains exactly the equations from $E_{\mathbb{I}}$ (defined in Sec.4.3.1) which correspond to the properties of associativity and commutativity of the symbols of the Interaction Language (IL). Indeed we have that *strict*, *seq*, *par* and *alt* are associative and *alt* and *par* are commutative.

By putting those equations from $E_{\mathbb{I}}$ in $T_{\mathbb{I}}$, we remove concerns of non-terminations related to those equations. However, there are still some equations in $E_{\mathbb{I}}$ that are problematic w.r.t. non-termination.

The distributivity of scheduling constructors w.r.t. *alt*

Indeed, in $E_{\mathbb{I}}$, there are 6 equations relating to the distributivity of the scheduling constructors (*strict*, *seq* and *par*) w.r.t. to the *alt* constructor. When simplifying an interaction term, we might need to use those equations in both directions. This is exemplified in Fig.4.10, where, in order to simplify the interaction at the top left into the one at the bottom right, we make use of both directions of the "Factorize-Right" transformation. Indeed we have that:

- the first transformation consists in applying "Factorize-Right" at position 2, in the direction that complexifies the term. Doing so reveals that the sub-interactions at positions 1 and 22 are identical (syntaxically).

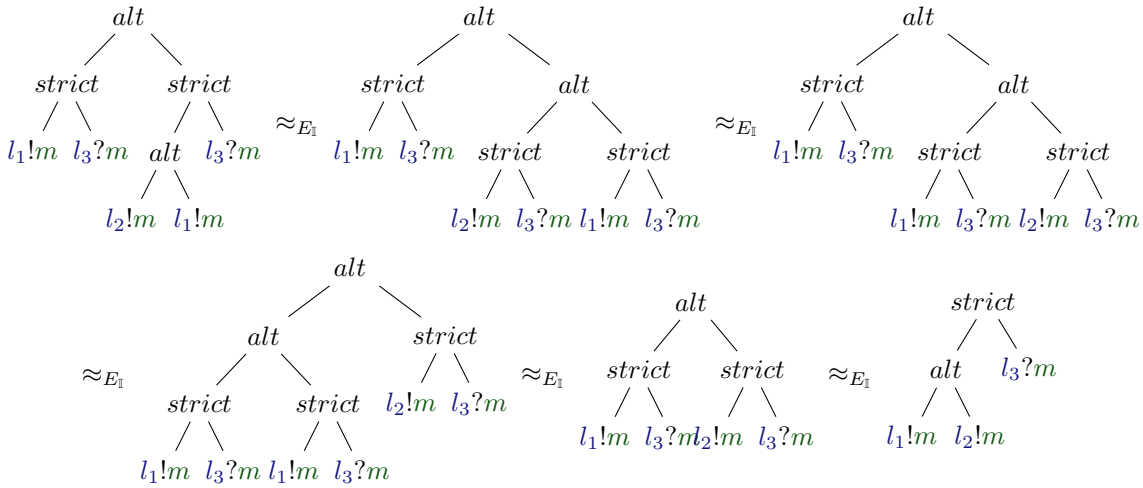


Figure 4.10: Simplification of an example interaction which uses distributivity in both directions

- then, we apply "Invert" at position 2 and then "Flush" at the root position ϵ. Those two successive transformations allow to place the two identical sub-interactions as siblings under an *alt* node.

- this then allows the use of "Duplicate" so as to eliminate a duplicate branch.

- finally, we apply "Factorize-Right" again but this time in the other direction, at the root position ϵ

In the example from Fig.4.10 we therefore use the distributivity of *strict* w.r.t. *alt* in both direction. A first time as $strict(alt(x, y), z) \rightsquigarrow alt(strict(x, z), strict(y, z))$ in order to reveal that there exists two identical branches of the alternative modulo AC. And then, once this duplicated branch has been removed, we use $alt(strict(x, z), strict(y, z)) \rightsquigarrow strict(alt(x, y), z)$. As a result, if one were to use a single rewrite system to normalize interactions in this fashion, we would have both directions of this equation and the rewrite system would be non-terminating.

This remark can be likewise made for all equations of $E_\mathbb{I}$ pertaining to the left and right distributivity of all three scheduling operators w.r.t. the alternative operator.

In order to solve this problem we simply define two distinct rewrite systems, which we then apply one after the other so as to normalize interactions. The first one uses distributivity in the direction that expands the terms and, at the same time, eliminates duplicated branches underneath alternatives. Then, the second rewrite system uses the distributivity in the direction that contracts the terms. In the following we may say that our process to normalize interactions is a two phases process, in which phases 1 and 2 respectively correspond to the application of the first and second rewrite system.

First phase of the process

Let us at first consider the set $R^0_{\mathbb{I}}$ of rewrite rules defined by:

$$R^0_{\mathbb{I}} = \left\{ \begin{array}{cccc} strict(x,\varnothing) \rightsquigarrow x & seq(x,\varnothing) \rightsquigarrow x & par(x,\varnothing) \rightsquigarrow x & \\ strict(\varnothing,x) \rightsquigarrow x & seq(\varnothing,x) \rightsquigarrow x & par(\varnothing,x) \rightsquigarrow x & \\ loop_S(\varnothing) \rightsquigarrow \varnothing & loop_H(\varnothing) \rightsquigarrow \varnothing & loop_W(\varnothing) \rightsquigarrow \varnothing & loop_P(\varnothing) \rightsquigarrow \varnothing \\ loop_S(loop_S(x)) \rightsquigarrow loop_S(x) & loop_S(loop_H(x)) \rightsquigarrow loop_H(x) & & \\ loop_S(loop_W(x)) \rightsquigarrow loop_W(x) & loop_S(loop_P(x)) \rightsquigarrow loop_P(x) & & \\ loop_H(loop_S(x)) \rightsquigarrow loop_H(x) & loop_H(loop_H(x)) \rightsquigarrow loop_H(x) & & \\ loop_H(loop_W(x)) \rightsquigarrow loop_W(x) & loop_H(loop_P(x)) \rightsquigarrow loop_P(x) & & \\ loop_W(loop_S(x)) \rightsquigarrow loop_W(x) & loop_W(loop_H(x)) \rightsquigarrow loop_W(x) & & \\ loop_W(loop_W(x)) \rightsquigarrow loop_W(x) & loop_W(loop_P(x)) \rightsquigarrow loop_P(x) & & \\ loop_P(loop_S(x)) \rightsquigarrow loop_P(x) & loop_P(loop_H(x)) \rightsquigarrow loop_P(x) & & \\ loop_P(loop_W(x)) \rightsquigarrow loop_P(x) & loop_P(loop_P(x)) \rightsquigarrow loop_P(x) & & \end{array} \right\}$$

This set $R^0_{\mathbb{I}}$ contains all the rules that are straightforward and do not pose any problem regarding non-termination. $R^0_{\mathbb{I}}$ will serve as a baseline rule set to define both rewrite systems.

We then define the set $R^1_{\mathbb{I}}$ of rewrite rules as follows:

$$R^1_{\mathbb{I}} = R^0_{\mathbb{I}} \cup \left\{ \begin{array}{c} alt(x,x) \rightsquigarrow x \\ strict(x, alt(y,z)) \rightsquigarrow alt(strict(x,y), strict(x,z)) \\ strict(alt(x,y), z) \rightsquigarrow alt(strict(x,z), strict(y,z)) \\ seq(x, alt(y,z)) \rightsquigarrow alt(seq(x,y), seq(x,z)) \\ seq(alt(x,y), z) \rightsquigarrow alt(seq(x,z), seq(y,z)) \\ par(x, alt(y,z)) \rightsquigarrow alt(par(x,y), par(x,z)) \\ par(alt(x,y), z) \rightsquigarrow alt(par(x,z), par(y,z)) \end{array} \right\}$$

The set $R^1_{\mathbb{I}}$ contains additional rules which are exclusive to the first phase of the process. Those rules contains:

- the elimination of duplicated branches underneath alternatives with $alt(x,x) \rightsquigarrow x$. Given that we use rewriting modulo AC, and given that the *alt* operator is AC, we only need a single rule to do so. Indeed, we have that for instance $alt(x, alt(y,x))$ or $alt(alt(x,y),x)$ and so on can all be simplified into $alt(x,y)$ modulo AC.
- and the use of the distributive properties of *strict*, *seq* and *par* w.r.t. *alt* in the direction that expands the terms with $f(x, alt(y,z)) \rightsquigarrow alt(f(x,y), f(x,z))$ for the left distributivity and $f(alt(x,y),z) \rightsquigarrow alt(f(x,z), f(y,z))$ for the right distributivity of any $f \in \{strict, seq, par\}$

We can then define phase 1 as the normalisation associated with rewrite relation $\rightarrow_{R^1_{\mathbb{I}}/T_{\mathbb{I}}}$, which we prove to

be convergent in Sec.4.3.3.

Second phase of the process

Likewise, we can define the second phase of the process in a similar fashion. We complement the set $R^0_{\mathbb{I}}$ of baseline rules into a set $R^2_{\mathbb{I}}$ which includes additional rules which exploit the distributivity of scheduling operators w.r.t. *alt* in the direction that contracts terms. However, as illustrated by the example from Fig.4.11 it does not suffice to include the rule $alt(strict(x,y), strict(x,z)) \rightsquigarrow strict(x, alt(y,z))$. Indeed with only that rule we cannot simplify the example from from Fig.4.11 without relying on using $x \rightsquigarrow strict(x, \varnothing)$.

Indeed, on Fig.4.11 we use at first $x \rightsquigarrow strict(x, \varnothing)$ ("Simpl-Right") at position 1 before being able to use the distributivity ("Factorize-Left") of *strict*. However, adding the rule $x \rightsquigarrow strict(x, \varnothing)$ is not compatible with having a terminating rewrite system.

To be able to operate the simplification from Fig.4.11 we need the additional rule $alt(x, strict(x,y)) \rightsquigarrow strict(x, alt(\varnothing, y))$. Then given that we operate rewriting modulo AC we do not need any additional rules. For instance it is no use looking for x at a deeper level that $strict(x,y)$ on the left-hand side because, if we have $strict(strict(x,y),z)$ we can always equate it to $strict(x, strict(y,z))$ thanks to the equational theory T (i.e. using AC-R).

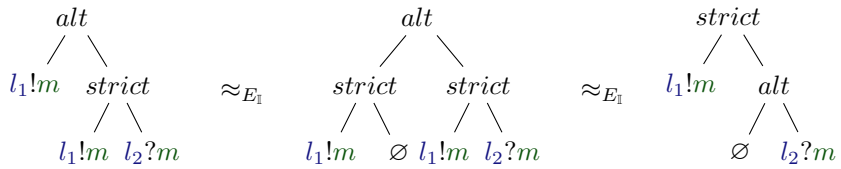


Figure 4.11: Simplification using distributivity in a particular case that is not covered by a single equation

This reasoning can be applied to all three scheduling operator and also to their properties of right-distributivity. As a result, we may define the set $R^2_{\mathbb{I}}$ of rewrite rules as follows:

$$R^2_{\mathbb{I}} = R^0_{\mathbb{I}} \cup \left\{ \begin{array}{c} alt(x, strict(x,y)) \rightsquigarrow strict(x, alt(\varnothing, y)) \\ alt(strict(x,y), strict(x,z)) \rightsquigarrow strict(x, alt(y,z)) \\ alt(strict(x,y), y) \rightsquigarrow strict(alt(x,\varnothing), y) \\ alt(strict(x,z), strict(y,z)) \rightsquigarrow strict(alt(x,y), z) \\ alt(x, seq(x,y)) \rightsquigarrow seq(x, alt(\varnothing, y)) \\ alt(seq(x,y), seq(x,z)) \rightsquigarrow seq(x, alt(y,z)) \\ alt(seq(x,y), y) \rightsquigarrow seq(alt(x,\varnothing), y) \\ alt(seq(x,z), seq(y,z)) \rightsquigarrow seq(alt(x,y), z) \\ alt(x, par(x,y)) \rightsquigarrow par(x, alt(\varnothing, y)) \\ alt(par(x,y), par(x,z)) \rightsquigarrow par(x, alt(y,z)) \\ alt(par(x,y), y) \rightsquigarrow par(alt(x,\varnothing), y) \\ alt(par(x,z), par(y,z)) \rightsquigarrow par(alt(x,y), z) \end{array} \right\}$$

With the definition of $R^2_{\mathbb{I}}$ we can then define phase 2 as the process of normalisation associated to the $\rightarrow_{R^2_{\mathbb{I}}/T_{\mathbb{I}}}$ rewrite relation.

We then define our process to normalize interactions as the successive application of the two rewrite systems which are $R^1_{\mathbb{I}}/T_{\mathbb{I}}$ and then $R^2_{\mathbb{I}}/T_{\mathbb{I}}$. In Sec.4.3.3 we provide automated proofs for the convergence (i.e. termination and confluence) of both rewrite systems. Given that those rewrite systems are convergent they define unique normal forms. For any interaction i let us denote by $R^1(i)$, and $R^2(i)$ the normal forms which are respectively associated to the application of $R^1_{\mathbb{I}}/T_{\mathbb{I}}$ and $R^2_{\mathbb{I}}/T_{\mathbb{I}}$. We then consider, for any interaction i, the normal form $R^2(R^1(i))$ which is uniquely defined (by composition). However, given that we are here considering rewriting modulo theories, and in particular modulo AC, $R^2(R^1(i))$ does not designates an interaction term but a class of equivalent interaction terms modulo AC.

Indeed, the process which we have formalized is a process modulo AC. As a result it yields classes of equivalence modulo AC and not normal forms of interactions. To actually implement an algorithm for computing a normal form, we need to impose a total rewrite ordering on terms. We define such an ordering in Sec.4.3.4.

Before introducing the rewrite ordering, let us, in the next section, prove the convergence of $R^1_{\mathbb{I}}/T_{\mathbb{I}}$ and $R^2_{\mathbb{I}}/T_{\mathbb{I}}$. As previously mentioned, those proofs consist in automated proofs of termination and confluence obtained from the use of dedicated software.

4.3.3 Automated proofs of convergence

There exist many software tools for automating proofs of termination or confluence for rewriting systems. We can for instance cite CiME3 [44] (termination and confluence), TTT2 [76] (termination), NaTT [126] (termination), CSI [127] (confluence), CoLL [121] (confluence) etc. Those tools may use various methods, which we will not address, to prove those results.

A standard format called the WST format (format for the international WorkShop on Termination [1]) enables the encoding of Class Rewriting Systems for use in various tools such as the ones that we have mentioned.

On Fig.4.12 we encode in WST format the first phase of our process i.e. the CRS $R^1_{\mathbb{I}}/T_{\mathbb{I}}$. Likewise, on Fig.4.13 we encode $R^2_{\mathbb{I}}/T_{\mathbb{I}}$.

Then, using TTT2 (version 1.20) and CSI (version 1.2.4) we respectively proved the termination and the confluence of both CRSs.

[1] `http://termination-portal.org/wiki/WST`

```
(VAR x y z)
(THEORY (AC alt) (AC par) (A strict) (A seq))
(RULES
  strict(x,o) → x
  seq(x,o)    → x
  par(x,o)    → x

  strict(o,x) → x
  seq(o,x)    → x
  par(o,x)    → x

  loopS(o) → o
  loopH(o) → o
  loopW(o) → o
  loopP(o) → o

  loopS(loopS(x)) → loopS(x)
  loopS(loopH(x)) → loopH(x)
  loopS(loopW(x)) → loopW(x)
  loopS(loopP(x)) → loopP(x)

  loopH(loopS(x)) → loopH(x)
  loopH(loopH(x)) → loopH(x)
  loopH(loopW(x)) → loopW(x)
  loopH(loopP(x)) → loopP(x)

  loopW(loopS(x)) → loopW(x)
  loopW(loopH(x)) → loopW(x)
  loopW(loopW(x)) → loopW(x)
  loopW(loopP(x)) → loopP(x)

  loopP(loopS(x)) → loopP(x)
  loopP(loopH(x)) → loopP(x)
  loopP(loopW(x)) → loopP(x)
  loopP(loopP(x)) → loopP(x)

  alt(x,x) → x

  strict(x,alt(y,z)) → alt(strict(x,y),strict(x,z))
  seq(x,alt(y,z))    → alt(seq(x,y),seq(x,z))
  par(x,alt(y,z))    → alt(par(x,y),par(x,z))

  strict(alt(x,y),z) → alt(strict(x,z),strict(y,z))
  seq(alt(x,y),z)    → alt(seq(x,z),seq(y,z))
  par(alt(x,y),z)    → alt(par(x,z),par(y,z))
)
```

Figure 4.12: Phase 1/2 for normalizing interactions

```
(VAR x y z)
(THEORY (AC alt) (AC par) (A strict) (A seq))
(RULES
  strict(x,o) → x
  seq(x,o)    → x
  par(x,o)    → x

  strict(o,x) → x
  seq(o,x)    → x
  par(o,x)    → x

  loopS(o) → o
  loopH(o) → o
  loopW(o) → o
  loopP(o) → o

  loopS(loopS(x)) → loopS(x)
  loopS(loopH(x)) → loopH(x)
  loopS(loopW(x)) → loopW(x)
  loopS(loopP(x)) → loopP(x)

  loopH(loopS(x)) → loopH(x)
  loopH(loopH(x)) → loopH(x)
  loopH(loopW(x)) → loopW(x)
  loopH(loopP(x)) → loopP(x)

  loopW(loopS(x)) → loopW(x)
  loopW(loopH(x)) → loopW(x)
  loopW(loopW(x)) → loopW(x)
  loopW(loopP(x)) → loopP(x)

  loopP(loopS(x)) → loopP(x)
  loopP(loopH(x)) → loopP(x)
  loopP(loopW(x)) → loopP(x)
  loopP(loopP(x)) → loopP(x)

  alt(x,strict(x,y))           → strict(x,alt(o,y))
  alt(strict(x,y),strict(x,z)) → strict(x,alt(y,z))
  alt(strict(x,y),y)           → strict(alt(x,o),y)
  alt(strict(x,z),strict(y,z)) → strict(alt(x,y),z)

  alt(x,seq(x,y))         → seq(x,alt(o,y))
  alt(seq(x,y),seq(x,z)) → seq(x,alt(y,z))
  alt(seq(x,y),y)         → seq(alt(x,o),y)
  alt(seq(x,z),seq(y,z)) → seq(alt(x,y),z)

  alt(x,par(x,y))         → par(x,alt(o,y))
  alt(par(x,y),par(x,z)) → par(x,alt(y,z))
  alt(par(x,y),y)         → par(alt(x,o),y)
  alt(par(x,z),par(y,z)) → par(alt(x,y),z)
)
```

Figure 4.13: Phase 2/2 for normalizing interactions

4.3.4 A total rewrite ordering on interactions

As mentioned previously, in order to be able to compute normal forms of interactions (as uniquely defined terms), we need to specify a total rewrite ordering on ground interaction terms. The purpose of this section is to define this order.

To do so, let us at first define a total order on the set of atomic actions $\mathbb{A}_\Omega$. We define such an order $\prec_{\mathbb{A}}$ in Def.4.17.

Definition 4.17: A total order on actions

Given a signature $\Omega = (L, M)$, let us consider the total orders $\prec_L$ and $\prec_M$ on the (finite) sets of lifelines L and messages M. Let us also denote by $\prec_\Delta$ the total order on $\{!, ?\}$ such that $! \prec_\Delta ?$.
We can then define a total order $\prec_{\mathbb{A}}$ on the set of actions $\mathbb{A}_\Omega$ such that for any two actions $a_1 = l_1 \Delta_1 m_1$ and $a_2 = l_2 \Delta_2 m_2$ with $l_1, l_2 \in L$, $\Delta_1, \Delta_2 \in \{!, ?\}$ and $m_1, m_2 \in M$ we have $a_1 \prec_{\mathbb{A}} a_2$ iff:

- either $l_1 \prec_L l_2$
- or $l_1 = l_2$ and $\Delta_1 \prec_\Delta \Delta_2$
- or $l_1 = l_2$ and $\Delta_1 = \Delta_2$ and $m_1 \prec_M m_2$

We then define in Def.4.18 the total orders $\prec_{\mathcal{F}_0}$, $\prec_{\mathcal{F}_1}$ and $\prec_{\mathcal{F}_2}$ on the symbols of our Interaction Language.

Definition 4.18: Total orders on operation symbols

Given $\mathcal{F}_0 = \{\varnothing\} \cup \mathbb{A}_\Omega$, we define the total order $\prec_{\mathcal{F}_0}$ such that:

$$\prec_{\mathcal{F}_0} = \prec_{\mathbb{A}} \cup \{(\varnothing, a) \mid a \in \mathbb{A}_\Omega\}$$

Given $\mathcal{F}_1 = \{strict, seq, par, alt\}$, we define the total order $\prec_{\mathcal{F}_1}$ such that:

$$par \prec_{\mathcal{F}_1} seq \prec_{\mathcal{F}_1} strict \prec_{\mathcal{F}_1} alt$$

Given $\mathcal{F}_2 = \{loop_S, loop_H, loop_W, loop_P\}$, we define the total order $\prec_{\mathcal{F}_2}$ such that:

$$loop_P \prec_{\mathcal{F}_2} loop_W \prec_{\mathcal{F}_2} loop_H \prec_{\mathcal{F}_2} loop_S$$

This finally allow us to define the total order $\prec_{\mathbb{I}}$ on interaction terms in Def.4.19.

Definition 4.19: A total order on ground interaction terms

We can then define a total order $\prec_{\mathbb{I}}$ on $\mathbb{I}_\Omega$ s.t. for any two interactions i_α and i_β we have $i_\alpha \prec_{\mathbb{I}} i_\beta$ iff:

- $i_\alpha(\epsilon) \in \mathcal{F}_{k_\alpha}$ and $i_\beta(\epsilon) \in \mathcal{F}_{k_\beta}$ with $k_\alpha < k_\beta$ i.e. at the root node we have operation symbols f_α and f_β s.t. the arity of f_α is smaller than that of f_β
- or the root nodes are symbols of the same arity k and then $i_\alpha \prec_{\mathbb{I}} i_\beta$ iff:
  - either $i_\alpha \prec_{\mathcal{F}_k} i_\beta$
  - or f_α and f_β are the same symbol f of arity n and, given $i_\alpha = f(i_{\alpha|1}, \cdots, i_{\alpha|n})$ and $i_\beta = f(i_{\beta|1}, \cdots, i_{\beta|n})$, we have that $\exists j \in [1, n]$ s.t. $\forall j' < j$, $i_{\alpha|j'} = i_{\beta|j'}$ and $i_{\alpha|j} \prec_{\mathbb{I}} i_{\beta|j}$

The total order on interaction terms defined in Def.4.19 is a rewrite ordering as per Lem.4.31.

Lemma 4.31: $\prec_{\mathbb{I}}$ is a total rewrite ordering on $\mathbb{I}_\Omega$

We have that $\prec_{\mathbb{I}}$ is a total rewrite ordering on $\mathbb{I}_\Omega$

Proof. $\prec_{\mathbb{I}}$ is antireflexive, transitive, antisymmetric, $\mathcal{F}$-compatible and stable under substitution. □

We can then define the normal form of any interaction term i as the unique interaction $R(i)$ which is irreducible by the Ordered Rewriting System (ORS) $\rightarrow_{T\succ_{\mathbb{I}}}$ and such that for any $i' \in R^2(R^1(i))$ we have $i' \rightarrow^*_{T\succ_{\mathbb{I}}} R(i)$ i.e. $R_\prec(i') = i$. We may then abusively use the notation $R(i) = R_\prec(R^2(R^1(i)))$.

4.3.5 Implementation & examples

We have implemented the process to normalize interaction terms using the previously defined mechanisms of ordered rewriting and class rewriting. In this section we illustrate the application of this process on some examples. In particular, we demonstrate the successful automation of the initially human-guided simplification of interaction terms which we have presented on Fig.4.10 and Fig.4.11.

On Fig.4.14 we illustrate the application of the normalisation process on two examples. We can note that the example on the left corresponds to that of Fig.4.11.

The initial interaction term i is represented at the top in each case. Then, successive applications of some atomic transformations, indicated by arrows, transform this initial term from top to bottom. The first phase of the process is illustrated in the area that is colored in blue. The term at the bottom of the area in blue (in each case) is the normal form $R_\prec(R^1(i))$ of the initial interaction i. This term is then passed on to the second phase, which application is represented in the area in green. Finally, the term at the bottom is the normal form $R(i) = R_\prec(R^2(R^1(i)))$.

On Fig.4.15 we provide two more examples. We can remark that the one on the left is that from Fig.4.10. Let us also remark that, on Fig.4.15, we have displayed all transformation sequences found by the implementation to transform the initial interaction into its normal form.

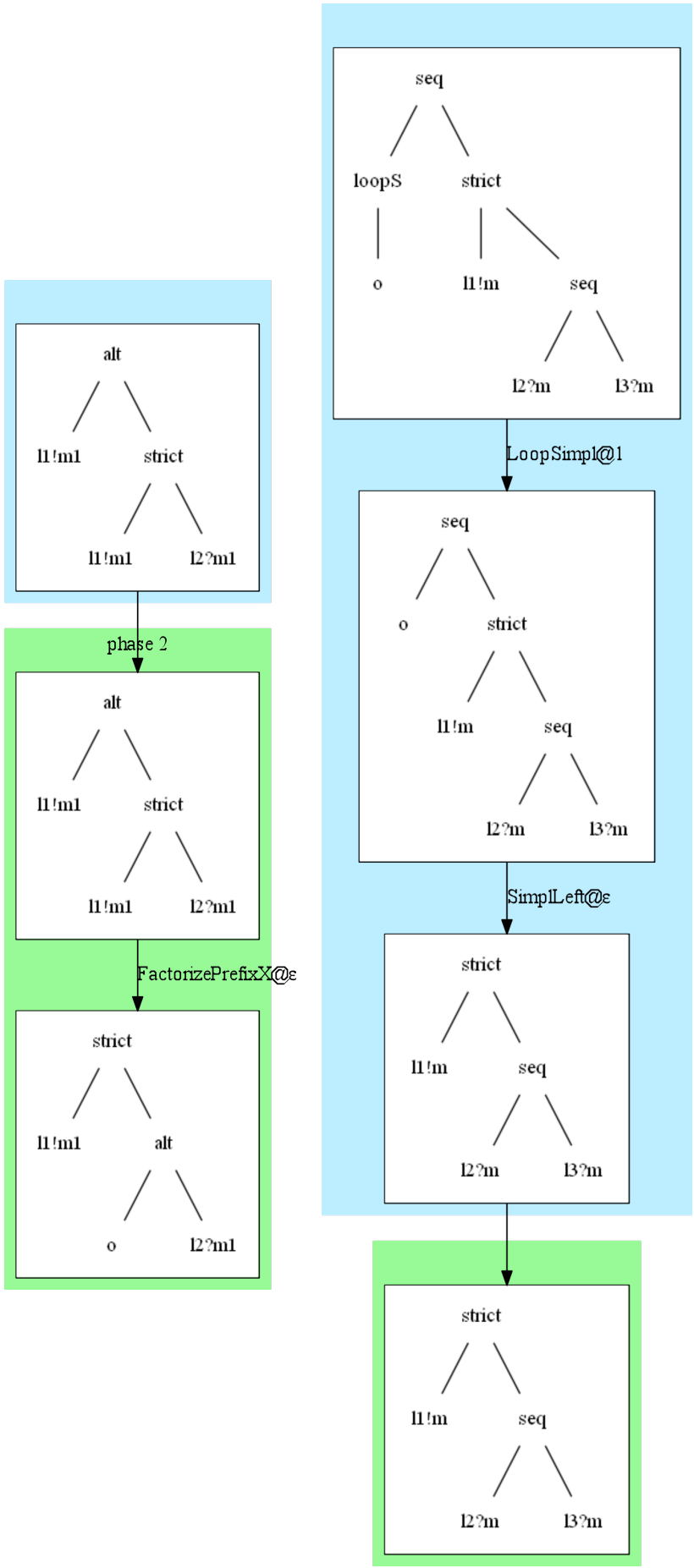


Figure 4.14: Applying our process to normalize interactions on some examples (1/2)

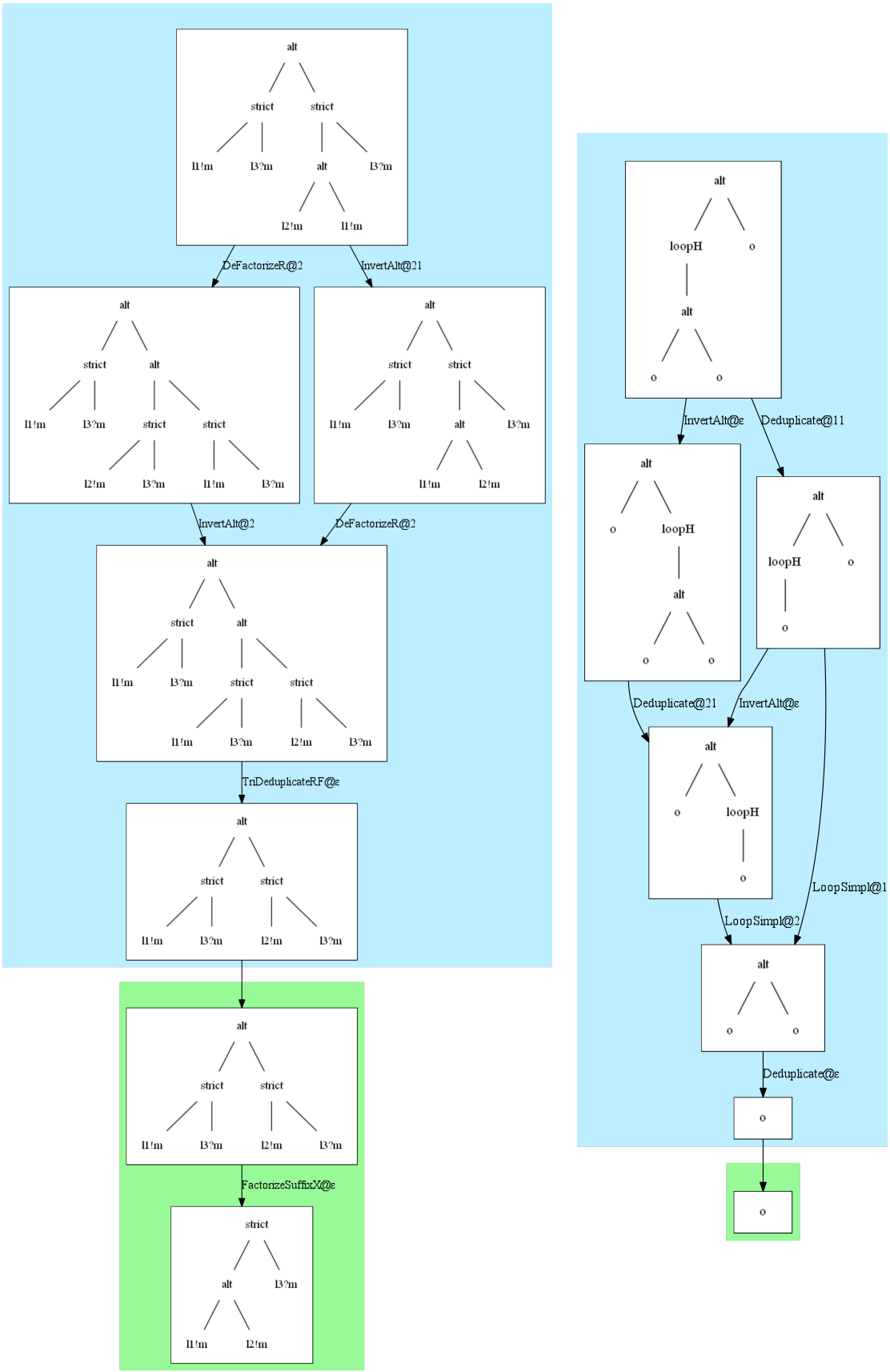


Figure 4.15: Normalizing example interactions (2/2) and displaying all transformation sequences

The process displayed on Fig.4.14 and Fig.4.15 is implemented in the HIBOU tool (those images are generated by HIBOU) which we present in Chap.12. It is possible to explore all paths from the initial interaction to its normal form (as on Fig.4.15) or only one (as on Fig.4.14). Given that the finite rewrite systems have been proven convergent, there is a finite number of such paths and all paths are finite and lead to the same end result.

Conclusion

In this chapter we have defined our Interaction Language (IL). This language takes the form of an algebra of terms that are build inductively from atomic actions which specify the occurrence of communication events and constructors which organize and schedule the occurrences of those events w.r.t. one another. Constructors of our IL allow the specification of strict sequencing so as to strictly order the occurrence of events, weak sequencing so as to order events occurring on the same sub-systems, interleaving so as to allow events to occur in any order, choice between the occurrence of events as well as four different kinds of repetitions. This algebra of terms can be associated to an algebra of sets of traces. A set of traces corresponds to a set of behaviors that may be expressed. Each interaction term can be associated to a correspond (possibly infinite) set of traces which constitutes its semantics. We can then characterize interaction terms which are syntaxically distinct but have the same semantics as members of a common class of equivalence. A specific and unique member of each such class, called the normal form can then be defined and computed using term rewriting.

In the next chapter we define another semantics of interactions in the style of operational semantics that can be found in the field of process calculus.

Chapter 5

A small-step operational semantics

Contents

In this chapter we define another semantics for our Interaction Language (IL). This new semantics, that we denote by σ_o is a small-step operational-style in contrast to σ_d which can be described as being denotational-style. This new semantics is presented in the fashion of structural operational semantics that can be found in the field of process calculus.

The plan of this chapter is as follows:

- in Sec.5.1, we introduce our small-step operational semantics σ_o,
- in Sec.5.2 we prove (with the support of a Coq proof in [88]) that both semantics are equivalent i.e that for any interaction i, the sets of traces $\sigma_d(i)$ and $\sigma_o(i)$ obtained using both methods are equal.

Let us remark that the content of this chapter, formalization and proof have been encoded in Coq in [88]. Results related to this chapter can be found in [95].

5.1 Definition of the Operational Semantics

The small-step operational semantics σ_o consists in reconstructing accepted traces of an original interaction i_0 via the dynamic execution of i_0. Fig.5.1 illustrates this principle. It describes the construction of a trace $t = a_1.\cdots.a_n$ that belongs to the semantics $\sigma_o(i_0)$ of the interaction i_0. This trace can be obtained (as all traces of the semantics) by the concatenation of individual actions a_k that can succeed each-other during the execution of the initial interaction term i_0.

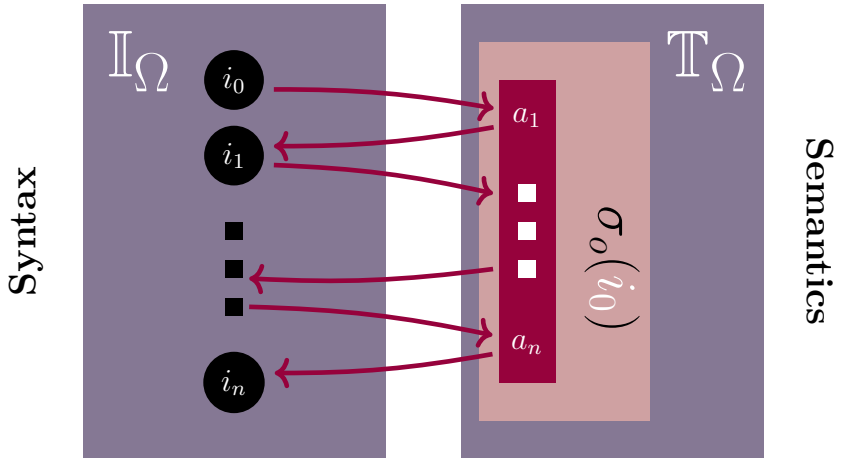


Figure 5.1: Principles of small-step operational semantics applied to interactions

The process described on Fig.5.1 to reconstruct this trace can be explained as follows:

- At first, action a_1 can be immediately executed in i_0. Doing so leads to a "follow-up" interaction i_1 which semantics is that of all traces of $\sigma_o(i_0)$ which start with the specific action a_1 (that exists at a given position within the term structure of interaction i_0). We note this $i_0 \xrightarrow{a_1} i_1$.
- In the same manner, there is a certain action a_2 that can be executed in i_1, which leads to another interaction i_2 (noted $i_1 \xrightarrow{a_2} i_2$) and so on.

- As a result, we obtain a path $i_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} i_n$ such that i_n can express the empty trace ϵ (which can be asserted statically).

- Since i_n expresses the empty trace ϵ, this guarantees that the trace $t = a_1.\cdots.a_n$ (in dark red on Fig.5.1) obtained by concatenation is an accepted trace of the initial interaction i_0. Of course there may exist many such paths, each corresponding to an accepted trace. The overall semantics of the original interaction i_0 is then the set regrouping all of those traces (the area in light red in Fig.5.1).

With the operational semantics, instead of considering all the traces that an interaction may express as a set, we rewrite the interaction on demand, so as to express certain actions. In particular, we may unfold a given loop on demand, when executing an action that is found within its sub-interaction. This formulation of the semantics is much more practical for later use in multi-trace analysis, as we will see in the second part of the thesis, beginning with Chap.8.

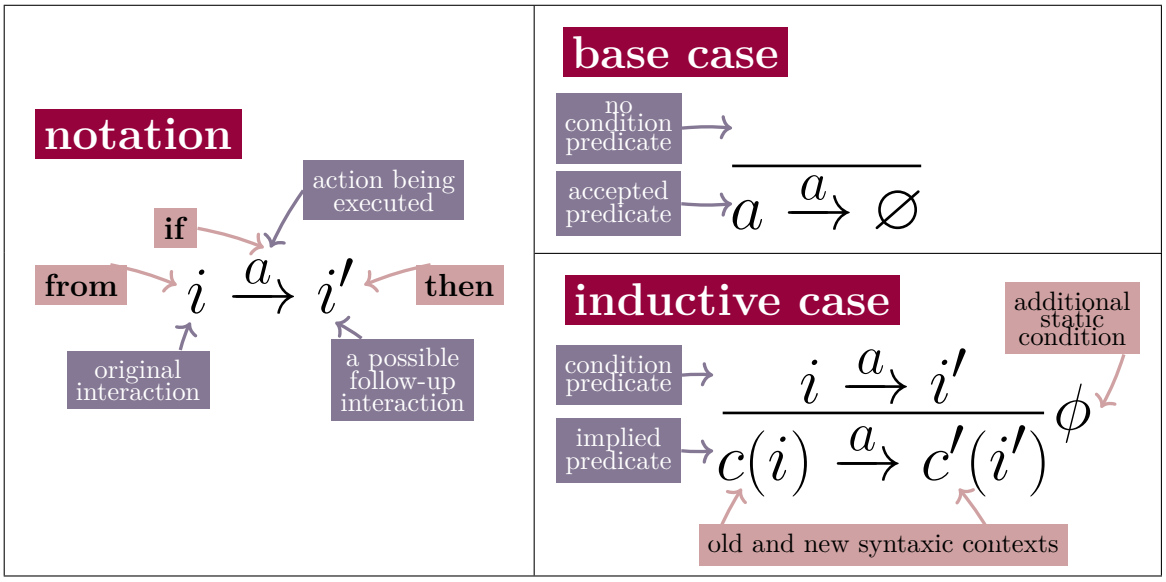


Figure 5.2: Formulation of the operational semantics "à la" process algebra

So as to formally define (i.e. identify as valid) transitions of the form $i \xrightarrow{a} i'$, we define a set of rules in the manner of a process algebra (that we have introduced in Chap.2). The principle of this rule-based approach is described on Fig.5.2:

- Some basic transitions are simply admitted to be true (like axioms in a theory). Those basic transitions corresponds to the executions of interactions that simply consists in individual actions $a \in \mathbb{A}_\Omega$. The single action that forms the interaction can of course be immediately executed; and only the empty interaction $\varnothing$ "remains" to be executed i.e. we have $a \xrightarrow{a} \varnothing$. This base case is represented in the top right cell of Fig.5.2.

- Any other (more complex) transition $i \xrightarrow{a} i'$ must be proved to be true using this base case and a set of rules that deals with constructors (*strict*, *seq*, *par*, *alt*, etc.) in an inductive manner. Those rules take the general form that is represented in the bottom right cell of Fig.5.2. If a certain transition

$i \xrightarrow{a} i'$ is valid (as well as some optional additional conditions on expressed by "ϕ" on Fig.5.2), then a more complex transition $c(i) \xrightarrow{a} c'(i')$ is valid; where c is a syntactic context in which we find i (for instance within $strict(i, i_2)$, $c = strict(\_, i_2)$ is the syntactic context that applies on i) and c' is the new syntactic context in which we include i' so as to form the follow-up interaction $c'(i')$ after the execution of a in $c(i)$.

5.1.1 Executing atomic actions

As we have seen earlier, the execution of atomic actions constitutes the base case for the operational semantics. The corresponding rule is the following:

For any action $a \in \mathbb{A}_\Omega$: $\dfrac{}{a \xrightarrow{a} \varnothing}$

Fig.5.3 illustrates the base rule (which enables the execution of interactions that consists in single actions) and its application on an emission (example 1, on the left) and a reception (example 2, on the right).

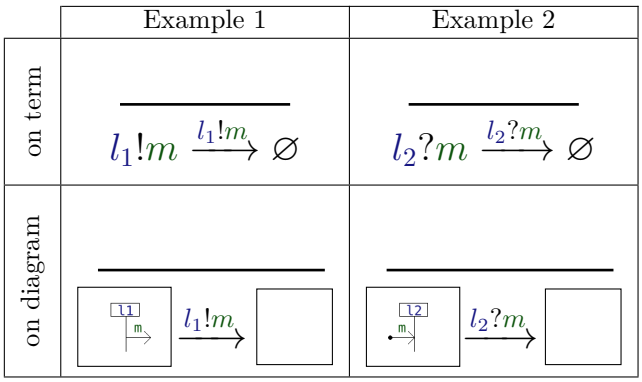


Figure 5.3: Examples for the execution of atomic actions

5.1.2 Executing actions on the left of *strict* and *seq* constructors

In the graphical representation of sequence diagrams, the top to bottom direction corresponds, by default, to the passing of time. Actions that are drawn below some other actions occur after them. In the term structure of a corresponding interaction, this top-to-bottom direction corresponds to a left-to-right relationship. Indeed, within the tree-structure of an interaction, actions are found on leaf nodes, and an action a_1 that is drawn above some other action a_2 in the sequence diagram correspond to a leaf node that is on the left of that of a_2. By "the left" it is meant that a_1 is a left cousin of a_2 w.r.t. some common ancestor which is a binary constructor.

This remark is meant for the default interpretation of the top-to-bottom direction i.e. w.r.t. the weak sequencing "*seq*" constructor. However it also holds for the strict sequencing "*strict*" constructor. Actions found on the left of a *strict* must occur before those found on its right.

Also, whenever an action a is executable within the left sub-interaction i_1 of an interaction $i = f(i_1, i_2)$ with $f \in \{strict, seq\}$, then it is also executable within i. As a result, we have the following two rules:

For any action $a \in \mathbb{A}_\Omega$: $\dfrac{i_1 \xrightarrow{a} i_1'}{strict(i_1, i_2) \xrightarrow{a} strict(i_1', i_2)}$ and $\dfrac{i_1 \xrightarrow{a} i_1'}{seq(i_1, i_2) \xrightarrow{a} seq(i_1', i_2)}$

Fig.5.4 illustrates this in the case of the *strict* operator. As explained earlier, the *strict* operator can be used to express the passing of a message m from a lifeline l_1 to another lifeline l_2, via $strict(l_1!m, l_2?m)$, which is the example used in Fig.5.4. Given that the execution of the emission $l_1!m$, when considered to be an interaction in its own right, is possible, that of $l_1!m$ within $strict(l_1!m, l_2?m)$ is authorized (by induction). In this context, the execution of $l_1!m$ leads to $strict(\varnothing, l_2?m)$, as illustrated on Fig.5.4.

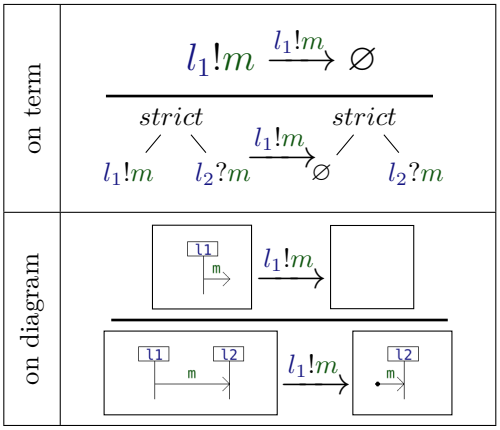


Figure 5.4: Example for executing an action on the left of a *strict* constructors

We illustrate the second rule, that for executing actions on the left of *seq* constructors on Fig.5.5. The example highlights the inductive nature of our rule-based operational semantics. Indeed, here the condition predicate is not an application of the base rule, but of the rule which we obtained in the last example from Fig.5.4.

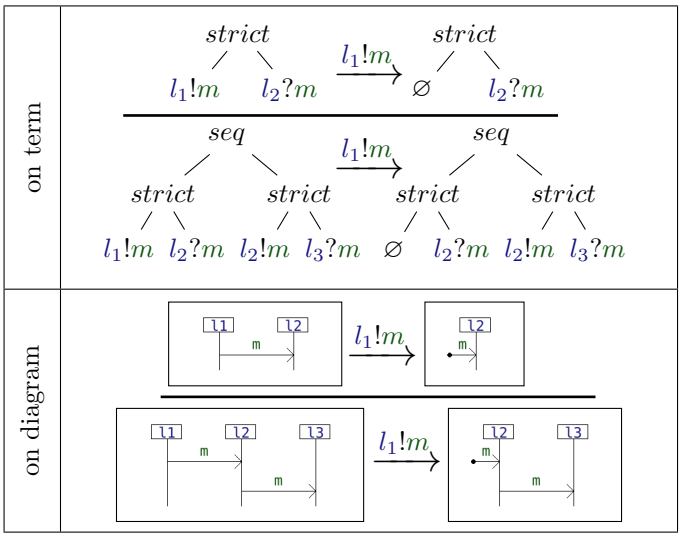


Figure 5.5: Example for executing an action on the left of a *seq* constructors

5.1.3 Executing actions on either side of a *par* constructor

The *par* constructor specifies a concurrent execution of two distinct behaviors, each modelled by a sub-interaction. Those executions being done in parallel, there is no difference in how they should be treated. As a result, there is a symmetry between the rule that specifies the execution of actions on the left of a *par* operator, and the rule that specifies the execution of actions on its right.

Given an interaction $i = par(i_1, i_2)$, if an action a is executable in any of the two sub-interactions, then it is executable in i. As a result, we have the following two rules:

For any action $a \in \mathbb{A}_\Omega$: $\dfrac{i_1 \xrightarrow{a} i_1'}{par(i_1, i_2) \xrightarrow{a} par(i_1', i_2)}$ and $\dfrac{i_2 \xrightarrow{a} i_2'}{par(i_1, i_2) \xrightarrow{a} par(i_1, i_2')}$

Fig.5.6 illustrates the application of the second rule, which is that of executing an action on the right of a *par* constructor, using a simple example.

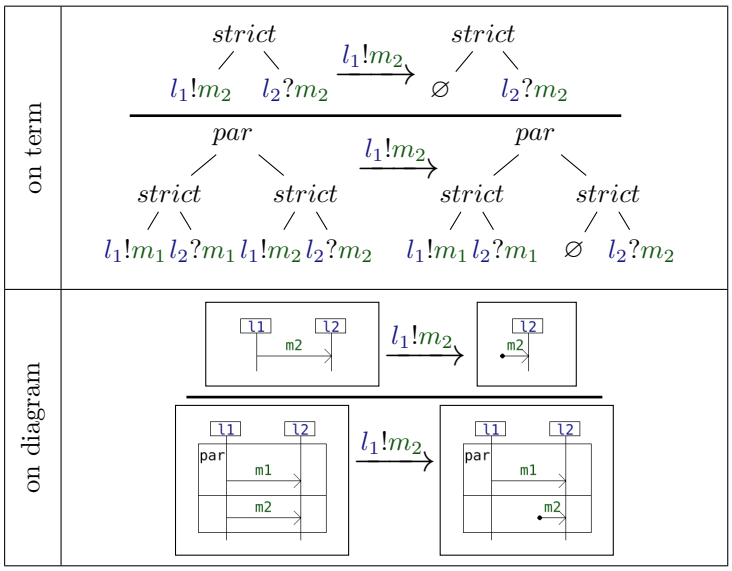


Figure 5.6: Example for executing an action on the right of a *par* constructor

In the example from Fig.5.6, the interaction models a parallel composition of two behaviors:

- a behavior describing the passing of a message m_1 from lifeline l_1 to lifeline l_2
- and another which describes the passing of another message m_2 from lifeline l_1 to lifeline l_2

In the second behavior (at the right in the term and on the bottom in the diagram), which execution is illustrated on Fig.5.6, m_2 can be emitted by l_1 so that only its reception $l_2?m_2$ by l_2 remains to be executed. The "right" rule for the *par* constructor can therefore be applied (which is shown on Fig.5.6). It implies that $l_1!m_2$ can be executed and that what remains to be executed is the parallelization of the original left sub-interaction on the left, and of $l_2?m_2$ on the right (which remains to be executed w.r.t. the right sub-behavior).

In the first behavior (at the left in the term and at the top in the diagram), m_1 can be emitted by l_1 so that only its reception $l_2?m_1$ by l_2 remains to be executed. Here, the "left" rule for the *par* operator could

be applied, which would likewise imply that $l_1!m_1$ can be executed and that what remains to be executed would be the parallelization of the original right sub-interaction on the right, and of $l_2?m_1$ on the left.

5.1.4 Executing actions on either side of a *alt* constructor

The *alt* constructor specifies an exclusive non-deterministic alternative between two sub-behaviors, each modelled by a sub-interaction. Similarly to the *par* constructor, the *alt* treats both its sub-interactions in the same "symmetric" manner. Given an interaction $i = alt(i_1, i_2)$, if an action a is executable in any of the sub-interactions, then it is executable in i.

Executing an action from one of the two sub-interaction means that it has been selected, at the expense of the other. The other sub-interaction can therefore be eliminated, and the choice to take it removed from the interaction term. This means that whenever an action is executed underneath an *alt* constructor, this *alt* constructor is eliminated in the follow-up interaction.

Concretely, executing actions underneath a *alt* constructor is formalized by the following two rules:

For any action $a \in \mathbb{A}_\Omega$: $\dfrac{i_1 \xrightarrow{a} i_1'}{alt(i_1, i_2) \xrightarrow{a} i_1'}$ and $\dfrac{i_2 \xrightarrow{a} i_2'}{alt(i_1, i_2) \xrightarrow{a} i_2'}$

Fig.5.7 illustrates the application of the second rule, which is that of executing an action on the right of a *alt* constructor, using a simple example. We do not provide another example for the left side given that it is similar.

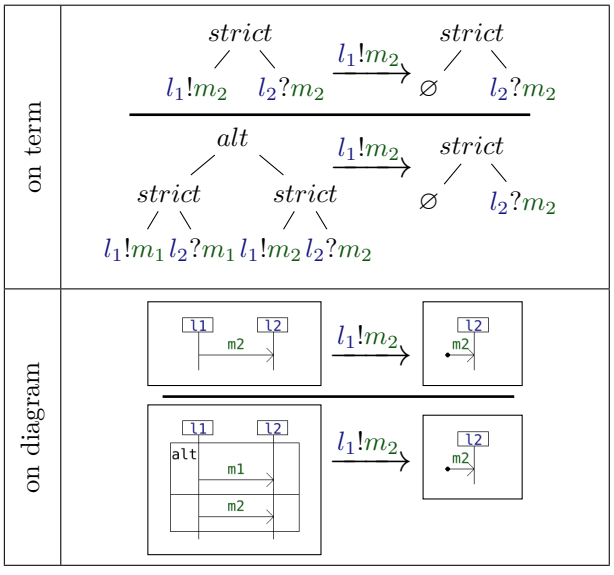


Figure 5.7: Example for executing an action on the right of a *alt* constructor

In the example from Fig.5.7, the interaction models a choice between of two behaviors:

- a behavior describing the passing of a message m_1 from lifeline l_1 to lifeline l_2
- and another which describes the passing of another message m_2 from lifeline l_1 to lifeline l_2

In the second behavior (at the right in the term and on the bottom in the diagram), which execution is illustrated on Fig.5.7, m_2 can be emitted by l_1 so that only its reception $l_2?m_2$ by l_2 remains to be executed.

The "right" rule for the *alt* constructor can therefore be applied (which is shown on Fig.5.7). It implies that $l_1!m_2$ can be executed and that what remains to be executed is simply $l_2?m_2$ (which remains to be executed w.r.t. the right sub-behavior, that has been selected at the expense of the other).

In the first behavior (at the left in the term and at the top in the diagram), m_1 can be emitted by l_1 so that only its reception $l_2?m_1$ by l_2 remains to be executed. Here, the "left" rule for the *alt* operator could be applied, which would likewise imply that $l_1!m_1$ can be executed and that what remains to be executed would be $l_2?m_1$.

5.1.5 Termination predicate

The next rule that we are going to introduce is that for executing actions on the right of *strict* constructors. However before doing so, we need to introduce an intermediary notion which is that of the termination of an interaction.

Termination

If an interaction can express the empty trace ϵ then it means that it can immediately terminate i.e. that it is able to not express anything anymore. The problem of whether or not an interaction i can immediately terminate can be answered systematically via the analysis of the term structure of i.

We provide a solution in the form of the termination predicate "$\downarrow$" given on Def.5.1. The formulation of that predicate is inspired from process algebras as in [98, 99], which we reviewed in Chap.2. For any interaction i, $i\downarrow$ signifies that i can terminate immediately.

Let us remark that the fact that the termination predicate indeed characterizes the ability to express the empty trace i.e. that for any interaction i we have $(i\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i))$ will be proven in Lem.5.4 of the following section (Sec.5.2).

Let us also remark that, in our previous publication [93] we have used a exp_ϵ function instead of the $\downarrow$ predicate. For any interaction i, $exp_\epsilon(i)$ returns whether or not i can express the empty trace ϵ. We used the verb "express" in the name exp_ϵ so as not to risk confusion between this simple static function and a fully-fledged semantics, for which we rather use the verb "accept". With the $\downarrow$ notation, the concept that is evoked is that of an immediate termination of the process. For any interaction i, $i\downarrow$ means that i may terminate immediately without the observation of any further actions. Those two concepts are therefore equivalent.

In any case, the $\downarrow$ predicate can be inferred inductively from the term structure of interactions:

- naturally the empty interaction $\varnothing$ only accepts ϵ, and can only terminate. As a result, we have $\varnothing\downarrow$
- any loop accepts ϵ because it is possible to repeat zero times its content. Therefore, for any $i \in \mathbb{I}_\Omega$, and any $k \in \{S, H, W, P\}$ we have $loop_k(i)\downarrow$
- for interactions of the form $alt(i_1, i_2)$, if either i_1 or i_2 terminates then $alt(i_1, i_2)$ terminates

- for interactions of the form $f(i_1, i_2)$ with f being a scheduling constructor $(strict, seq, par)$ it is required that both i_1 and i_2 terminate for $f(i_1, i_2)$ to terminate

Definition 5.1: Termination "$\downarrow$" predicate

We define inductively the predicate $\downarrow \subset \mathbb{I}_\Omega$ such that for any two interactions i_1 and i_2 from $\mathbb{I}_\Omega$, for any $f \in \{strict, seq, par\}$ and for any $k \in \{S, H, W, P\}$ we have:

$$\frac{}{\varnothing \downarrow} \qquad \frac{i_1 \downarrow}{alt(i_1, i_2) \downarrow} \qquad \frac{i_2 \downarrow}{alt(i_1, i_2) \downarrow} \qquad \frac{i_1 \downarrow \qquad i_2 \downarrow}{f(i_1, i_2) \downarrow} \qquad \frac{}{loop_k(i_1) \downarrow}$$

Non-Termination

Similarly, we can define (Def.5.2) a "non-termination" predicate $\not\downarrow$ from a static analysis of the structure of interactions:

- an interaction that consists of a single action $a \in \mathbb{A}_\Omega$ do not express ϵ (a must be executed) and cannot terminate. As a result we have $a \not\downarrow$
- for interactions of the form $alt(i_1, i_2)$, if both $i_1 \not\downarrow$ and $i_2 \not\downarrow$ then we have $alt(i_1, i_2) \not\downarrow$
- for interactions of the form $f(i_1, i_2)$ with $f \in \{strict, seq, par\}$ (scheduling constructors), it suffices that either $i_1 \not\downarrow$ or $i_2 \not\downarrow$ in order to have $f(i_1, i_2) \not\downarrow$

Definition 5.2: Non-Termination "$\not\downarrow$" predicate

We define inductively the predicate $\not\downarrow \subset \mathbb{I}_\Omega$ such that for any action $a \in \mathbb{A}_\Omega$, for any two interactions i_1 and i_2 from $\mathbb{I}_\Omega$ and for any $f \in \{strict, seq, par\}$ we have:

$$\frac{}{a \not\downarrow} \qquad \frac{i_1 \not\downarrow \qquad i_2 \not\downarrow}{alt(i_1, i_2) \not\downarrow} \qquad \frac{i_1 \not\downarrow}{f(i_1, i_2) \not\downarrow} \qquad \frac{i_2 \not\downarrow}{f(i_1, i_2) \not\downarrow}$$

The termination and non-termination predicates are negations of one another, and, for any interaction term i, one can always decide whether $i \downarrow$ or $i \not\downarrow$.

Lemma 5.1: Termination & non-termination

We have that for any interaction $i \in \mathbb{I}_\Omega$:

$$(i \downarrow) \vee (i \not\downarrow) \qquad \text{and} \qquad (i \downarrow) \Leftrightarrow \neg(i \not\downarrow)$$

Proof. Immediate by induction on the term structure of interactions. □

Illustrating example

Let us consider the example from Fig.5.8. On the right of Fig.5.8 is represented the syntaxic structure of an interaction i, and, on the left of Fig.5.8 the corresponding drawing as a sequence diagram.

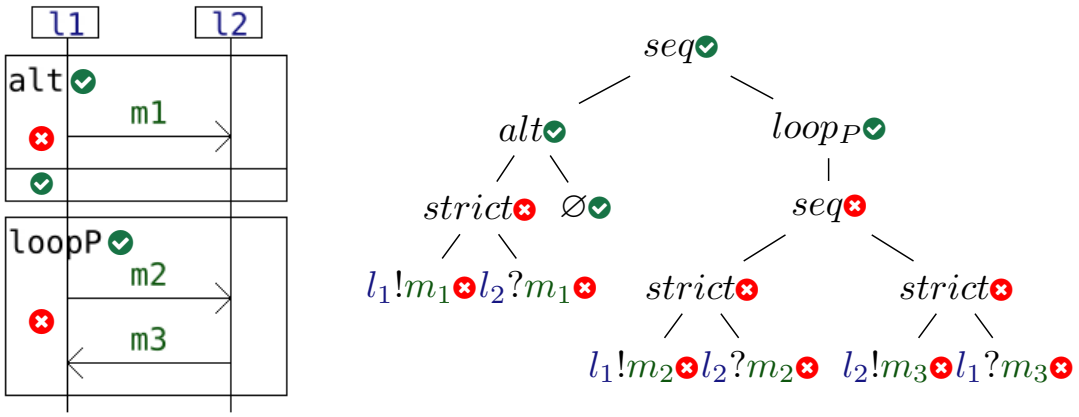


Figure 5.8: Illustration of the termination predicate

In this illustration, we represent the application of the termination predicate $\downarrow$ on the syntaxic structure of the term using two symbols: ✅ and ❌. For any node at position p in the tree structure, if it is decorated with ✅ then it means that $i_{|p} \downarrow$ and if it is decorated with ❌ then it means that $i_{|p} \not\downarrow$. In order to decorate all the nodes of the tree we can proceed as follows:

- as a first step, we can immediately decorate the leaf nodes:
  - if a leaf node contains the empty interaction $\varnothing$ then we decorate it with ✅
  - if a leaf node contains an action $a \in \mathbb{I}_\Omega$ then we decorate it with ❌
- then, the decoration of parent nodes is inferred by their natures (the operation symbol they carry) and by the decoration of child nodes:
  - for nodes containing *strict*, *seq* or *par*, we decorate them with ✅ if both children are decorated with ✅ and, otherwise, we decorate them with ❌
  - for nodes containing *alt*, we decorate them with ✅ if any one child is decorated with ✅ and, otherwise, we decorate them with ❌
  - we always decorate *loop* nodes with ✅

On the example from Fig.5.8, given that the root node is decorated by ✅ we can conclude that the interaction terminates. Indeed, we can choose the right branch of the alternative and choose not to instantiate the loop which results in expressing the empty trace.

On the left of Fig.5.8 we have roughly illustrated the counterpart of this process on the sequence diagram representation.

5.1.6 Executing actions on the right of *strict* operators

We have seen earlier how actions on the left of a *strict* operator might be executed. Given the nature of the *strict* operator we might at first glance imagine that it is not possible to execute actions on the right of

strict but there are situations in which it is actually possible. For instance, we have the trivial $strict(\varnothing, a)$, with $a \in \mathbb{A}_\Omega$ a certain action that is immediately executable so that $strict(\varnothing, a) \xrightarrow{a} \varnothing$.

In fact, an action on the right of a *strict* in a certain $strict(i_1, i_2)$ and such that $i_2 \xrightarrow{a} i_2'$ can be executed if the sub-interaction i_1 (that on the left) can express the empty trace ϵ. Indeed, if it is the case, then nothing prevents action a to be at the first position in a trace expressed by $strict(i_1, i_2)$. We can see that this notion of "expressing the empty trace" corresponds to the termination predicate $\downarrow$ that we have previously defined.

Given $i = strict(i_1, i_2)$, if a can be executed in i_2 and if $i_1 \downarrow$, then a can be executed in i and the follow-up interaction is i_2', which is the follow-up from the execution of a within i_2. Indeed, the execution of a within i in that configuration forces the choice, for i_1 to express the empty trace. As a result, in the next step of the execution, nothing remains to be executed in i_1 and therefore i_1 can be eliminated. Given that i_1 can express the empty trace, this elimination of i_1 does not violate its semantics.

As a result, the definition of the termination predicate $\downarrow$ allows us to define the rule for executing actions on the right of *strict* constructors. This rule is formalized as follows:

For any action $a \in \mathbb{A}_\Omega$: $$\frac{i_2 \xrightarrow{a} i_2'}{strict(i_1, i_2) \xrightarrow{a} i_2'} \; i_1 \downarrow$$

Fig.5.9 illustrates, using two examples, both the application of the $\downarrow$ predicate and that of the rule for the execution of actions on the right of *strict* constructors. To illustrate the application of $\downarrow$, we use the same ✗ and ✓ notations as we did in the previous section.

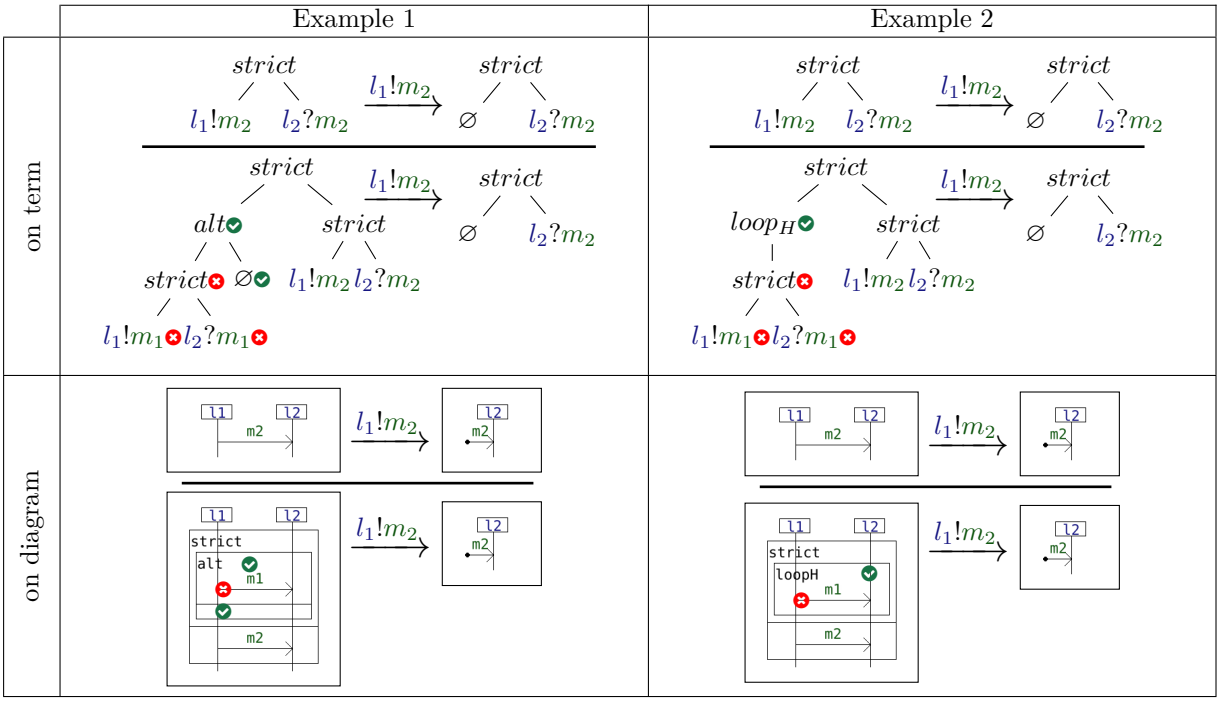


Figure 5.9: Two examples for executing an action on the right of a *strict* constructor

Let us consider "Example 1", on the left of Fig.5.9. We have on the left of the *strict* an alternative, where one branch holds the empty interaction $\varnothing$. As a result, this left sub-interaction terminates. Given that the left sub-interaction i_1 verifies $i_1 \downarrow$ and that an action (here $l_1!m_2$) can be executed on the right sub-interaction i_2, then, as per the previous rule, this action can be executed in $strict(i_1, i_2)$, as shown on

Fig.5.9. The application of the rule leads to the elimination of i_1 and only remains i_2' which results from the execution of the action in the right sub-interaction i_2. In "Example 1" from Fig.5.9, this yields $l_2?m_2$.

The second example from Fig.5.9 can be understood similarly, except that here, the sub-interaction on the left of the *strict*, which is $loop_H(strict(l_1!m_1, l_2?m_1))$ is a $loop_H$ instead of an *alt*. Likewise, it can express the empty trace, but the choice that allows it to do so is not a choice of an alternative branch, but that of not instantiating the $loop_H$.

5.1.7 Evasion & Collision predicates

Let us now introduce the three following notions:

- that, for an interaction, to "evade" a lifeline,
- that, for an interaction, to "collide" with a lifeline
- and that, for an interaction, to be pruned w.r.t. a lifeline (which we will address in Sec.5.1.8)

Evasion

For any interaction $i \in \mathbb{I}_\Omega$, its termination $i\downarrow$ states that it is able to express the empty trace. "Evasion" is a similar, although weaker, notion than "Termination" that can be described as a form of local termination.

For any interaction i, and for any lifeline l, we say that i evades l, which is denoted by the predicate $i \downarrow^{\not\bowtie} l$, if i is able to express at least one trace that does not contain any action occurring on lifeline l. In other words, $i \downarrow^{\not\bowtie} l$ iff there exists a trace t expressed by i such that $\neg(t \not\bowtie l)$ (with the notation from Sec.4.1) i.e. such that t does not have any conflict w.r.t. l (this is proven in Lem.5.5 of Sec.5.2).

Like "termination", "evasion" i.e. the fact that an interaction can (or cannot) express traces that do not involve a specific lifeline can be inferred statically (similarly to $\downarrow$) from the term structure of interactions. We indeed define "$\downarrow^{\not\bowtie}$" in Def.5.3 as an inductive predicate s.t.:

- the empty interaction $\varnothing$ "evades" every lifeline i.e.$\forall\ l \in L$ we have $\varnothing \downarrow^{\not\bowtie} l$
- for an interaction that is reduced to a single action $a \in \mathbb{A}_\Omega$, a evades a lifeline l iff it occurs on a different lifeline i.e. we have $\theta(a) \neq l$
- similarly to termination, any loop may evade any lifeline because it is always possible to express the empty trace, which does not have conflicts w.r.t. l
- for binary constructors the definition of $\downarrow^{\not\bowtie}$ is similar to that of $\downarrow$. For *alt* is it sufficient that one of the two direct sub-interactions evades a lifeline for the parent to evade it. For the scheduling constructors $(seq, strict, par)$, both have to evade the lifeline.

Definition 5.3: Evasion "$\downarrow^{\divideontimes}$" predicate

We define the predicate $\downarrow^{\divideontimes} \subset \mathbb{I}_\Omega \times L$ such that for any lifeline $l \in L$, for any action $a \in \mathbb{A}_\Omega$, for any two interactions i_1 and i_2 from $\mathbb{I}_\Omega$, for any $f \in \{strict, seq, par\}$ and for any $k \in \{S, H, W, P\}$ we have:

$$\frac{}{\varnothing \downarrow^{\divideontimes} l} \qquad \frac{\theta(a) \neq l}{a \downarrow^{\divideontimes} l}$$

$$\frac{i_1 \downarrow^{\divideontimes} l}{alt(i_1, i_2) \downarrow^{\divideontimes} l} \qquad \frac{i_2 \downarrow^{\divideontimes} l}{alt(i_1, i_2) \downarrow^{\divideontimes} l} \qquad \frac{i_1 \downarrow^{\divideontimes} l \qquad i_2 \downarrow^{\divideontimes} l}{f(i_1, i_2) \downarrow^{\divideontimes} l} \qquad \frac{}{loop_k(i_1) \downarrow^{\divideontimes} l}$$

As a side note, about the proximity of $\downarrow$ and $\downarrow^{\divideontimes}$, let us remark that:

- on the one hand, for any interaction $i \in \mathbb{I}_\Omega$, if we have $i \downarrow$ then this implies that $\forall\ l \in L$, we have $i \downarrow^{\divideontimes} l$. Indeed, if an interaction can express the empty trace ϵ then it is able to express a trace with no conflict w.r.t. every lifeline.
- on the other hand, the opposite implication is false: if we suppose that for $\forall\ l \in L$ we have $i \downarrow^{\divideontimes} l$, we do not necessarily have $i \downarrow$. It suffices to consider the following example: $i = alt(l_1!m, l_2!m)$ with $L = \{l_1, l_2\}$. Here we have $i \downarrow^{\divideontimes} l_1$ because we can take the right branch of the *alt*, and $\downarrow^{\divideontimes} l_2$ because we can take the left branch of the *alt*. However we do not have $i \downarrow$ because i must express either $l_1!m$ or $l_2!m$.

Collision

For any interaction i, and for any lifeline l, we say that i collides with l, which is denoted by the predicate $i \not\downarrow^{\divideontimes} l$, if all the traces expressed by i have conflicts w.r.t. l i.e. contain actions occurring on lifeline l.

Like "termination" and "evasion", "collision" can be inferred statically from the term structure of interactions. We indeed define "$\not\downarrow^{\divideontimes}$" in Def.5.4 as an inductive predicate:

- for interactions that consist in a single action $a \in \mathbb{A}_\Omega$, a collides with a lifeline l iff it occurs on l i.e. we have $\theta(a) = l$
- for interactions of the form $alt(i_1, i_2)$ if both i_1 and i_2 collide with l then i also collides with l
- for interactions of the form $f(i_1, i_2)$ with $f \in \{strict, seq, par\}$ (scheduling constructor), it suffices that either i_1 or i_2 collides with l for i to collide with l

Definition 5.4: Collision "$\not\downarrow^{\divideontimes}$" predicate

We define the predicate $\not\downarrow^{\divideontimes} \subset \mathbb{I}_\Omega \times L$ such that for any lifeline $l \in L$, for any action $a \in \mathbb{A}_\Omega$, for any two interactions i_1 and i_2 from $\mathbb{I}_\Omega$ and for any $f \in \{strict, seq, par\}$ we have:

$$\frac{\theta(a) = l}{a \not\downarrow^{\divideontimes} l} \qquad \frac{i_1 \not\downarrow^{\divideontimes} l \qquad i_2 \not\downarrow^{\divideontimes} l}{alt(i_1, i_2) \not\downarrow^{\divideontimes} l} \qquad \frac{i_1 \not\downarrow^{\divideontimes} l}{f(i_1, i_2) \not\downarrow^{\divideontimes} l} \qquad \frac{i_2 \not\downarrow^{\divideontimes} l}{f(i_1, i_2) \not\downarrow^{\divideontimes} l}$$

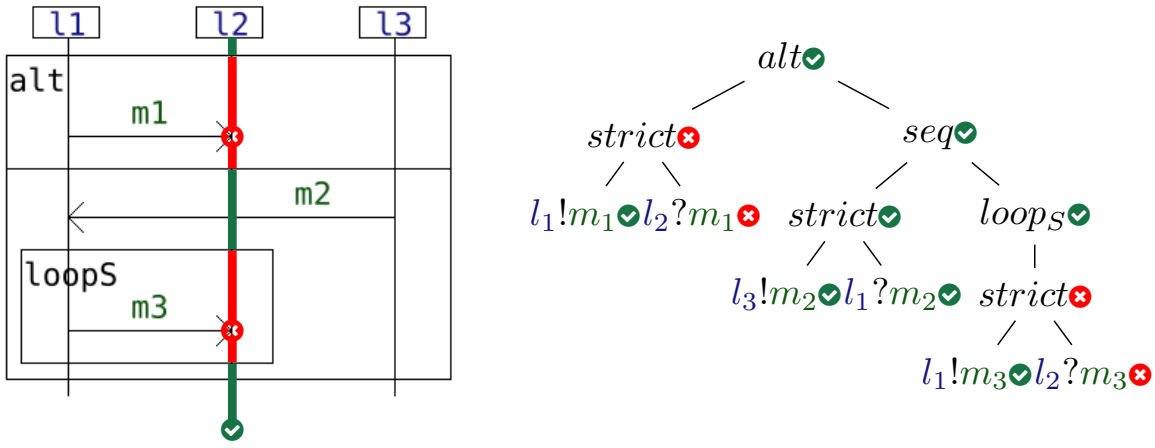


Figure 5.10: Illustration of the evasion predicate (here w.r.t. lifeline l_2)

The evasion and collision predicates are negations of one another (bivalence), and, for any interaction term i and lifeline l, one can always decide whether i evades or collides with l (decidability). Those properties are stated in Lem.5.2.

Lemma 5.2: Decidability & bivalence of the evasion & collision predicates

We have that for any interaction $i \in \mathbb{I}_\Omega$ and any lifeline $l \in L$:

$$(i \downarrow^* l) \vee (i \not\downarrow^* l) \qquad \text{and} \qquad (i \downarrow^* l) \Leftrightarrow \neg(i \not\downarrow^* l)$$

Proof. Immediate by induction on the term structure of interactions. □

Illustrating example

Let us consider the example from Fig.5.10. On the right of Fig.5.10 is represented the syntaxic structure of an interaction i, and, on the left of Fig.5.10 the corresponding drawing as a sequence diagram. In a similar fashion as what we did on the illustrating example for the termination predicate (Fig.5.8), we have here illustrated the application of the evasion predicate (w.r.t. lifeline l_2) on the term structure via the use of ✅ and ❌ symbols. Let us recall that, for any node at position p in the tree structure, if it is decorated with ✅ then it means that $i_{|p} \downarrow^* l$ and if it is decorated with ❌ then it means that $i_{|p} \not\downarrow^* l$. In order to decorate all the nodes of the tree we can proceed as follows:

- as a first step, we can immediately decorate the leaf nodes (which are either actions or $\varnothing$) with the ✅ and ❌ symbols
- then, the decoration of parent nodes is inferred by their natures (the operation symbol they carry) and by the decoration of child nodes

On the left of Fig.5.10 we can illustrate the application of the evasion predicate by drawing a line over that of the lifeline of interest, which is l_2 in this example. This line can be decomposed into several segments

that correspond to specific areas of the diagram (operands) and that are colored either in green or in red. The coloration of the segment depends on whether or not the sub-interaction corresponding to the operand evades or collides with the lifeline of interest. If the first and last segments are colored in green then it means that the diagram evades the lifeline of interest.

In the particular example represented on Fig.5.10, we can see that by taking the right branch of the alternative and by choosing not to instantiate the loop, we can express traces that have no conflict w.r.t. lifeline l_2. As a result the interaction i verifies $i \downarrow^{\nleftrightarrow} l_2$.

5.1.8 The pruning of interactions

Pruning is a process by which an interaction i that can evade a lifeline l (i.e. such that $i \downarrow^{\nleftrightarrow} l$) is rewritten into a new interaction which characterizes exactly all the executions of i that do not involve lifeline l. In other words, the pruning process removes from the input interaction i all the behaviors (and only those behaviors) that may enter into conflict with the input lifeline l. We denote by $i \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'$ the fact that the pruning process applied to interaction i w.r.t. lifeline l returns interaction i'.

Definition

To formalize the notion of pruning we define $i \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'$ as an inductive relation in Def.5.5. Then, in Lem.5.3, we prove, for any interaction i such that $i \downarrow^{\nleftrightarrow} l$, the existence and unicity of the "pruned" interaction i' such that $i \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'$.

Let us remark that the fact that the pruning process applied to i and l indeed returns an interaction i' which characterizes exactly all the executions of i that do not involve lifeline l i.e. that we have $\sigma_d(i') = \{t \in \sigma_d(i) \mid \neg(t \nleftrightarrow l)\}$ will be proven in Lem.5.6 of the following section (Sec.5.2).

Definition 5.5: Pruning relation "$\nleftrightarrow\!\!\!\rightarrow$"

We define the pruning relation $\nleftrightarrow\!\!\!\rightarrow \subset \mathbb{I}_\Omega \times L \times \mathbb{I}_\Omega$ such that for any lifeline $l \in L$, for any $f \in \{strict, seq, par\}$ and for any $k \in \{S, H, W, P\}$:

$$\frac{}{\varnothing \overset{l}{\nleftrightarrow\!\!\!\rightarrow} \varnothing} \qquad \frac{\theta(a) \neq l}{a \overset{l}{\nleftrightarrow\!\!\!\rightarrow} a} \qquad \frac{i_1 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_1 \qquad i_2 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_2}{f(i_1, i_2) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} f(i'_1, i'_2)}$$

$$\frac{i_1 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_1 \qquad i_2 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_2}{alt(i_1, i_2) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} alt(i'_1, i'_2)} \qquad \frac{i_1 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_1}{alt(i_1, i_2) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_1}\ i_2 \not\downarrow^{\nleftrightarrow} l \qquad \frac{i_2 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_2}{alt(i_1, i_2) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_2}\ i_1 \not\downarrow^{\nleftrightarrow} l$$

$$\frac{i_1 \overset{l}{\nleftrightarrow\!\!\!\rightarrow} i'_1}{loop_k(i_1) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} loop_k(i'_1)} \qquad \frac{}{loop_k(i_1) \overset{l}{\nleftrightarrow\!\!\!\rightarrow} \varnothing}\ i_1 \not\downarrow^{\nleftrightarrow} l$$

Properties

In the following we will state some properties of the inductive relation of Def.5.5. The pruning relation and the evasion predicate can be used to describe similar characteristics of interactions terms. Whenever an

interaction evades a lifeline then it means that is can be pruned w.r.t. that lifeline. Reciprocally, whenever an interaction can be pruned w.r.t. a lifeline then it means that it evades that lifeline. Those observations as well as the unicity of pruning are given in Lem.5.3

Lemma 5.3: Conditional existence & unicity of pruned interaction

For any interaction $i \in \mathbb{I}_\Omega$ and for any lifeline $l \in L$ we have:

$$(\exists\ i' \in \mathbb{I}_\Omega \text{ s.t. } i \overset{l}{\nrightarrow} i') \Rightarrow (i \downarrow^{\nrightarrow} l) \quad \text{and} \quad (i \downarrow^{\nrightarrow} l) \Rightarrow (\exists!\ i' \in \mathbb{I}_\Omega \text{ s.t. } i \overset{l}{\nrightarrow} i')$$

Proof. We can reason by induction on the term structure of interactions. □

Description & illustrative examples

The pruning relation $i \overset{l}{\nrightarrow} i'$ eliminates from a given interaction i (s.t. the precondition $i \downarrow^{\nrightarrow} l$ is satisfied) all actions occurring on lifeline l while preserving "a maxima" the original semantics of i i.e. so that $\sigma_d(i') \subseteq \sigma_d(i)$ and $\sigma_d(i')$ is the maximum subset of $\sigma_d(i)$ that contains no trace in which there are actions occurring on l.

So as to preserve the semantics, the interaction term i can only be modified in two manners with the aim to eliminate actions:

1. by forcing the choice of a given sub-interaction in *alt* nodes (illustrated on Fig.5.11)
2. by choosing to forbid the repetition of a sub-interaction in *loop* nodes (illustrated on Fig.5.12)

Those two kinds of modifications strictly correspond to the elimination of some possible executions of i while preserving all others.

We describe in the following the mechanism of pruning formalised in Def.5.5. Let us consider a lifeline l. We then have:

- $\varnothing \overset{l}{\nrightarrow} \varnothing$ because there is nothing to eliminate
- for any action $a \in \mathbb{A}_\Omega$, a is prunable iff $a \downarrow^{\nrightarrow} l$. Therefore, a is not an action that needs to be eliminated and $a \overset{l}{\nrightarrow} a$
- for $i = alt(i_1, i_2)$, in order for the precondition $i \downarrow^{\nrightarrow} l$ to be satisfied, we have either or both of $i_1 \downarrow^{\nrightarrow} l$ or $i_2 \downarrow^{\nrightarrow} l$. If both branches evade l they can be pruned and kept in the interaction term. If only a single one does, we only keep the pruned version of this single branch
- for any scheduling constructor f, if $i = f(i_1, i_2)$, in order to have $i \downarrow^{\nrightarrow} l$ we must have both $i_1 \downarrow^{\nrightarrow} l$ and $i_2 \downarrow^{\nrightarrow} l$. In that case the unique interaction i' such that $i \overset{l}{\nrightarrow} i'$ is simply defined as the scheduling, using f, of the pruned versions of i_1 and i_2
- finally, for loops, i.e. with i of the form $loop_k(i_1)$ with $k \in \{S, H, W, P\}$, we distinguish two cases:

– if $i_1 \not\downarrow^{*} l$ then any execution of i_1 will yield a trace containing actions occurring on l. Therefore it is necessary to forbid the repetition of the loop. This is done by specifying that $loop_k(i_1) \not\overset{l}{\mapsto} \varnothing$

– if $i_1 \downarrow^{*} l$ then it is not necessary to forbid the repetition of the loop, given that sub-interaction i_1 can be pruned and therefore may not yield traces with actions occurring on l. This being the modification which preserves a maximum amount of traces of the semantics, we have, if $i_1 \not\overset{l}{\mapsto} i_1'$ then $loop_k(i_1) \not\overset{l}{\mapsto} loop_k(i_1')$

The recursive nature of the prune relation then guarantees that only the minimally required modifications are done on the interaction term so as to eliminate from it undesired actions.

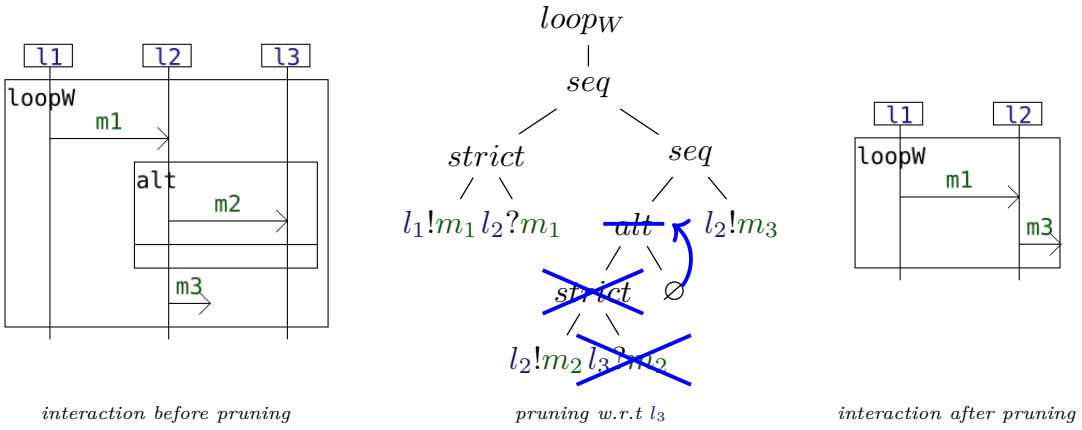


Figure 5.11: Illustration of pruning in the case where a branch of alternative is cut-out

Fig.5.11 illustrates a specific application of the pruning process. We consider an interaction i, drawn on the left part of Fig.5.11 and which term is given in the middle part of Fig.5.11. i is defined over the set $L = \{l_1, l_2, l_3\}$ of lifelines. We then apply pruning w.r.t. lifeline l_3 on i to obtain the interaction drawn on the right part of Fig.5.11. The blue lines represent the rewriting orchestrated by the pruning process. The only action occurring on l_3 in i is $l_3?m_2$. It must be eliminated. As its parent is a scheduling constructor (*strict*), it must also be eliminated. The grand-parent node is an *alt* operator. The right cousin underneath this *alt* is $\varnothing$, which evades l_3. Therefore, we can force the choice of the right branch of this *alt* to solve the pruning. The remaining interaction then does not contain any action occurring on l_3. As explained earlier, the pruning made the minimal modifications to i so as to eliminate $l_3?m_2$. For instance, we could have simply (and naively) forbidden the repetition of the *loop* at the root position; but this would also have eliminated from the semantics of the remaining interaction a number of traces which we do not want to be eliminated.

Fig.5.12 illustrates another specific application of the pruning process. We consider the same interaction i as in Fig.5.11, However we consider the pruning w.r.t. lifeline l_1 instead of l_3. The only action occurring on l_1 in i is $l_1!m_1$. It must be eliminated. Its parent and grand-parent being respectively a *strict* and a *seq* constructor, they must both be eliminated. Finally, a *loop* node is reached. At this point, the only choice is to forbid the repetition of this loop. We therefore replace it by the empty interaction $\varnothing$ (as indicated in blue) to obtain the interaction on the right.

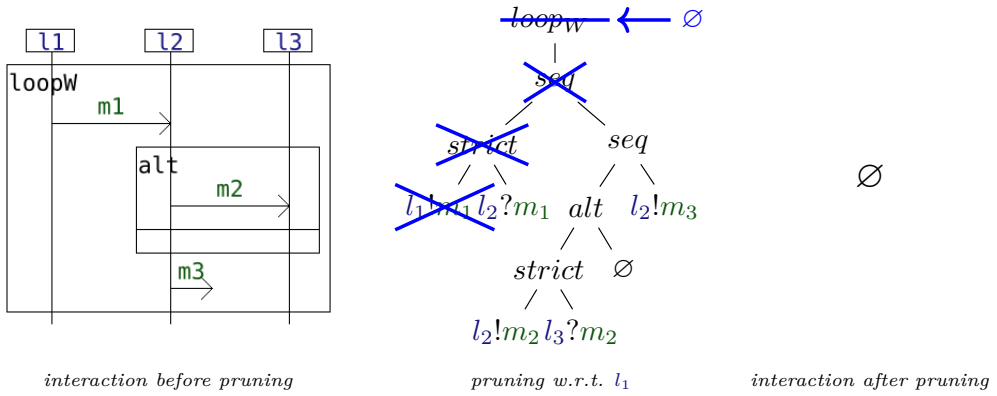


Figure 5.12: Illustration of pruning in the case where the repetition of a loop is forbidden

5.1.9 Executing actions on the right of *seq* constructors

seq being less constraining than *strict*, it is also possible to execute actions that are on the right of a *seq* operator. Given an interaction $seq(i_1, i_2)$, if $i_2 \xrightarrow{a} i_2'$ then it is possible to execute a from $seq(i_1, i_2)$ if a certain condition is met on i_1. However this condition is not as strong as what would be the case for a *strict* operator (i.e. we would require $i_1 \downarrow$).

Here we only need i_1 to be able to express a trace that do not contain any action occurring on lifeline $\theta(a)$ i.e. on the lifeline on which a occurs. Indeed, if this is the case then it means that it is possible to observe no action at all on lifeline $\theta(a)$ during an execution of i_1. This therefore implies that it is possible, during an execution of $seq(i_1, i_2)$ that the first action that is observed on lifeline $\theta(a)$ comes from i_2. The reader may have recognized that this condition is in fact $i_1 \downarrow^{\divideontimes} \theta(a)$, with the evasion predicate that we have defined in Sec.5.1.7.

In the case of *strict*, when executing an action on the right, we simply eliminate entirely the sub-interaction on the left i.e. in $strict(i_1, i_2) \xrightarrow{a} i_2'$ we only keep i_2' and discard i_1. However, for *seq*, we might need to keep i_1 or at least some reconstruction of i_1 on the left. This reconstruction of i_1 that is kept is, in fact, the result i_1' of the pruning process $i_1 \overset{\theta(a)}{\divideontimes\!\longrightarrow} i_1'$ which we introduced in Sec.5.1.8. Keeping i_1' on the left ensures that the first action occurring on lifeline $\theta(a)$ can be a and that we will not contradict the order specified by the *seq* by having some other actions occurring on $\theta(a)$ being later executed from within the left sub-term. Indeed, i_1' do not contain any action occurring on $\theta(a)$. Moreover, keeping i_1' instead of simply getting rid of the left sub-term ensures that we preserve the expressive power of the model i.e. that we do not get an under-approximated model as a follow-up interaction.

As a result, the execution of actions on the right of *seq* constructors is formalized by the following rule:

For any action $a \in \mathbb{A}_\Omega$:
$$\frac{i_1 \overset{\theta(a)}{\divideontimes\!\longrightarrow} i_1' \qquad i_2 \xrightarrow{a} i_2'}{seq(i_1, i_2) \xrightarrow{a} seq(i_1', i_2')}$$

Let us now consider the example from Fig.5.13. Here, lifeline l_1 sends message m to lifeline l_3 and then (with weak sequentially *seq*) lifeline l_2 sends another message m to lifeline l_3. Even though the emission of m by l_2 is within the right sub-interaction, it can happen immediately because the left sub-interaction $strict(l_1!m, l_3?m)$ do not involve in any way lifeline l_2, which can be determined via the predicate

$strict(l_1!m, l_3?m) \downarrow^{*} l_2$. On Fig.5.13, we illustrated the use of the $\downarrow^{*}$ predicate in the same manner as we did in Sec.5.1.7. We can see, on the bottom row, in the overall interaction term, that the left sub-interaction evades the lifeline l_2 which hosts the action to execute $l_2!m$. We can interpreted that as having no interruption between the top of the lifeline and the location on this lifeline of said action. Here we represented this with a bold green line. Indeed, we have no intermediate action that can interrupt the bold green line between the top of lifeline l_2 and $l_2!m$. In this particular example, we did not need to rewrite the left sub-interaction $strict(l_1!m, l_3?m)$, and we used it unaltered so as to reconstruct the follow-up interaction $seq(strict(l_1!m, l_3?m), l_3?m)$. This corresponds to the fact that $strict(l_1!m, l_3?m) \not\xrightarrow{l_2} strict(l_1!m, l_3?m)$. However, as we will see in another example, there are cases where we need to do some actual rewriting of the left sub-interaction using the pruning process.

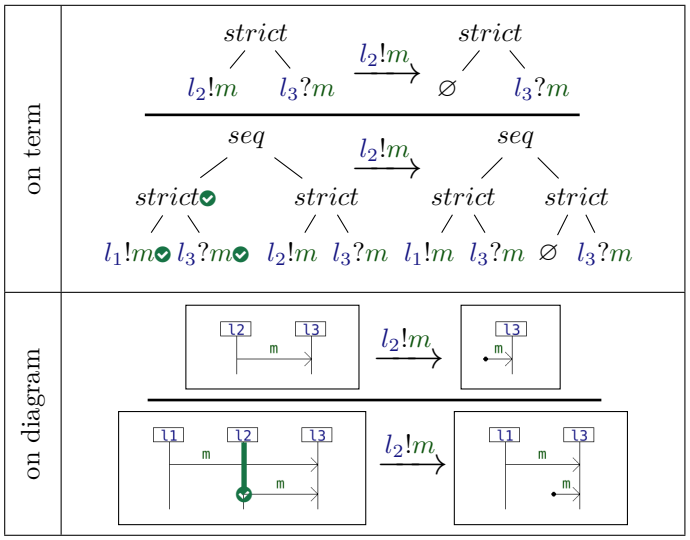


Figure 5.13: Example for executing an action on the right of a *seq* constructor without pruning

Let us now consider the example from Fig.5.14. On the right of the *seq*, we want to execute the action $l_2!m$. Here, the sub-interaction on the left of the *seq* is an alternative $alt(strict(l_1!m, l_2?m), l_1?m)$. It avoids lifeline l_2 because we can choose to take the right branch of the alternative (as illustrated by the ✅ symbols and the bold green line in the diagram representation. However, when reconstructing the follow-up interaction, we need to rewrite the left sub-interaction $alt(strict(l_1!m, l_2?m), l_1?m)$ so as to enforce the choice of the alternative that allows lifeline l_2 to be avoided. This is done by the pruning process, which effect on the interaction term is illustrated on Fig.5.14 in blue. The problematic action in the term is $l_2?m$. If we were to let it be in the reconstructed interaction, we risk a contradiction in the semantics, with $l_2?m$ being able to occur after $l_2!m$ (violating the intended semantics of the *seq* constructor). As a result, we need to eliminate $l_2?m$, which we illustrate with a blue cross on the node. Given that its parent is a *strict* constructor, it must also be eliminated (no choice can be made here), which we illustrate with another blue cross. In the grand-parent node, an *alt* constructor, a choice can be made to solve the contradiction / conflict caused by the action $l_2?m$. Here we can enforce the choice of the right branch of the *alt* by replacing the *alt* node by its right child, which is represented by the blue arrow. At this point we have finished rewriting the left

sub-interaction under the root *seq* constructor. Finally we obtain the follow-up interaction $seq(l_1?m, l_3?m)$.

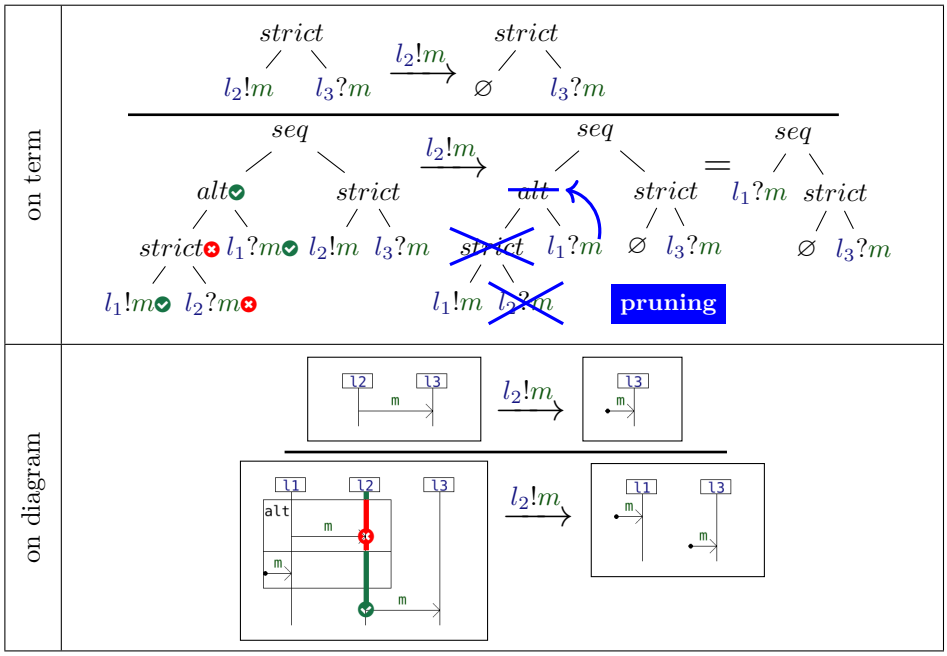


Figure 5.14: Example for executing an action on the right of a *seq* constructor with pruning

5.1.10 Executing actions underneath a $loop_S$ constructor

Repetition constructors (or loops) specify the repetition, according to a certain scheduling policy, of a given behavior. They have a single sub-interaction that is the model of this repeatable behavior. Let us start with the strict loop $loop_S$.

Let us consider $i = loop_S(i_1)$, with i_1 an interaction. Let us suppose the existence of an action a that can be executed in sub-interaction i_1 and let us suppose that its execution in i_1 yields i_1' as a follow-up interaction, Given that i models the repetition of i_1, if a can be executed in i_1 then it can be executed in i too. However, once it is executed, this means that a full repetition of i_1, which starts by a has to be executed. This full repetition to complete corresponds to the execution of i_1' which remains to be done. As a result, the follow-up interaction to i is $strict(i_1', i)$. Indeed, we use *strict* to schedule this first repetition w.r.t. some other repetitions that may occur later-on. The initial interaction $i = loop_S(i_1)$ remains unaltered because it is the model for further repetitions.

Similarly to the case of the *alt* constructor, the execution of an action comes with forcing a choice, which is here that of repeating once (at least) the repeatable behavior. This choice can be understood as that of "instantiating" the loop i.e. creating an instance of the sub-behavior that is repeated. It is then the execution of this instance (here i_1') that corresponds to the occurrence of the repeated behavior. And it is this instance that is scheduled w.r.t. the original loop (which can be later repeated and therefore must not be altered) using (in this case) the *strict* scheduling constructor.

The execution of actions underneath $loop_S$ constructors therefore correspond to the following rule:

For any action $a \in \mathbb{A}_\Omega$: $$\frac{i_1 \xrightarrow{a} i_1'}{loop_S(i_1) \xrightarrow{a} strict(i_1', loop_S(i_1))}$$

Let us remark that executing actions within loops, and therefore instanciating those loops, generally complexifies the interaction term. Indeed, instantiating a loop might add new nodes and new actions to the interaction term. This offers a contrast with the execution of actions in any other case, which rather simplifies the interaction term by removing actions.

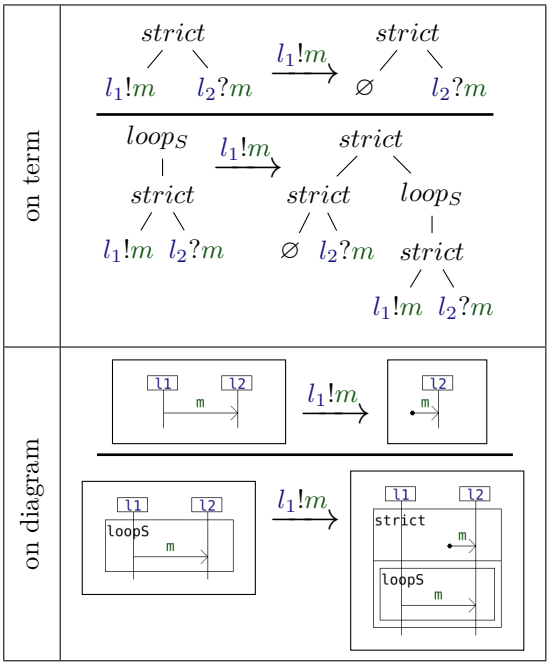


Figure 5.15: Example for executing an action underneath a $loop_S$ constructor

Fig.5.15 illustrates, on a simple example, the application of the rule. In this example, the interaction $i = loop_S(i_1)$ models the repetition, according to strict sequencing, of a behavior describing the passing of a message m from lifeline l_1 to lifeline l_2, which is modelled by interaction $i_1 = strict(l_1!m, l_2?m)$

In this behavior, m can be emitted by l_1 so that only its reception $l_2?m$ by l_2 remains to be executed. The rule for the $loop_S$ constructor can therefore be applied (which is shown on Fig.5.15). It implies that $l_1!m$ can be executed and that what remains to be executed is the strict sequencing of $l_2?m$, which is what remains to be executed of the instantiated behavior and the original interaction i.

5.1.11 Executing actions underneath a $loop_P$ constructor

Executing actions underneath $loop_P$ constructors works in a similar fashion except that instead of scheduling the remaining behavior w.r.t. the loop using *strict*, we rather use *par*. This therefore correspond to the following rule:

For any action $a \in \mathbb{A}_\Omega$: $$\frac{i_1 \xrightarrow{a} i_1'}{loop_P(i_1) \xrightarrow{a} par(i_1', loop_P(i_1))}$$

Fig.5.16 illustrates, on a simple example, the application of the rule. We have here the sub-behavior $i_1 = strict(l_1!m, l_2?m)$ that is repeated, similarly to the previous example from Fig.5.15 but, instead of

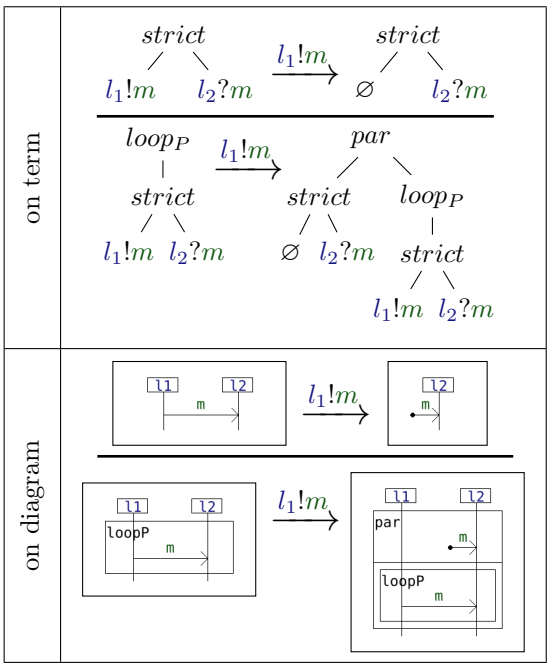


Figure 5.16: Example for executing an action underneath a $loop_P$ constructor

using the $loop_S$ constructor, we use here the $loop_P$ constructor.

Action $l_1!m$ can be immediately executed and there remains to be executed $par(l_2?m, loop_P(strict(l_1!m, l_2?m)))$. In the previous example, in which there remained $strict(l_2?m, loop_P(strict(l_1!m, l_2?m)))$ to be executed we could not once again immediately execute $l_1!m$. With this kind of loop however, thanks to the use of the *par* constructor, we can do so. The action $l_1!m$ underneath the loop can once again be directly executed and this results in a second instanciation of the behavior. This is illustrated on Fig.5.17.

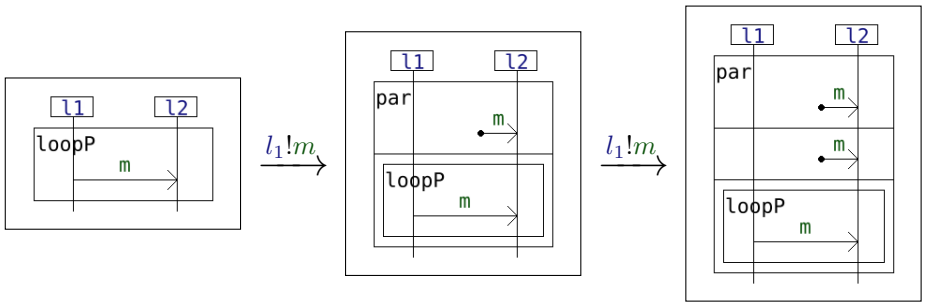


Figure 5.17: Showcasing several consecutive instantiations of the $loop_P$ from the example of Fig.5.16

Fig.5.17 illustrates two points:

- the iterative nature of the execution of actions within interaction terms. This allows the construction of paths of executions which are fundamental for the definition of the operational semantics. In Fig.5.17, we can see that $l_1!m.l_1!m$ is a prefix of traces accepted by the initial interaction $loop_P(strict(l_1!m, l_2?m))$
- the fact that it is not necessary for previous instances of a sub-behavior to be entirely executed before that another instance can be created. Here a second instance of the behavior is created (the second pending reception $l_2?m$) even though the first one has not been executed entirely. This of course depends depends on the nature of the loop constructor that is used, as explained in Sec.4.2.1.

5.1.12 Executing actions underneath a $loop_H$ constructor

Executing actions underneath $loop_H$ constructors also works similarly. Here, instead of scheduling the remaining behavior w.r.t. the loop using *strict* or *par*, we use *seq*. This therefore correspond to the following rule:

For any action $a \in \mathbb{A}_\Omega$: $$\frac{i_1 \xrightarrow{a} i_1'}{loop_H(i_1) \xrightarrow{a} seq(i_1', loop_H(i_1))}$$

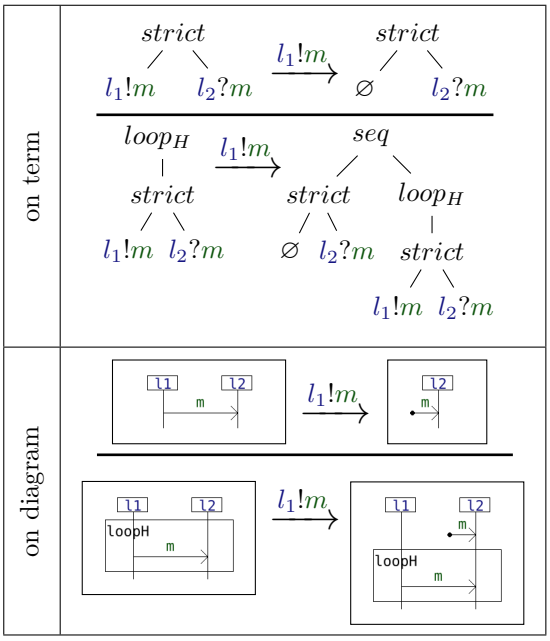


Figure 5.18: Example for executing an action underneath a $loop_H$ constructor

Fig.5.18 illustrates, on a simple example, the application of the rule. As in the two previous examples, we repeat the sub-behavior $i_1 = strict(l_1!m, l_2?m)$. We can see that the follow-up interaction is $seq(l_2?m, loop_H(strict(l_1!m, l_2?m)))$. As was the case for the $loop_P$ constructor, we can also here immediately execute $l_1!m$ once again, resulting in the creation of a second instance of the repeatable behavior. We illustrate this on Fig.5.19.

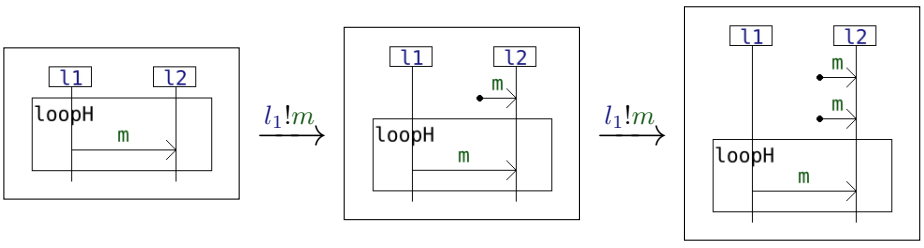


Figure 5.19: Showcasing several consecutive instantiations of the $loop_H$ from the example of Fig.5.18

In the following we will introduce the last repetition constructor which is $loop_W$. $loop_W$ also uses weak sequencing as a scheduler. However we will see that it is not the same constructor as $loop_H$. This will be made clear using an illustrative example.

5.1.13 Executing actions underneath a $loop_W$ constructor

Whenever we execute an action underneath a $loop_W$ constructor we cannot know in advance if this action comes from the first instance of the repeated behavior or if it comes from later instances. Indeed, it is possible to execute at first an action that comes from another instance than the first. This is illustrated on Fig.5.20 on an example, which is the same example that we have used:

- on page 92 to showcase the difference between the weak Kleene closure $\overset{\cdot}{*}$ and the weak Head-First closure $\overset{\cdot}{*}$ operators on sets of traces
- and on page 102 (Fig.4.6) to showcase the difference between the $loop_H$ and $loop_W$ constructors from the point of view of the denotational semantics

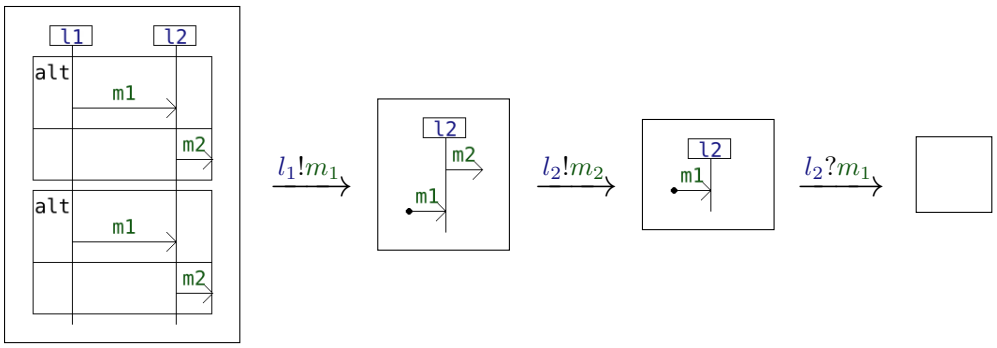


Figure 5.20: Executing at first an action from the second instance of a behavior

Here, we have two instances of the repeatable behavior that is $i_1 = alt(strict(l_1!m_1, l_2?m_1), l_2!m_2)$ and those two instances are repeated using the weak sequencing constructor seq so that the overall interaction is $seq(i_1, i_1)$. As can be seen on Fig.5.20 both instances of action $l_1!m_1$ are immediately executable. The second one (at the bottom) is indeed immediately executable because, thanks to pruning, we can force the choice of the right branch of the first alternative which evades lifeline l_1. Using the aforementioned rules for executing actions on the right of seq constructors, we therefore reach the interaction displayed on Fig.5.20 (second from the left). From that point on the execution of the remainder is straightforward, and, in total, we can conclude that the trace $t = l_1!m_1.l_2!m_2.l_2?m_1$ is expressed by the interaction.

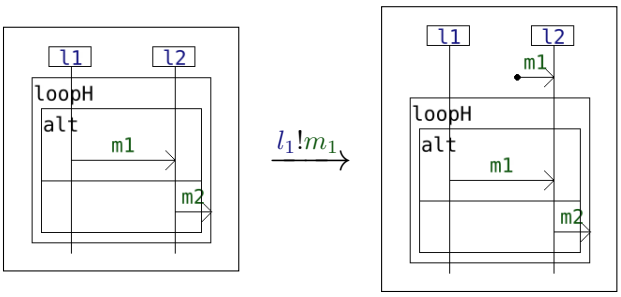


Figure 5.21: Illustrating the restriction associated to the $loop_H$ "Head-First" loop

Now, if we were to try to reproduce this behavior (i.e. the trace $t = l_1!m_1.l_2!m_2.l_2?m_1$) using a $loop_H$ constructor i.e. with $loop_H(i_1)$, we would get what is illustrated on Fig.5.21. We can manage to execute the

first action $l_1!m_1$ and then there remains to be executed the interaction represented on the right of Fig.5.21. We can see that the second action of t which is $l_2!m_2$ is not immediately executable in this interaction. Indeed, the presence of $l_2?m_1$ at the top of the diagram prevents it to be executed.

As we have indicated multiple times earlier (and by definition) the $loop_H$ constructor is associated to the weak HF-closure $\overset{\frown}{;}{}^*$ and not to the K-closure $;^*$. It is therefore expected that t could not be recognized by $loop_H(i_1)$ in this example.

We will now see how we can define an operational rule for $loop_W$ so that $loop_W(i_1)$ is able to recognize t. This rule is as follows:

For any action $a \in \mathbb{A}_\Omega$: $$\frac{i_1 \xrightarrow{a} i_1' \qquad loop_W(i_1) \overset{\theta(a)}{\nrightarrow} i'}{loop_W(i_1) \xrightarrow{a} seq(i', seq(i_1', loop_W(i_1)))}$$

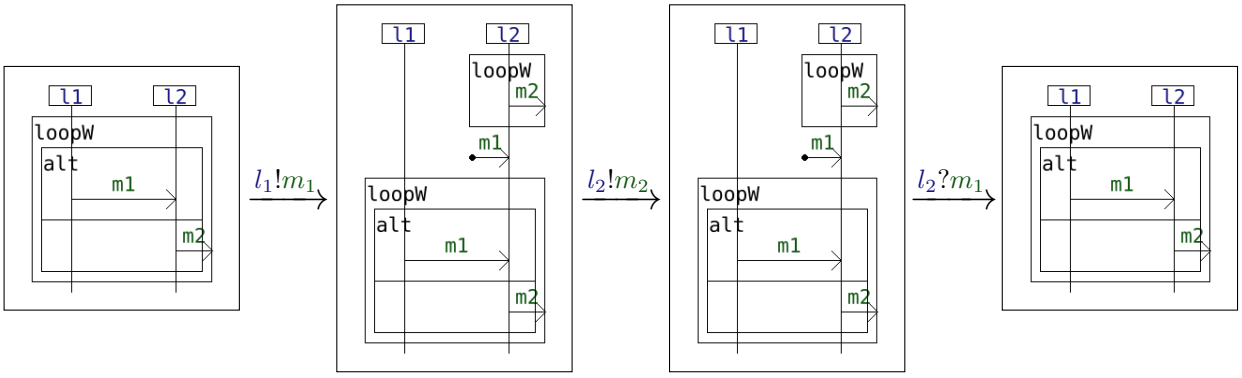


Figure 5.22: Reproducing trace t using the $loop_W$ constructor

On Fig.5.22 we then use this rule to execute the interaction $loop_W(i_1)$ so as to show that it can indeed recognize the trace t (with $i_1 = alt(strict(l_1!m_1, l_2?m_1), l_2!m_2)$ and $t = l_1!m_1.l_2!m_2.l_2?m_1$ from the previous examples). Starting from the left of Fig.5.22 we can, as in the previous case, execute $l_1!m_1$. However, by contrast to the previous case, the follow-up interaction (second from the left) contains $loop_W(l_2!m_2)$ at the top of the diagram (resulting from the pruning of the original $loop_W$). This then allows the execution of $l_2!m_2$, leading to the third interaction. From that third interaction, $l_2?m_1$ can then be executed given that we can choose not to instantiate the above loop. Hence we have recognized t.

With this special rule for $loop_W$ we can recognize traces obtained using the weak Kleene closure $\overset{\frown}{;}{}^*$ (this will be formally proven in Sec.5.2). Let us remark that this can be described as delaying the determination of as part of which instance of the loop the initial action $l_1!m_1$ was executed. If we have then immediately executed $l_2?m_1$ then this would mean that $l_1!m_1$ was executed as part of the first instance. However, given that we have executed $l_2?m_2$ once, this means that $l_1!m_1$ was executed as part of the second instance.

Let us recall that in Lem.4.26 from Chap.4 we have proven that for the strict sequencing and interleaving operators, their respective HF and K closures were equivalent. As a result the constructors $loop_S$ and $loop_P$ are sufficient to recognize traces obtained from the strict Kleene closure and the interleaving Kleene closure. However, given that the weak HF closure and the weak Kleene closure are different operators, we have defined $loop_H$ and $loop_W$ as distinct counterparts in the world of interaction terms.

5.1.14 Formalisation of the set of rules

Now that we have introduced and illustrated the different manners in which actions can be executed in interactions, we formalize this process as an execution relation in Def.5.6.

Definition 5.6: Execution relation for labelled interactions

We define the execution relation $\rightarrow \subset \mathbb{I}_\Omega \times \mathbb{A}_\Omega \times \mathbb{I}_\Omega$ such that for any action $a \in \mathbb{A}_\Omega$:

$$\frac{}{a \xrightarrow{a} \varnothing}$$

$$\frac{i_1 \xrightarrow{a} i_1'}{alt(i_1, i_2) \xrightarrow{a} i_1'} \qquad \frac{i_2 \xrightarrow{a} i_2'}{alt(i_1, i_2) \xrightarrow{a} i_2'}$$

$$\frac{i_1 \xrightarrow{a} i_1'}{par(i_1, i_2) \xrightarrow{a} par(i_1', i_2)} \qquad \frac{i_2 \xrightarrow{a} i_2'}{par(i_1, i_2) \xrightarrow{a} par(i_1, i_2')}$$

$$\frac{i_1 \xrightarrow{a} i_1'}{strict(i_1, i_2) \xrightarrow{a} strict(i_1', i_2)} \qquad \frac{i_2 \xrightarrow{a} i_2'}{strict(i_1, i_2) \xrightarrow{a} i_2'}\ i_1 \downarrow$$

$$\frac{i_1 \xrightarrow{a} i_1'}{seq(i_1, i_2) \xrightarrow{a} seq(i_1', i_2)} \qquad \frac{i_1 \not\xrightarrow{\theta(a)} i_1' \qquad i_2 \xrightarrow{a} i_2'}{seq(i_1, i_2) \xrightarrow{a} seq(i_1', i_2')}$$

$$\frac{i_1 \xrightarrow{a} i_1'}{loop_S(i_1) \xrightarrow{a} strict(i_1', loop_S(i_1))} \qquad \frac{i_1 \xrightarrow{a} i_1'}{loop_H(i_1) \xrightarrow{a} seq(i_1', loop_H(i_1))}$$

$$\frac{i_1 \xrightarrow{a} i_1' \qquad loop_W(i_1) \not\xrightarrow{\theta(a)} i'}{loop_W(i_1) \xrightarrow{a} seq(i', seq(i_1', loop_W(i_1)))} \qquad \frac{i_1 \xrightarrow{a} i_1'}{loop_P(i_1) \xrightarrow{a} par(i_1', loop_P(i_1))}$$

This execution relation defines, for any interaction, which of its actions can be executed, and, if so, which interactions may result from those executions. This constitutes the "small-step" of the small-step operational semantics σ_o which we define in Def.5.7.

Definition 5.7: Operational semantics for labelled interactions

For any signature $\Omega = (L, M)$, we define $\sigma_o : \mathbb{I}_\Omega \rightarrow \mathcal{P}(\mathbb{T}_\Omega)$ by:

$$\frac{i \downarrow}{\epsilon \in \sigma_o(i)} \qquad \frac{t \in \sigma_o(i') \qquad i \xrightarrow{a} i'}{a.t \in \sigma_o(i)}$$

In the following section, we ensure the correctness of the operational-style semantics σ_o we have just defined with regards to the fundational denotational-style semantics σ_d from Chap.4. This proof of equivalence amounts to proving that, for any interaction i, the sets $\sigma_o(i)$ and $\sigma_d(i)$ are equal.

5.2 Proof of equivalence between σ_o and σ_d

In this section, a proof for the equivalence between the denotational semantics σ_d from Sec.4 and the operational semantics σ_o from Sec.5.1 will be presented.

Given that we want to characterize σ_o w.r.t. σ_d, a first step is to characterize the intermediate notions that enter into the definition of σ_o w.r.t. σ_d:

- in Sec.5.2.1 we characterize the notions of termination and evasion w.r.t. σ_d,
- in Sec.5.2.2 we characterize the notion of pruning w.r.t. σ_d.

Once this is done, the proof can be written in two steps:

- in Sec.5.2.3 we prove that for any interaction i we have $\sigma_o(i) \subset \sigma_d(i)$,
- in Sec.5.2.4 we prove that for any interaction i we have $\sigma_o(i) \supset \sigma_d(i)$.

5.2.1 Properties of σ_d w.r.t. the termination and evasion predicates

The termination predicate $\downarrow$ characterizes the fact that an interaction can express the empty trace ϵ and therefore that it is in its semantics. As a result we formulate and prove this in Lem.5.4.

Lemma 5.4: Characterization of termination w.r.t. σ_d

For any $i \in \mathbb{I}_\Omega$:

$$(i\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i))$$

Proof. Let us prove the equivalence of both predicate by induction on the term structure of i.

- If $i = \varnothing$ the empty interaction, then we have both $\varnothing\downarrow$ and $\epsilon \in \sigma_d(\varnothing)$.
- If $i \in \mathbb{A}_\Omega$, we have neither $i\downarrow$ nor $\epsilon \in \sigma_d(i)$.
- Let us now suppose that i is of the form $strict(i_1, i_2)$, with i_1 and i_2 two sub-interactions that satisfy the induction hypotheses $(i_1\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i_1))$ and $(i_2\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i_2))$.

  $\Leftarrow$ Let us suppose that $\epsilon \in \sigma_d(i)$. By definition of σ_d for the *strict* constructor, this implies the existence of $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ such that $\epsilon \in (t_1; t_2)$. Then, as per Lem.4.8, this implies that $t_1 = \epsilon$ and $t_2 = \epsilon$. We can therefore apply the induction hypotheses, to obtain that $i_1\downarrow$ and $i_2\downarrow$. This in turn means that $strict(i_1, i_2)\downarrow$ by definition of the termination predicate.

  $\Rightarrow$ Reciprocally, if $strict(i_1, i_2)\downarrow$, this means that both $i_1\downarrow$ and $i_2\downarrow$. As per the induction hypotheses, this means that $\epsilon \in \sigma_d(i_1)$ and $\epsilon \in \sigma_d(i_2)$. Therefore $\epsilon \in \sigma_d(i)$.
- For interactions of the form $par(i_1, i_2)$ and $seq(i_1, i_2)$, the reasoning is the same as for the previous case except that we use properties on (respectively) the operators $||$ (Lem.4.3) and $;_*$ (Lem.4.13).
- Let us now suppose that i is of the form $alt(i_1, i_2)$, with i_1 and i_2 two sub-interactions that satisfy the induction hypotheses $(i_1\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i_1))$ and $(i_2\downarrow) \Leftrightarrow (\epsilon \in \sigma_d(i_2))$.

⇐ Let us suppose that $\epsilon \in \sigma_d(i)$. By definition of σ_d, this means that either $\epsilon \in \sigma_d(i_1)$ or $\epsilon \in \sigma_d(i_2)$ or both. Let us suppose that it is in $\sigma_d(i_1)$ (the other cases can be treated similarly). As per the induction hypothesis, we therefore have $i_1 \downarrow$. Then, by definition of the termination predicate, this implies that given that $alt(i_1, i_2) \downarrow$.

⇒ Reciprocally, if $alt(i_1, i_2) \downarrow$, this means that either $i_1 \downarrow$ or $i_2 \downarrow$ (or both). Let us suppose we have $i_1 \downarrow$. As per the induction hypothesis, this means that $\epsilon \in \sigma_d(i_1)$. Therefore $\epsilon \in \sigma_d(i)$.

- Let us finally consider the case where i is of the form $loop_k(i_1)$, with $k \in \{S, H, W, P\}$. By definition, we always have $i \downarrow$ and $\epsilon \in \sigma_d(i)$.

□

As explained in Sec.5.1.7, the evasion predicate $\downarrow^{\not\times}$, applied to an interaction i and a lifeline l, characterizes the fact for i to be able to express traces that have no conflict w.r.t. lifeline l i.e. that "evades" l. This characterization, is formulated w.r.t. the semantics σ_d and proven in Lem.5.5.

Lemma 5.5: Characterization of $\downarrow^{\not\times}$ w.r.t. σ_d

For any lifeline $l \in L$ and any interaction $i \in \mathbb{I}_\Omega$:

$$(i \downarrow^{\not\times} l) \Leftrightarrow (\exists\ t \in \sigma_d(i), \neg(t \not\times l))$$

In other words, i evades l iff it can express a trace t that do not have any conflict w.r.t. l.

Proof. Let us consider a certain lifeline $l \in L$, and let us reason by induction on the term structure of i.

- If $i = \varnothing$ the empty interaction, then we have of course $\varnothing \downarrow^{\not\times} l$ and $\epsilon \in \sigma_d(\varnothing)$ satisfies $\neg(\epsilon \not\times l)$.
- If $i = a \in \mathbb{A}_\Omega$, we have:
  - $a \downarrow^{\not\times} l$ iff $\theta(a) \neq l$
  - and, given that $\sigma_d(i) = \{a\}$, a single trace made of that single action, we have that trace a verifies $\neg(a \not\times l)$ iff $\theta(a) \neq l$

  Those two conditions are therefore equivalent.
- Let us now suppose that i is of the form $strict(i_1, i_2)$, with i_1 and i_2 two sub-interactions that satisfy the induction hypotheses.

  ⇒ Let us suppose that $i \downarrow^{\not\times} l$. By definition, this implies that both $i_1 \downarrow^{\not\times} l$ and $i_2 \downarrow^{\not\times} l$. We can therefore apply the induction hypotheses, which lets us consider two traces $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ such that $\neg(t_1 \not\times l)$ and $\neg(t_2 \not\times l)$. By definition of σ_d, we have $(t_1; t2) \subset \sigma_d(i)$. Then, by Lem.4.14, we have that $\exists\ t \in (t_1; t2)$ s.t. $\neg(t \not\times l)$ and this trace is in $\sigma_d(i)$.

⇐ Reciprocally, if $\exists\ t \in \sigma_d(strict(i_1, i_2))$ s.t. $\neg(t \divideontimes l)$, then, by definition of σ_d, this means that there exist two traces $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ s.t. $t \in (t_1; t_2)$. Then, per Lem.4.14, we have $\neg(t_1 \divideontimes l)$ and $\neg(t_2 \divideontimes l)$. We can therefore apply the induction hypothesis which implies that $i_1 \downarrow^{\divideontimes} l$ and $i_2 \downarrow^{\divideontimes} l$, which, per the definition of $\downarrow^{\divideontimes}$, implies that $i \downarrow^{\divideontimes} l$.

- For interactions of the form $par(i_1, i_2)$ and $seq(i_1, i_2)$, the reasoning is the same as for the previous case except that we reason on (respectively) the operators $||$ and $;_{\divideontimes}$ over both of which the conflict $\divideontimes$ is also distributive according to Lem.4.14.

- Let us now suppose that i is of the form $alt(i_1, i_2)$, with i_1 and i_2 two sub-interactions that satisfy the induction hypotheses.

  ⇒ Let us suppose that $i \downarrow^{\divideontimes} l$. By definition of $\downarrow^{\divideontimes}$, we have either or both $i_1 \downarrow^{\divideontimes} l$ and $i_2 \downarrow^{\divideontimes} l$. Let us suppose we have $i_1 \downarrow^{\divideontimes} l$ (the other cases are similar). By the induction hypothesis, we have $\exists\ t_1 \in \sigma_d(i_1)$ s.t. $\neg(t_1 \divideontimes l)$. We then simply observe that $\sigma_d(i_1) \subset \sigma_d(i)$.

  ⇐ Reciprocally, if $\exists\ t \in \sigma_d(alt(i_1, i_2))$ s.t. $\neg(t \divideontimes l)$, then, by definition of σ_d, this means that this trace t must be found in either $\sigma_d(i_1)$ or $\sigma_d(i_2)$. Let us suppose the first case (the other is similar). We can therefore apply the induction hypothesis which implies that $i_1 \downarrow^{\divideontimes} l$. Then, by the definition of $\downarrow^{\divideontimes}$, this implies that $i \downarrow^{\divideontimes} l$.

- Let us finally consider the case where i is of the form $loop_k(i_1)$, with $k \in \{S, H, W, P\}$. By definition, we always have $i \downarrow^{\divideontimes} l$ and the empty trace $\epsilon \in \sigma_d(i)$ verifies $\neg(\epsilon \divideontimes l)$.

□

5.2.2 Properties of σ_d w.r.t. pruning

As explained in Sec.5.1.8, for any lifeline $l \in L$, pruning can be applied to an interaction i that satisfies $i \downarrow^{\divideontimes} l$ to obtain a unique (Lem.5.3) rewritten interaction which exactly expresses the traces that can be expressed by i and that do not involve lifeline l. This characterization of pruning w.r.t. the σ_d semantics is formulated and proven in Lem.5.6.

Lemma 5.6: Characterization of pruning w.r.t. σ_d

For any lifeline $l \in L$ and for any two interactions i and i' from $\mathbb{I}_\Omega$:

$$(i \overset{l}{\divideontimes\!\!\rightarrow} i') \Rightarrow (\sigma_d(i') = \{t \in \sigma_d(i) \mid \neg(t \divideontimes l)\})$$

Proof. Let us consider a certain lifeline $l \in L$, and let us reason by induction on the term structure of i.

- If $i = \varnothing$ the empty interaction, we have $\varnothing \overset{l}{\divideontimes\!\!\rightarrow} \varnothing$ and $\sigma_d(\varnothing) = \{\epsilon\}$. Given that the empty trace ϵ has no conflict with any lifeline, the property is satisfied.

- If $i = a \in \mathbb{A}_\Omega$, the precondition $a \downarrow^{\divideontimes} l$ implies that $\theta(i) \neq l$. Also, we have $a \overset{l}{\divideontimes\!\!\rightarrow} a$ and $\sigma_d(a) = \{a\}$ has a single trace with no conflict w.r.t. lifeline l. The property therefore holds.

- Let us now suppose that i is of the form $strict(i_1, i_2)$, with i_1 and i_2 two sub-interactions that satisfy induction hypotheses i.e. such that, given $i_1 \overset{l}{\divideontimes\!\!\rightarrow} i'_1$ and $i_2 \overset{l}{\divideontimes\!\!\rightarrow} i'_2$ we have $\sigma_d(i'_1) = \{t_1 \in \sigma_d(i_1) \mid \neg(t_1 \divideontimes l)\}$ and $\sigma_d(i'_2) = \{t_2 \in \sigma_d(i_2) \mid \neg(t_2 \divideontimes l)\}$. Also, by definition of pruning, we have $i \overset{l}{\divideontimes\!\!\rightarrow} strict(i'_1, i'_2)$ and let us denote it by $i' = strict(i'_1, i'_2)$ for short.

  $\subset$ Let us then consider $t \in \sigma_d(i')$. By definition there exist $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i'_2)$ such that $t \in (t_1; t_2)$. By the induction hypothesis, we have $t_1 \in \sigma_d(i_1)$ and $\neg(t_1 \divideontimes) l$ and $t_2 \in \sigma_d(i_2)$ and $\neg(t_2 \divideontimes) l$. Therefore we have $t \in \sigma_d(i_1); \sigma_d(i_2) = \sigma_d(i)$, and, as per Lem.4.14, $\neg(t \divideontimes l)$.

  $\supset$ Let us consider $t \in \sigma_d(i)$ such that $\neg(t \divideontimes l)$. By definition of σ_d on the *strict* constructor this implies the existence of $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ s.t. $t \in (t_1; t_2)$. Then, as per Lem.4.14, the fact that $\neg(t \divideontimes l)$ implies that both $\neg(t_1 \divideontimes l)$ and $\neg(t_2 \divideontimes l)$. According to the induction hypothesis, this means that $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i'_2)$. Therefore $(t_1; t_2) \subset \sigma_d(i')$ and hence $t \in \sigma_d(i')$.

- For interactions of the form $par(i_1, i_2)$ and $seq(i_1, i_2)$, the reasoning is the same as for the previous case except that we reason on (respectively) the operators $||$ and $;_{\divideontimes}$ over both of which the conflict $\divideontimes$ is also distributive according to Lem.4.14.

- Let us now suppose that i is of the form $loop_k(i_1)$ with $k \in \{S, H, W, P\}$ and let us note $\diamond$ the corresponding operator on sets of traces, i.e. $\diamond = ;$ if $k = S$, $\diamond = ;^{\curvearrowleft}_{\divideontimes}$ if $k = H$, $\diamond = ;_{\divideontimes}$ if $k = W$ and $\diamond = ||$ if $k = P$. We then have:

  – if $i_1 \not\downarrow^{\divideontimes} l$, then $loop_k(i_1) \overset{l}{\divideontimes\!\!\rightarrow} \varnothing$. As per the reciprocate of Lem.5.5, if i_1 does not avoid l, this means that all the traces from $\sigma_d(i_1)$ have conflicts with l. As per Lem.4.19 this means that all the traces obtained from merging traces from i_1 have conflicts with l. Therefore the empty trace ϵ is the only trace from $\sigma_d(loop_k(i_1))$ which has no conflict with l. Given that $\sigma_d(\varnothing) = \{\epsilon\}$, the property holds.

  – if $i_1 \downarrow^{\divideontimes} l$, then there exists a unique i'_1 such that $i_1 \overset{l}{\divideontimes\!\!\rightarrow} i'_1$ and we suppose the induction hypothesis $\sigma_d(i'_1) = \{t_1 \in \sigma_d(i_1) \mid \neg(t_1 \divideontimes l)\}$. Then, using Lem.4.19 we have:

$$\begin{aligned}
\sigma_d(loop_k(i'_1)) &= \sigma_d(i'_1)^{\diamond *} \\
&= \{t \in \sigma_d(i_1) \mid \neg(t \divideontimes l)\}^{\diamond *} \\
&= \{t \in \sigma_d(i_1)^{\diamond *} \mid \neg(t \divideontimes l)\} \\
&= \{t \in \sigma_d(loop_k(i_1))) \mid \neg(t \divideontimes l)\}
\end{aligned}$$

  Indeed, as per Lem.4.19, any trace obtained from merging traces from i_1 has no conflict w.r.t. l iff it is obtained from merging traces from i_1 that all have no conflict with l. Therefore, traces that have no conflict w.r.t. l are exactly those that are obtained from merging traces with no conflicts

w.r.t. l. Those traces are those from $loop_k(i'_1)$ as per the induction hypothesis. Therefore the property holds.

□

5.2.3 Left inclusion

In this section, we prove that, for any interaction $i \in \mathbb{I}_\Omega$, we have $\sigma_o(i) \subset \sigma_d(i)$. A first Lemma (Lem.5.7) proposes a characterization of the execution relation "$\rightarrow$" w.r.t. σ_d.

Lemma 5.7: Characterization (left side) of $\rightarrow$ w.r.t. σ_d

For any action $a \in \mathbb{A}_\Omega$, for any trace $t \in \mathbb{T}_\Omega$ and for any interactions i and i' from $\mathbb{I}_\Omega$:

$$\Big((i \xrightarrow{a} i') \wedge (t \in \sigma_d(i')) \Big) \Rightarrow (a.t \in \sigma_d(i))$$

Proof. Let us consider i and i' in $\mathbb{I}_\Omega$ and a in $\mathbb{A}_\Omega$ and $t \in \mathbb{T}_\Omega$. Let us then suppose that $i \xrightarrow{a} i'$ and that $t \in \sigma_d(i')$. Let us then reason by induction on the cases that makes the hypothesis $i \xrightarrow{a} i'$ possible. There are 13 such cases, as per the 13 rules from Def.5.6:

1. when executing an atomic action, we have $i \in \mathbb{A}_\Omega$ and $i' = \varnothing$. Then $\sigma_d(i) = \{i\}$ and $\sigma_d(\varnothing) = \{\epsilon\}$. The property $i.\epsilon = i \in \sigma_d(i)$ holds.

2. when executing an action on the left of an alternative, we have i of the form $alt(i_1, i_2)$, and $i' = i'_1$ such that $i_1 \xrightarrow{a} i'_1$. By construction of σ_d, we have that $t \in \sigma_d(i'_1)$. By the induction hypothesis on the sub-interaction i_1, we have that $a.t \in \sigma_d(i_1)$. Given that $\sigma_d(i_1) \subset \sigma_d(i)$, the property holds.

3. when executing an action on the right of an alternative, we have i of the form $alt(i_1, i_2)$, and $i' = i'_2$ such that $i_2 \xrightarrow{a} i'_2$. By construction of σ_d, we have that $t \in \sigma_d(i'_2)$. By the induction hypothesis on the sub-interaction i_2, we have that $a.t \in \sigma_d(i_2)$. Given that $\sigma_d(i_2) \subset \sigma_d(i)$, the property holds.

4. when executing an action on the left of a par, we have i of the form $par(i_1, i_2)$, and $i' = par(i'_1, i_2)$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(par(i'_1, i_2))$. By definition of σ_d, we have that there exist $(t'_1, t_2) \in \sigma_d(i'_1) \times \sigma_d(i_2)$ s.t. $t \in (t'_1||t_2)$. Therefore we have $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t'_1 \in \sigma_d(i_1)$. Given that $\sigma_d(par(i_1, i_2))$ is the union of all the $(t_\alpha||t_\beta)$ with t_α and t_β traces from i_1 and i_2, we have that $(a.t'_1||t_2) \subset \sigma_d(i)$. In particular, we know that $t \in (t'_1||t_2)$, so, by definition of the $||$ operator, we have that $a.t \in (a.t'_1||t_2)$. Therefore the property holds.

5. when executing an action on the left of a par, we have i of the form $par(i_1, i_2)$, and $i' = par(i_1, i'_2)$ such that $i_2 \xrightarrow{a} i'_2$. We have that $t \in \sigma_d(par(i_1, i'_2))$. By definition of σ_d, we have that there exist $(t_1, t'_2) \in \sigma_d(i_1) \times \sigma_d(i'_2)$ s.t. $t \in (t_1||t'_2)$. Therefore we have $i_2 \xrightarrow{a} i'_2$ and $t'_2 \in \sigma_d(i'_2)$. Hence we

can apply the induction hypothesis on sub-interaction i_2, which implies that $a.t'_2 \in \sigma_d(i_2)$. Given that $\sigma_d(par(i_1, i_2))$ is the union of all the $(t_\alpha||t_\beta)$ with t_α and t_β traces from i_1 and i_2, we have that $(t_1||a.t'_2) \subset \sigma_d(i)$. In particular, we know that $t \in (t_1||a.t'_2)$, so, by definition of the $||$ operator, we have that $a.t \in (t_1||a.t'_2)$. Therefore the property holds.

6. when executing an action on the left of a *strict*, we have i of the form $strict(i_1, i_2)$, and $i' = strict(i'_1, i_2)$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(strict(i'_1, i_2))$. By definition of σ_d, we have that there exist $(t'_1, t_2) \in \sigma_d(i'_1) \times \sigma_d(i_2)$ s.t. $t \in (t'_1; t_2)$. Therefore we have $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t'_1 \in \sigma_d(i_1)$. Given that $\sigma_d(strict(i_1, i_2))$ is the union of all the $(t_\alpha; t_\beta)$ with t_α and t_β traces from i_1 and i_2, we have that $(a.t'_1; t_2) \subset \sigma_d(i)$. In particular, we know that $t \in (t'_1; t_2)$, so, by definition of the ; operator, we have that $a.t \in (a.t'_1; t_2)$. Therefore the property holds.

7. when executing an action on the right of a *strict*, we have i of the form $strict(i_1, i_2)$, and $i' = i'_2$ such that $i_2 \xrightarrow{a} i'_2$ with the added hypothesis that $i_1 \downarrow$. We have that $t \in \sigma_d(i'_2)$. Therefore we have $i_2 \xrightarrow{a} i'_2$ and $t \in \sigma_d(i'_2)$. Hence we can apply the induction hypothesis on sub-interaction i_2, which implies that $a.t \in \sigma_d(i_2)$. Given that $\sigma_d(strict(i_1, i_2))$ includes $\sigma_d(i_2)$ when $i_1 \downarrow$, and given that we know $i_1 \downarrow$ to be true, the property holds.

8. when executing an action on the left of a *seq*, we have i of the form $seq(i_1, i_2)$, and $i' = seq(i'_1, i_2)$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(seq(i'_1, i_2))$. By definition of σ_d, we have that there exist $(t'_1, t_2) \in \sigma_d(i'_1) \times \sigma_d(i_2)$ s.t. $t \in (t'_1;_\divideontimes t_2)$. Therefore we have $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t'_1 \in \sigma_d(i_1)$. Given that $\sigma_d(seq(i_1, i_2))$ is the union of all the $(t_\alpha;_\divideontimes t_\beta)$ with t_α and t_β traces from i_1 and i_2, we have that $(a.t'_1;_\divideontimes t_2) \subset \sigma_d(i)$. In particular, we know that $t \in (t'_1;_\divideontimes t_2)$, so, by definition of the $;_\divideontimes$ operator, we have that $a.t \in (a.t'_1;_\divideontimes t_2)$. Therefore the property holds.

9. when executing an action on the right of a *seq*, we have i of the form $seq(i_1, i_2)$, and $i' = seq(i'_1, i'_2)$ such that $i_1 \overset{\theta(a)}{\divideontimes\!\!\longrightarrow} i'_1$ and $i_2 \xrightarrow{a} i'_2$ with the added hypothesis that $i_1 \downarrow^\divideontimes \theta(a)$ implied by the fact that i_1 prunes into i'_1. We have that $t \in \sigma_d(i')$. Hence there exists $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i'_2)$ s.t. $t \in (t_1;_\divideontimes t_2)$.

   - We have $i_2 \xrightarrow{a} i'_2$ and $t_2 \in \sigma_d(i'_2)$. Hence we can apply the induction hypothesis on sub-interaction i_2, which implies that $a.t_2 \in \sigma_d(i_2)$.
   - the fact that $t_1 \in \sigma_d(i'_1)$ implies, as per Lem.5.6 that $\neg(t_1 \divideontimes \theta(a))$. As a result, by definition of the weak sequencing operator, $(t_1;_\divideontimes a.t_2)$ includes $a; (t_1;_\divideontimes t_2)$ and therefore $a.t$.
   - Given that $\sigma_d(i)$ includes all $(t_\alpha;_\divideontimes t_\beta)$ s.t. $t_\alpha \in \sigma_d(i_1)$ and $t_\beta \in \sigma_d(i_2)$, we have, in particular $(t_1;_\divideontimes a.t_2) \subset \sigma_d(i)$. Hence the property holds.

10. when executing an action underneath a $loop_S$, we have i of the form $loop_S(i_1)$ and $i' = strict(i'_1, loop_S(i_1))$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(i')$. Therefore there exists $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i)$ s.t. $t \in (t_1; t_2)$.

- We have $i_1 \xrightarrow{a} i'_1$ and $t_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t_1 \in \sigma_d(i_1)$.
- As a result, given that $t_2 \in \sigma_d(loop_S(i_1)) = \sigma_d(i_1)^{;*}$, and $a.t_1 \in \sigma_d(i_1)$, we have, $(a.t_1; t_2) \subset \sigma_d(i_1)^{;*}$ i.e. $(a.t_1; t_2) \subset \sigma_d(i)$
- Also, given that $t \in (t_1; t_2)$, we have immediately that $a.t \in (a.t_1; t_2)$ because it is always possible to add actions from the left.
- Therefore $a.t \in \sigma_d(i)$, so the property holds.

11. when executing an action underneath a $loop_H$, we have i of the form $loop_H(i_1)$ and $i' = seq(i'_1, loop_H(i_1))$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(i')$. Therefore there exists $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i)$ s.t. $t \in (t_1 ;_* t_2)$.
    - We have $i_1 \xrightarrow{a} i'_1$ and $t_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t_1 \in \sigma_d(i_1)$.
    - As a result, given that $t_2 \in \sigma_d(loop_H(i_1)) = \sigma_d(i_1)^{;^\neg_*}$, and $a.t_1 \in \sigma_d(i_1)$, we have, $(a.t_1 ;^\neg_* t_2) \subset \sigma_d(i_1)^{;^\neg *}$ i.e. $(a.t_1 ;^\neg_* t_2) \subset \sigma_d(i)$
    - Also, given that $t \in (t_1 ;_* t_2)$, we have immediately that $a.t \in (a.t_1 ;^\neg_* t_2)$ because it is always possible to add actions from the left and here we select a on the left.
    - Therefore $a.t \in \sigma_d(i)$, so the property holds.
12. when executing an action underneath a $loop_W$, we have i of the form $loop_W(i_1)$ and $i' = seq(i'_0, seq(i'_1, loop_W(i_1)))$ such that $i_1 \xrightarrow{a} i'_1$ and $i \overset{\theta(a)}{\ast\!\!\longrightarrow} i'_0$. We have that $t \in \sigma_d(i')$. Therefore there exists $t_0 \in \sigma_d(i'_0)$, $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i)$ s.t. $t \in (t_0 ;_* t_1 ;_* t_2)$.
    - Given that $i \overset{\theta(a)}{\ast\!\!\longrightarrow} i'_0$ we have that:
        - $\sigma_d(i'_0) \subset \sigma_d(i)$ and, given that $\sigma_d(i) = \sigma_d(i_1)^{;_**}$, is a weak Kleene closure we have that $\sigma_d(i) ;_* \sigma_d(i) \subset \sigma_d(i)$ and therefore $\sigma_d(i'_0) ;_* \sigma_d(i) \subset \sigma_d(i)$
        - and, given that $t_0 \in \sigma_d(i'_0)$, we have as per Lem.5.6 that $\neg(t_0 \ast\!\!\theta(a))$. Therefore, for any t_β and any $t_\alpha \in (t_0 ;_* t_\beta)$ we have $a.t_\alpha \in (t_0 ;_* a.t_\beta)$ given that we can take action a, which have no conflict w.r.t t_0, on the right side
    - We have $i_1 \xrightarrow{a} i'_1$ and $t_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t_1 \in \sigma_d(i_1)$
    - Given that $t_2 \in \sigma_d(loop_W(i_1)) = \sigma_d(i_1)^{;_**}$, and $a.t_1 \in \sigma_d(i_1)$, we have, $(a.t_1 ;_* t_2) \subset \sigma_d(i_1)^{;_**}$ i.e. $(a.t_1 ;_* t_2) \subset \sigma_d(i)$
    - Hence we have $(t_0 ;_* a.t_1 ;_* t_2) \subset (\sigma_d(i'_0) ;_* \sigma_d(i)) \subset \sigma_d(i)$
    - Finally, given that $a.t \in (t_0 ;_* a.t_1 ;_* t_2)$, the property holds.

13. when executing an action underneath a $loop_P$, we have i of the form $loop_P(i_1)$ and $i' = par(i'_1, loop_P(i_1))$ such that $i_1 \xrightarrow{a} i'_1$. We have that $t \in \sigma_d(i')$. Therefore there exists $t_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i)$ s.t. $t \in (t_1||t_2)$.

    - We have $i_1 \xrightarrow{a} i'_1$ and $t_1 \in \sigma_d(i'_1)$. Hence we can apply the induction hypothesis on sub-interaction i_1, which implies that $a.t_1 \in \sigma_d(i_1)$.
    - As a result, given that $t_2 \in \sigma_d(loop_P(i_1)) = \sigma_d(i_1)^{||*}$, and $a.t_1 \in \sigma_d(i_1)$, we have, $(a.t_1||t_2) \subset \sigma_d(i_1)^{||*}$ i.e. $(a.t_1||t_2) \subset \sigma_d(i)$
    - Also, given that $t \in (t_1||t_2)$, we have immediately that $a.t \in (a.t_1||t_2)$ because it is always possible to add actions from the left.
    - Therefore $a.t \in \sigma_d(i)$, so the property holds.

□

Thanks to the previous Lemma (Lem.5.7) as well as the characterization from Lem.5.4 we can conclude on the inclusion of the σ_o semantics into the σ_d semantics. Indeed, those two Lemmas state that the σ_d semantics accepts the same two construction rules (that for the empty trace ϵ and that for non empty traces of the form $a.t$) as those that define σ_o inductively. As a result any trace that might be accepted according to σ_o must also be accepted according to σ_d. However it does not imply the reciprocate (i.e. whether or not σ_d is included in σ_o). Indeed, it may be so that, if it were formulated using construction rules, σ_d would also verify some other construction rules in addition to the aforementioned two, which would allow the acceptation of some more traces.

Theorem 5.1: Inclusion of σ_o in σ_d

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_o(i) \subset \sigma_d(i)$$

Proof. Let us consider $i \in \mathbb{I}_\Omega$ and $t \in \sigma_o(i)$ and let us reason by induction on the trace t.

- If $t = \epsilon$, then, as per the definition of σ_o, this means that $i \downarrow$. Then as per Lem.5.4, this means that $\epsilon \in \sigma_d(i)$.
- If $t = a.t'$ then, by definition of σ_o, $a.t' \in \sigma_o(i)$ iff $\exists\ i' \in \mathbb{I}_\Omega$ s.t. $i \xrightarrow{a} i'$ and $t' \in \sigma_o(i')$. Let us therefore consider such an interaction i'. By the induction hypothesis on trace t', we have $(t' \in \sigma_o(i')) \Rightarrow (t' \in \sigma_d(i'))$. As a result we have $i \xrightarrow{a} i'$ and $t' \in \sigma_d(i')$. We can therefore apply Lem.5.7 to conclude that $a.t' \in \sigma_d(i)$. Hence the property holds.

□

In the next section we tackle the other inclusion that we have to prove i.e. that of σ_d into σ_o.

5.2.4 Right inclusion

In this section, we prove that, for any interaction $i \in \mathbb{I}_\Omega$, we have $\sigma_d(i) \subset \sigma_o(i)$. An intermediate Lemma, Lem.5.8, which is, in a certain manner, the reciprocate of Lem.5.7, is required at first.

Lemma 5.8: Characterization (right side) of $\rightarrow$ w.r.t. σ_d

For any action $a \in \mathbb{A}_\Omega$, for any trace $t \in \mathbb{T}_\Omega$ and for any interaction $i \in \mathbb{I}_\Omega$:

$$(a.t \in \sigma_d(i)) \Rightarrow \left(\exists\, i' \in \mathbb{I}_\Omega, \quad (i \xrightarrow{a} i') \wedge (t \in \sigma_d(i')) \right)$$

Proof. Let us consider $i \in \mathbb{I}_\Omega$, $a \in \mathbb{A}_\Omega$ and $t \in \mathbb{T}_\Omega$. Let us suppose that $a.t \in \sigma_d(i)$ and let us reason by induction on the term structure of i.

- we cannot have $i = \varnothing$ because it contradicts $a.t \in \sigma_d(i)$
- if $i \in \mathbb{A}_\Omega$ then $a.t \in \sigma_d(i)$ implies that $i = a$ and $t = \epsilon$. We then have the existence of $i' = \varnothing$ which indeed satisfies that $a \xrightarrow{a} \varnothing$ and $\epsilon \in \sigma_d(\varnothing)$
- if i is of the form $alt(i_1, i_2)$ then $a.t \in \sigma_d(i)$ implies either $a.t \in \sigma_d(i_1)$ or $a.t \in \sigma_d(i_2)$. Let us suppose it is the first case (the second is identical). Then, we can apply the induction hypothesis on sub-interaction i_1, which reveals the existence of i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $t \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $alt(i_1, i_2) \xrightarrow{a} i'_1$. As a result, we have identified $i' = i'_1$ which satisfies the property.
- if i is of the form $par(i_1, i_2)$ then $a.t \in \sigma_d(i)$ implies the existence of traces t_1 and t_2 such that $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ and $a.t \in (t_1||t_2)$. Then, as per Lem.4.2, this implies:
  - either that t_1 is of the form $a.t'_1$ and $t \in (t'_1||t_2)$
  - or t_2 is of the form $a.t'_2$ and $t \in (t_1||t'_2)$

  As both case can be treated identically, let us suppose it is the first case. Given that we have $a.t'_1 \in \sigma_d(i_1)$, we can apply the induction hypothesis on sub-interaction i_1, which reveals the existence of i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $par(i_1, i_2) \xrightarrow{a} par(i'_1, i_2)$. By definition of σ_d, given that $t'_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i_2)$, we have $(t'_1||t_2) \subset \sigma_d(par(i'_1, i_2))$. Then, given that $t \in (t'_1||t_2)$, this implies that $t \in \sigma_d(par(i'_1, i_2))$. We therefore have identified $i' = par(i'_1, i_2)$ which satisfies the property.
- if i is of the form $strict(i_1, i_2)$ then $a.t \in \sigma_d(i)$ implies the existence of traces t_1 and t_2 such that $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ and $a.t \in (t_1; t_2)$. Then, as per Lem.4.7, this implies:
  - either that t_1 is of the form $a.t'_1$ and $t \in (t'_1; t_2)$. In that case we can apply the induction hypothesis on sub-interaction i_1, which reveals the existence of i'_1 s.t. $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. By definition

of the execution relation "$\rightarrow$", this implies that $strict(i_1, i_2) \xrightarrow{a} strict(i'_1, i_2)$. By definition of σ_d, given that $t'_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i_2)$, we have $(t'_1; t_2) \subset \sigma_d(strict(i'_1, i_2))$. Then, given that $t \in (t'_1; t_2)$ this implies that $t \in \sigma_d(strict(i'_1, i_2))$. We therefore have identified $i' = strict(i'_1, i_2)$ which satisfies the property.

- or that $t_1 = \epsilon$ and $t_2 = a.t$. On the one hand, the fact that $t_1 = \epsilon \in \sigma_d(i_1)$ implies, as per Lem.5.4, that $i_1 \downarrow$. On the other hand, with $t_2 = a.t \in \sigma_d(i_2)$, we can apply the induction hypothesis on sub-interaction i_2, which reveals the existence of i'_2 s.t. $i_2 \xrightarrow{a} i'_2$ and $t \in \sigma_d(i'_2)$. By definition of the execution relation "$\rightarrow$", and because the precondition $i_1 \downarrow$ is verified, this implies that $strict(i_1, i_2) \xrightarrow{a} i'_2$. As a result, we have identified $i' = i'_2$ which satisfies the property.

- if i is of the form $seq(i_1, i_2)$ then $a.t \in \sigma_d(i)$ implies the existence of traces t_1 and t_2 such that $t_1 \in \sigma_d(i_1)$ and $t_2 \in \sigma_d(i_2)$ and $a.t \in (t_1;_{\divideontimes} t_2)$. Then, as per Lem.4.12, this implies:

  - either that $t_1 = a.t'_1$ and $t \in (t'_1;_{\divideontimes} t_2)$. In that case we can apply the induction hypothesis on sub-interaction i_1, which reveals the existence of i'_1 s.t. $i_1 \xrightarrow{a} i'_1$ and $t'_1 \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $seq(i_1, i_2) \xrightarrow{a} seq(i'_1, i_2)$. By definition of σ_d, given that $t'_1 \in \sigma_d(i'_1)$ and $t_2 \in \sigma_d(i_2)$, we have $(t'_1;_{\divideontimes} t_2) \subset \sigma_d(seq(i'_1, i_2))$. Then, given that $t \in (t'_1;_{\divideontimes} t_2)$, this implies that $t \in \sigma_d(seq(i'_1, i_2))$. We therefore have identified $i' = seq(i'_1, i_2)$ which satisfies the property.

  - or that $\neg(t_1 \divideontimes \theta(a))$ and that t_2 is of the form $a.t'_2$ with $t \in (t_1;_{\divideontimes} t'_2)$. On the one hand, the fact that $t_1 \in \sigma_d(i_1)$ is such that $\neg(t_1 \divideontimes \theta(a))$, ensures, as per Lem.5.5, that $i_1 \downarrow^{\divideontimes} \theta(a)$ and therefore, as per Lem.5.3, that there exists a unique i'_1 such that $i_1 \xrightarrow{\divideontimes \theta(a)} i'_1$. On the other hand, with $t_2 = a.t'_2 \in \sigma_d(i_2)$, we can apply the induction hypothesis on sub-interaction i_2, which reveals the existence of i'_2 s.t. $i_2 \xrightarrow{a} i'_2$ and $t'_2 \in \sigma_d(i'_2)$. By definition of the execution relation "$\rightarrow$", this implies that $seq(i_1, i_2) \xrightarrow{a} seq(i'_1, i'_2)$. Given that $t_1 \in \sigma_d(i_1)$ is such that $\neg(t_1 \divideontimes \theta(a))$, as per Lem.5.6, this implies that $t_1 \in \sigma_d(i'_1)$. By definition of σ_d, given that $t_1 \in \sigma_d(i'_1)$ and $t'_2 \in \sigma_d(i'_2)$, we have $(t_1;_{\divideontimes} t'_2) \subset \sigma_d(seq(i'_1, i'_2))$. Then, given that $t \in (t_1;_{\divideontimes} t'_2)$, this implies that $t \in \sigma_d(seq(i'_1, i'_2))$. We therefore have identified $i' = seq(i'_1, i'_2)$ which satisfies the property.

- if i is of the form $loop_S(i_1)$ then $a.t \in \sigma_d(i)$ means that $a.t \in \sigma_d(i_1)^{;*}$ and hence, as per Lem.4.25, this implies the existence of a trace t' such that $a.t' \in \sigma_d(i_1)$ and $t \in \{t'\}; \sigma_d(i_1)^{;*}$. Then, the fact that $a.t' \in \sigma_d(i_1)$ allows us to apply the induction hypothesis on sub-interaction i_1 to reveal the existence of i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $t' \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $loop_S(i_1) \xrightarrow{a} strict(i'_1, loop_S(i_1))$. By definition of σ_d, given that $t' \in \sigma_d(i'_1)$ and that $t \in \{t'\}; \sigma_d(i_1)^{;*}$, we have that $t \in \sigma_d(strict(i'_1, loop_S(i_1)))$. We therefore have identified $i' = strict(i'_1, loop_S(i_1))$ which satisfies the property.

- if i is of the form $loop_H(i_1)$ then $a.t \in \sigma_d(i)$ means that $a.t \in \sigma_d(i_1)^{;^{\curlywedge}_{\divideontimes}*}$. By definition this means that there exists a $j > 0$ such that $a.t \in \sigma_d(i_1)^{;^{\curlywedge}_{\divideontimes} j} = \sigma_d(i_1);^{\curlywedge}_{\divideontimes} \sigma_d(i_1)^{;^{\curlywedge}_{\divideontimes}(j-1)}$, and, given that $\sigma_d(i_1)^{;^{\curlywedge}_{\divideontimes}(j-1)} \subset$

$\sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*}$ we can identify a trace $t'' \in \sigma_d(i_1)$ such that $a.t \in \{t''\};^{\curvearrowleft}_{\divideontimes} \sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*}$. Because the restricted operator $;^{\curvearrowleft}_{\divideontimes}$ only allows to take the first action of recomposed traces from the left hand side, the fact that $a.t \in \{t''\};^{\curvearrowleft}_{\divideontimes} \sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*}$ implies that action a is taken from $\{t''\}$ and therefore t'' is of the form $a.t'$ and $t \in \{t'\};^{\curvearrowleft}_{\divideontimes} \sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*} \subset \{t'\};_{\divideontimes} \sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*}$. Then, the fact that $t'' = a.t' \in \sigma_d(i_1)$ allows us to apply the induction hypothesis on sub-interaction i_1 to reveal the existence of i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $t' \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $loop_H(i_1) \xrightarrow{a} seq(i'_1, loop_H(i_1))$. By definition of σ_d, given that $t' \in \sigma_d(i'_1)$ and that $t \in \{t'\};_{\divideontimes} \sigma_d(i_1)^{;^{\curvearrowleft}_{\divideontimes}*}$, we have that $t \in \sigma_d(seq(i'_1, loop_H(i_1)))$. We therefore have identified $i' = seq(i'_1, loop_H(i_1))$ which satisfies the property.

- if i is of the form $loop_W(i_1)$ then $a.t \in \sigma_d(i)$ means that $a.t \in \sigma_d(i_1)^{;_{\divideontimes}*}$. By definition this means that there exists a $n > 0$ such that $a.t \in \sigma_d(i_1)^{;_{\divideontimes}n} = \sigma_d(i_1);_{\divideontimes} \cdots ;_{\divideontimes} \sigma_d(i_1)$ (n times). As a result, we can identify traces t_1 through t_n such that for any $j \in [1, n]$, $t_j \in \sigma_d(i_1)$ and $a.t \in \{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\}$. Then, by definition of the weak sequencing operator, action a is taken from a certain t_j with $j \in [1, n]$ which is therefore of the form $t_j = a.t'_j$ and we have, for any $k < j$, $\neg(t_k \divideontimes \theta(a))$ (otherwise we could not take a from t_j and $t \in \{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_{j-1}\};_{\divideontimes} \{t'_j\};_{\divideontimes} \{t_{j+1}\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\}$. We can then remark the following:
  - considering i'_0 such that $loop_W(i_1) \xrightarrow{\theta(a)}_{\divideontimes} i'_0$ (which existence is guaranteed by Lem.5.3 given that a loop always evades any lifeline), because, for any $k < j$, we have $\neg(t_k \divideontimes \theta(a))$ then, as per Lem.5.6, for all $k < j$, we have $t_k \in \sigma_d(i'_0)$. Then:
    - If all the t_k are the empty trace then $\{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_{j-1}\} = \{\epsilon\} \subset \sigma_d(i'_0)$ ($\sigma_d(i'_0)$ contains at least ϵ because $loop_W(i_1)$ does)
    - If at least one t_k is not the empty trace then i'_0 is a non empty $loop_W$, and, because it is a $loop_W$, given that for all $k < j$, we have $t_k \in \sigma_d(i'_0)$ then $\{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_{j-1}\} \subset \sigma_d(i'_0)$ (because a $loop_W$ is closed under repetition by $;_{\divideontimes}$)

    In any case we have $\{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_{j-1}\} \subset \sigma_d(i'_0)$ and therefore
    $t \in \{t_1\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_{j-1}\};_{\divideontimes} \{t'_j\};_{\divideontimes} \{t_{j+1}\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\} \subset \sigma_d(i'_0);_{\divideontimes} \{t'_j\};_{\divideontimes} \{t_{j+1}\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\}$
  - given that, for any $k > j$ we have that $t_k \in \sigma_d(i)$ and because i is a loop ($i = loop_W(i_1)$), then $\{t_{j+1}\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\} \subset \sigma_d(i)$ and therefore $t \in \sigma_d(i'_0);_{\divideontimes} \{t'_j\};_{\divideontimes} \{t_{j+1}\};_{\divideontimes} \cdots ;_{\divideontimes} \{t_n\} \subset \sigma_d(i'_0);_{\divideontimes} \{t'_j\};_{\divideontimes} \sigma_d(i)$
  - given that $t_j = a.t'_j \in \sigma_d(i_1)$ we can apply the induction hypothesis on sub-interaction i_1 to reveal the existence of i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $t'_j \in \sigma_d(i'_1)$. By definition of the execution relation "$\rightarrow$", this implies that $loop_W(i_1) \xrightarrow{a} seq(i'_0, seq(i'_1, loop_W(i_1)))$
  - finally, given that we have shown that $t \in \sigma_d(i'_0);_{\divideontimes} \{t'_j\};_{\divideontimes} \sigma_d(i)$ and because $t'_j \in \sigma_d(i'_1)$, by definition of σ_d we can conclude that $t \in \sigma_d(seq(i'_0, seq(i'_1, loop_W(i_1))))$. Therefore we have identified $i' = seq(i'_0, seq(i'_1, loop_W(i_1)))$ which satisfies the property.

□

Thanks to the Lem.5.8 as well as the characterization from Lem.5.4 we can conclude on the inclusion of the σ_d semantics into the σ_o semantics.

Theorem 5.2: Inclusion of σ_d in σ_o

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_o(i) \supset \sigma_d(i)$$

Proof. Let us consider $i \in \mathbb{I}_\Omega$ and $t \in \sigma_d(i)$ and let us reason by induction on the trace t.

- If $t = \epsilon$, the fact that $t = \epsilon \in \sigma_d(i)$ implies, as per Lem.5.4, that $i \downarrow$. Then, by definition of σ_o, this means that $\epsilon \in \sigma_o(i)$.
- If $t = a.t'$ then, the fact that $a.t' \in \sigma_d(i)$ implies, as per Lem.5.8, that there exists an interaction i' such that $i \xrightarrow{a} i'$ and $t' \in \sigma_d(i')$. Let us therefore consider such an interaction i'. By the induction hypothesis on trace t', we have $(t' \in \sigma_d(i')) \Rightarrow (t' \in \sigma_o(i'))$. As a result we have $i \xrightarrow{a} i'$ and $t' \in \sigma_o(i')$. By definition of the operational semantics, this implies that $a.t' \in \sigma_o(i)$. Hence the property holds.

□

We have finally proven both inclusion and we conclude with Th.5.3 that the operational semantics σ_o that we have defined in Sec.5.1 is indeed equivalent to the fundational denotational-style semantics σ_d from Chap.4.

Theorem 5.3: Equivalence of the σ_d and σ_o semantics

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_o(i) = \sigma_d(i)$$

Proof. Immediately implies by Th.5.1 and Th.5.2. □

Hence we have proven the correctness of our operational semantics w.r.t. the denotational-style semantics (which acts as a mathematical foundation).

Conclusion

In this chapter we have defined a structural operational semantics for our Interaction Language (IL). This semantics is a small-step operational semantics in so far as it defines accepted traces $t \in \sigma_o(i_1)$ by unveiling sequences of small-steps $i_1 \xrightarrow{a_1} i_2 \xrightarrow{a_2} i_3 \cdots$ corresponding to the occurences of atomic communication events and such that the accepted trace t is a concatenation of the actions a_j that are successively expressed. The semantics is also structural given that its definition, in the form of a set of rules, is inductive on the structure of interaction terms. We have then proven the equivalence of this operational semantics with regards to the denotational semantics from Chap.4 which acts as a reference and mathematical foundation. By equivalence, we mean that for any interaction, its set of accepted traces is the same according to both semantics.

In the next chapter, we propose an algorithmization of this operational semantics to facilitate its implementation and use in Formal Verification (FV).

Chapter 6

Some execution semantics

Contents

The purpose of this chapter is to define an "algorithmicized" version of the operational semantics from the previous chapter that is expressed in functional style i.e. in the style of functional programming languages. We name "execution semantics" this "algorithmicized" operational semantics. Additionally, we define variants of this execution semantics which incorporate simplifications, as allowed by the $\approx_{E_{\mathbf{I}}}$ relation from Chap.4.

The plan of this chapter is as follows:

- in Sec.6.1, we present the initial execution semantics σ_e,
- in Sec.6.2, we prove that this new semantics σ_e is in facts equivalent to σ_o i.e. that for any interaction i, we have $\sigma_e(i) = \sigma_o(i)$,
- in Sec.6.3, we use results from equational theory and term rewriting seen in Chap.4 to define variants of σ_e which include term simplification.

Let us remark that the contents of Sec.6.1 and Sec.6.2 i.e. the formalization of the execution semantics and the proof of its equivalence w.r.t. the previous semantics have been encoded in Coq in [88].

6.1 Definition of the execution semantics

In this section we redefine, in functional style, the operational semantics σ_o. However we add a twist in that re-definition by separating concerns between, on the one side, the determination of which actions are immediately executable, and, on the other side, the determination or computation of follow-up interactions.

This section is organized as follows:

- in Sec.6.1.1 we redefine the predicates of termination, evasion and the pruning relation in functional style.
- in Sec.6.1.2 we present a first algorithmization of the execution relation and explain its limitations which leads to proposing a second algorithmization which relies on a separation of concerns as previously mentioned.
- in Sec.6.1.3 we define the frontier of execution, which computes the positions of all immediately executable actions.
- in Sec.6.1.4 we define the execution function, which computes the unique follow-up of the execution of an action at a specific position.
- finally, in Sec.6.1.5, we define the execution semantics.

6.1.1 Termination, evasion & pruning in functional style

In this section we redefine $\downarrow$, $\downarrow^{\divideontimes}$ and $\overset{\divideontimes}{\longmapsto}$ in functional style. We use a style of pseudo-code with Pattern-Matching inspired from Pattern-Matching notations found in functional programming languages such as ML

(OCaml, etc.), Rust or Haskell. The use of this style underlines the ease of implementation of the various definitions proposed in this chapter.

Definition 6.1: Termination in functional style

$\texttt{trm} : \mathbb{I}_\Omega \to \mathit{bool}$ is the function s.t. for any $i \in \mathbb{I}_\Omega$:

- $\texttt{trm}(i)$ **match** i **with**

$\mid \varnothing$	$\to$	$\top$
$\mid a \in \mathbb{A}_\Omega$	$\to$	$\bot$
$\mid f(i_1, i_2)$	$\to$	$\texttt{trm}(i_1) \ \wedge \ \texttt{trm}(i_2)$ **for** $f \in \{\mathit{strict}, \mathit{seq}, \mathit{par}\}$
$\mid \mathit{alt}(i_1, i_2)$	$\to$	$\texttt{trm}(i_1) \ \vee \ \texttt{trm}(i_2)$
$\mid \mathit{loop}_k(i_1)$	$\to$	$\top$ **for** $k \in \{S, H, W, P\}$

Lemma 6.1: Correctness of "trm" w.r.t. "$\downarrow$"

For any $i \in \mathbb{I}_\Omega$, $(\texttt{trm}(i) = \top) \Leftrightarrow (i \downarrow)$

Proof. By induction on the term structure of interactions.

□

Definition 6.2: Evasion in functional style

$\texttt{vad} : \mathbb{I}_\Omega \times L \to \mathit{bool}$ is the function s.t. for any $(i, l) \in \mathbb{I}_\Omega \times L$:

- $\texttt{vad}(i, l) =$ **match** i **with**

$\mid \varnothing$	$\to$	$\top$
$\mid a \in \mathbb{A}_\Omega$	$\to$	$(\theta(a) \neq l)$
$\mid f(i_1, i_2)$	$\to$	$\texttt{vad}(i_1, l) \ \wedge \ \texttt{vad}(i_2, l)$ **for** $f \in \{\mathit{strict}, \mathit{seq}, \mathit{par}\}$
$\mid \mathit{alt}(i_1, i_2)$	$\to$	$\texttt{vad}(i_1, l) \ \vee \ \texttt{vad}(i_2, l)$
$\mid \mathit{loop}_k(i_1)$	$\to$	$\top$ **for** $k \in \{S, H, W, P\}$

Lemma 6.2: Correctness of "vad" w.r.t. "$\downarrow^{\divideontimes}$"

For any $i \in \mathbb{I}_\Omega$ and any $l \in L$, $(\texttt{vad}(i, l) = \top) \Leftrightarrow (i \downarrow^{\divideontimes} l)$

Proof. By induction on the term structure of interactions. □

Definition 6.3: Pruning in functional style

$\texttt{prn} : \mathbb{I}_\Omega \times L \to \mathbb{I}_\Omega$ is s.t. for any $(i,l) \in \mathbb{I}_\Omega \times L$ verifying $\downarrow^{\divideontimes} (i,l)$:

• **provided** $\texttt{vad}(i,l) = \top$ **then** $\texttt{prn}(i,l) =$ **match** i **with**

$$
\begin{array}{lcl}
\mid \varnothing & \to & \varnothing \\
\mid a \in \mathbb{A}_\Omega & \to & a \\
\mid alt(i_1,i_2) & \to & \begin{cases}
\texttt{prn}(i_2,l) & \textbf{if } (\texttt{vad}(i_1,l) = \bot) \ \wedge \ (\texttt{vad}(i_2,l) = \top) \\
\texttt{prn}(i_1,l) & \textbf{if } (\texttt{vad}(i_1,l) = \top) \ \wedge \ \neg(\texttt{vad}(i_2,l) = \bot) \\
alt(\texttt{prn}(i_1,l), \texttt{prn}(i_2,l)) & \textbf{if } (\texttt{vad}(i_1,l) = \bot) \ \wedge \ (\texttt{vad}(i_2,l) = \top)
\end{cases} \\
\mid f(i_1,i_2) & \to & f(\texttt{prn}(i_1,l), \texttt{prn}(i_2,l)) \textbf{ for } f \in \{strict, seq, par\} \\
\mid loop_k(i_1) & \to & \left| \begin{cases}
loop_k(\texttt{prn}(i_1,l)) & \textbf{if } (\texttt{vad}(i_1,l) = \bot) \\
\varnothing & \textbf{else}
\end{cases} \right| \textbf{ for } k \in \{S,H,W,P\}
\end{array}
$$

Lemma 6.3: Correctness of "prn" w.r.t. "$\overset{\divideontimes}{\longrightarrow}$"

For any $i \in \mathbb{I}_\Omega$ and any $l \in L$:

$$(\exists\, i' \in \mathbb{I}_\Omega,\ s.t.\ i \overset{l}{\divideontimes\!\!\to} i') \Rightarrow (\texttt{prn}(i,l) = i') \qquad \text{and} \qquad (i \downarrow^{\divideontimes} l) \Rightarrow (i \overset{l}{\divideontimes\!\!\to} \texttt{prn}(i,l))$$

Proof. By induction on the term structure of interactions and using Lem.6.2 and Lem.5.3. □

Given Lem.6.1 and Lem.6.2 we will, in the remainder of this chapter, use the notations $\downarrow$ and $\downarrow^{\divideontimes}$ instead of that of their functional style equivalent.

6.1.2 An argument for separating concerns when determining follow-ups

With the operational-style semantics from Chap.5, the determination of which are the actions that can be executed within an interaction term, and the computation of the new interactions that remain after the executions of those actions are intertwined. Indeed, any predicate $i \xrightarrow{a} i'$ must be inferred inductively without prior knowledge of which actions a can be executed. As a result, so as to know which are the possible follow-ups to a given interaction i, one has to infer all the possible predicates of the form $i \xrightarrow{a} i'$.

This can be done for instance with the algorithm given in pseudo-code on Def.6.4. This "nxt" function exploits the small-step operational semantics to infer, given an interaction i, which are the follow-up interactions resulting from the execution of actions in i.

Definition 6.4: Computing all follow-ups with "nxt"

$\texttt{nxt} : \mathbb{I}_\Omega \to (\mathbb{A}_\Omega \times \mathbb{I}_\Omega)^*$ is s.t. for any $i \in \mathbb{I}_\Omega$:

- $\texttt{nxt}(i) \;=\; \textbf{match } i \textbf{ with}$

$$
\begin{array}{lll}
\mid \varnothing & \to & [\,] \\
\mid a \in \mathbb{A}_\Omega & \to & [(a, \varnothing)] \\
\mid strict(i_1, i_2) & \to & \left\{ \begin{array}{ll} [(a, strict(i_1', i_2)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] & \textbf{if } (\texttt{vad}(i_1, l) = \bot) \\ [(a, strict(i_1', i_2)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] + \texttt{nxt}(i_2) & \textbf{if } (\texttt{vad}(i_1, l) = \top) \end{array} \right. \\
\mid seq(i_1, i_2) & \to & \left(\begin{array}{l} [(a, seq(i_1', i_2)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] \\ +[(a, seq(\texttt{prn}(i_1, \theta(a)), i_2')) \textbf{ for } (a, i_2') \in \texttt{nxt}(i_2) \textbf{ if } (\texttt{vad}(i_1, \theta(a)) = \top)] \end{array} \right) \\
\mid par(i_1, i_2) & \to & \left(\begin{array}{l} [(a, par(i_1', i_2)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] \\ +[(a, par(i_1, i_2')) \textbf{ for } (a, i_2') \in \texttt{nxt}(i_2)] \end{array} \right) \\
\mid alt(i_1, i_2) & \to & \texttt{nxt}(i_1) + \texttt{nxt}(i_2) \\
\mid loop_S(i_1) & \to & [(a, strict(i_1', i)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] \\
\mid loop_H(i_1) & \to & [(a, seq(i_1', i)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] \\
\mid loop_W(i_1) & \to & [(a, seq(\texttt{prn}(i, \theta(a)), seq(i_1', i))) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)] \\
\mid loop_P(i_1) & \to & [(a, par(i_1', i)) \textbf{ for } (a, i_1') \in \texttt{nxt}(i_1)]
\end{array}
$$

With the "nxt" algorithm from Def.6.4, we exploit the tree-like structure of interaction terms and the inductive definition of the semantics to return, for a given interaction i, all its follow-ups as a list of tuples (a, i') such that the predicate $i \xrightarrow{a} i'$ holds true. The algorithm takes the form of a recursive function $\texttt{nxt}$ which:

- if applied to a base case interaction, immediately returns all its follow-ups. In details this means that:
  - for the empty interaction $\varnothing$, given that there are no follow-ups, $\texttt{nxt}(\varnothing)$ returns an empty list
  - for atomic actions $a \in \mathbb{A}_\Omega$, given that there is a single possible follow-up, $\texttt{nxt}(a)$ returns a list $[(a, \varnothing)]$ with a single element such that $a \xrightarrow{a} \varnothing$
- if applied to a more complex interaction, $\texttt{nxt}$ is called recursively on the children of the interaction term (i.e. its immediate sub-interactions). Then, follow-up interactions are reconstructed, as specified by the rules of the operational semantics from Def.5.6, from the follow-ups of the sub-interactions that have been computed.

However, we argue that this manner of algorithmizing the operational semantics is not ideal, especially when one consider applications for some Formal Verification (FV) techniques. Indeed we do not necessarily need to systematically compute all follow-up interactions. To avoid those unnecessary computations, we propose to separate the process into two parts:

- the computation of a "frontier of execution", which enumerates all the actions that are immediately executable
- the computation of specific follow-up interactions, obtained after the execution of a specific action from the frontier of execution

With this method, we can specifically choose to execute a precise action so as to obtain a specific follow-up interaction. In the operational semantics, the small-steps are of the form $i \xrightarrow{a} i'$. However, given that there can be several occurrences of the same action a within the interaction i, there can exist several such executions. For instance we can have two distinct interactions i'_A and i'_B such that $i \xrightarrow{a} i'_A$ and $i \xrightarrow{a} i'_B$. Given that the execution semantics aims at allowing the execution of specific actions, we need a manner to unambiguously identify those actions within an interaction. This manner of doing so is positions, which we have introduced in Chap.4.

As explained earlier, the execution semantics is an approach that is equivalent to that of the operational semantics but in which we decorrelate the identification of the actions that are immediately executable from the computation of the corresponding follow-up interactions. This approach is described schematically on Fig.6.1. It consists:

- at first in identifying all the positions of the actions found within the interaction term i that can be immediately executed. The set of those positions is the frontier of interaction i.
- then, for any one of those actions a that is found at a specific position $p \in pos(i)$ i.e. such that $a = i_{|p}$, we can execute it within i and compute a uniquely defined follow-up interaction i'

Figure 6.1: Description of the principles of the execution semantics

6.1.3 Frontier of execution

Among all the actions that can be found at the leaf nodes of an interaction term, only some of them are immediately executable. We name this subset of leaf nodes the frontier of execution of the interaction. Given an interaction $i \in \mathbb{I}_\Omega$, we note $\texttt{frt}(i)$ its frontier of execution. Each node being uniquely identified by its

position within i, we use those positions when defining "$\texttt{frt}$" so that $\texttt{frt}(i) \subseteq pos(i)$ (with $pos(i)$ the set of positions in the term i). Let us indeed recall that there can be several instances of a same action at different positions, and several of those can be present in a frontier of execution at the same time. Actions $a = i_{|p}$ corresponding to frontier positions $p \in \texttt{frt}(i)$ are called frontier actions. The "$\texttt{frt}$" function is formally defined in Def.6.5.

Definition 6.5: Frontier of execution

$\texttt{frt} : \mathbb{I}_\Omega \to \mathcal{P}(\{1,2\}^*)$ is the function s.t. for any $i \in \mathbb{I}_\Omega$:

- $\texttt{frt}(i) =$ **match** i **with**

$$
\begin{array}{lcl}
\mid \varnothing & \to & \emptyset \\
\mid a \in \mathbb{A}_\Omega & \to & \{\epsilon\} \\
\mid \mathit{strict}(i_1, i_2) & \to & \begin{cases} 1.\texttt{frt}(i_1) \cup 2.\texttt{frt}(i_2) & \textbf{if } i_1 \downarrow \\ 1.\texttt{frt}(i_1) & \textbf{else} \end{cases} \\
\mid \mathit{seq}(i_1, i_2) & \to & 1.\texttt{frt}(i_1) \cup \{p \mid p \in 2.\texttt{frt}(i_2)\ ,\ i_1 \downarrow^{\divideontimes} \theta(i_{|p})\} \\
\mid f(i_1, i_2) & \to & 1.\texttt{frt}(i_1) \cup 2.\texttt{frt}(i_2) \textbf{ for } f \in \{\mathit{alt}, \mathit{par}\} \\
\mid \mathit{loop}_k(i_1) & \to & 1.\texttt{frt}(i_1) \textbf{ for } k \in \{S, H, W, P\}
\end{array}
$$

The frontier of an interaction can be statically inferred from its term structure. The "$\texttt{frt}$" function is defined inductively such that:

- the empty interaction has an empty frontier: $\texttt{frt}(\varnothing) = \emptyset$
- for any interaction that consists in a single atomic action $a \in \mathbb{A}_\Omega$, $\texttt{frt}(a) = \{\epsilon\}$, because the position ϵ, which designates the root node of the interaction then designates the action leaf node itself (ϵ is the position of a which is immediately executable)
- for any interaction i of the form $f(i_1, i_2)$, with f any of the four binary constructors, $\texttt{frt}(i)$ is inferred from $\texttt{frt}(i_1)$ and $\texttt{frt}(i_2)$.
  - In all cases, actions that are immediately executable within the left sub-interaction i_1 are also immediately executable in i. Indeed, the term being read from left to right, all constructors, if they introduce ordering constraints, will only do so on the right sub-interaction i_2. Thus $1.\texttt{frt}(i_1)$ is included in $\texttt{frt}(i)$.
  - As for the actions that are immediately executable within the right sub-interaction i_2, whether or not they are also immediately executable in i depends on the nature of the constructor f and that of the left sibling interaction i_1.
    - If $f = \mathit{alt}$ or $f = \mathit{par}$, $2.\texttt{frt}(i_2)$ is also included in $\texttt{frt}(i)$ because no constraint may prevent the execution of actions from i_2.

* If $f = strict$, any action from i_2 can only be executed in i if no action from i_1 is executed in the same execution (otherwise it would violate the strict sequencing). Therefore $2.\texttt{frt}(i_2)$ is included in $\texttt{frt}(i)$ iff i_1 accepts the empty trace i.e. iff $i_1 \downarrow$.
* If $f = seq$, elements p from $2.\texttt{frt}(i_2)$ are included in $\texttt{frt}(i)$ iff i_1 accepts an execution that does not involve the lifeline on which the action $i_{|p}$ occurs i.e. iff $i_1 \downarrow^{\not} \theta(i_{|p})$.

- for any interaction i of the form $loop_k(i_1)$, with $k \in \{S, H, W, P\}$, we have $\texttt{frt}(i) = 1.\texttt{frt}(i_1)$ because the loop can be instantiated through the execution of any one of the immediately executable actions from its sub-interaction, which specifies the behaviors that can be repeated.

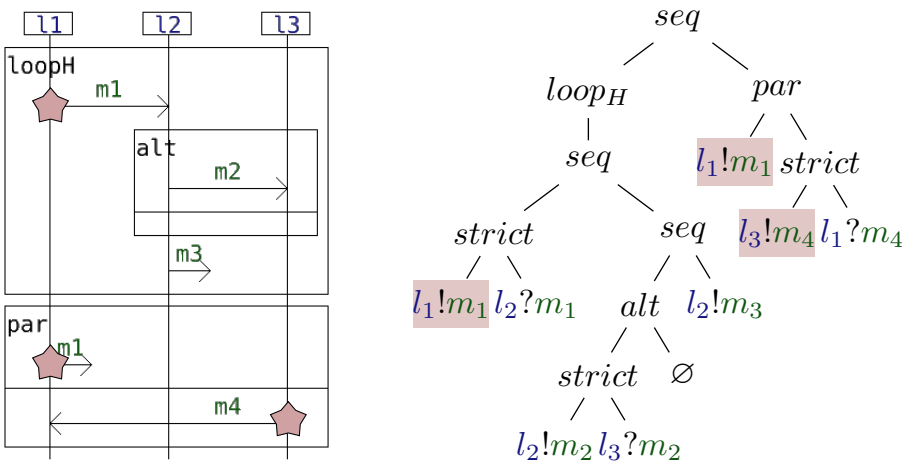


Figure 6.2: Frontier actions highlighted on an example interaction

Fig.6.2 illustrates the definition of "frt" on an example interaction. In this interaction, we have 8 different actions on leaves therefore $\texttt{frt}(i) \subseteq \{1111, 1112, 112111, 112112, 1122, 21, 221, 222\}$. Let us now proceed by elimination to determine which ones are in $\texttt{frt}(i)$. Actions on the right of every *strict* operators are prevented from being executed by those on their left and as such are not in the frontier. This eliminates $\{1112, 112112, 222\}$. $l_2!m_2$ and $l_2!m_3$ are prevented from being executed by $l_2?m_1$ which is a cousin on their left w.r.t the *seq* operator at position 11. This eliminates $\{1112, 1122\}$. Then, by elimination, $\texttt{frt}(i) = \{1111, 21, 221\}$.

Another manner to describe the frontier of the example interaction from Fig.6.2 is to say that it is composed of three distinct observable actions and we can explain why they are immediately executable as follows:

- both of the instances of the emission of message m_1 by lifeline l_1 can immediately occur.
  - The one within the loop, at position 1111, is at the top of the diagram and therefore nothing prevents it from being executed.
  - The other one, at position 21 can be immediately executed because we can choose not to instantiate the loop. If the loop is repeated zero times, then nothing prevents the emission of that message.

- the emission of message m_4 by lifeline l_3, at position 221 can immediately occur. Indeed, we can, for instance, choose not to instantiate the loop. Then, nothing prevents the emission of that message.

6.1.4 Execution function

With "`frt`", we have isolated the responsibility of determining which actions are executable, and proposed an implementation of it. Now remains the ability to reconstruct efficiently "follow-up" interactions issued from the execution of said actions. As explained earlier and in Chap.5, the follow-up interaction specifies all the continuations of the executions of the original interaction that start with the execution of a specific action.

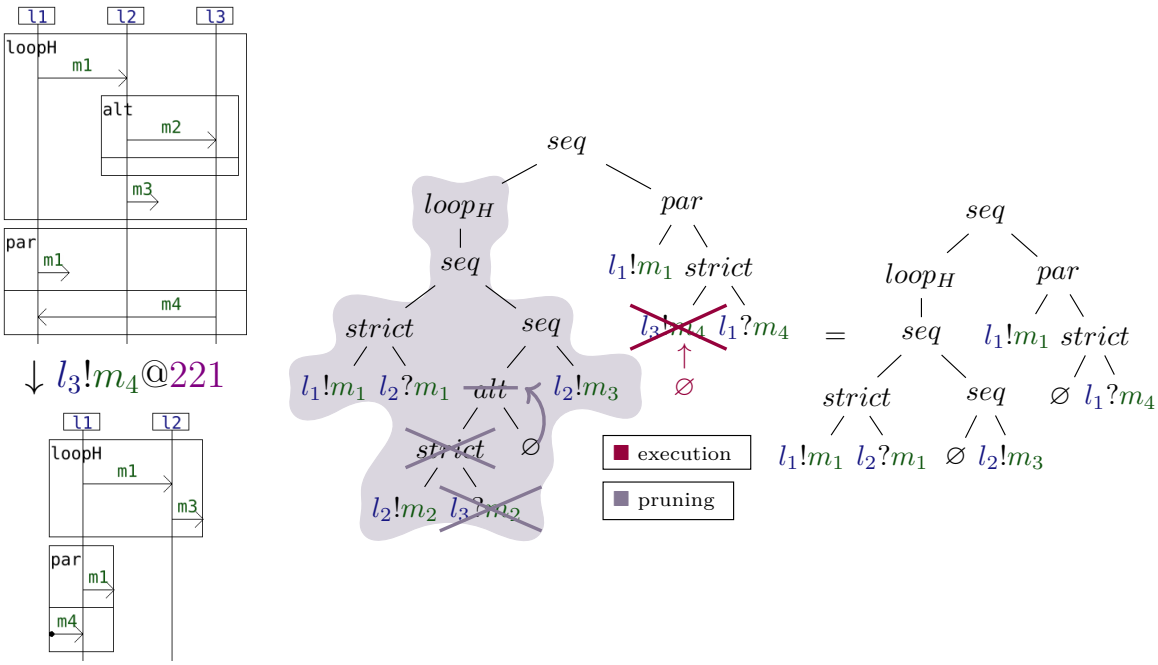


Figure 6.3: Illustration of the execution of an action at a specific position

Let us start by an example. On Fig.6.3 is reproduced the example from Fig.6.2. Let us remember that this example interaction has three frontier actions (as shown on Fig.6.2). Notably, the action $l_3!m_4$ at position 221 is in the frontier. Let us then execute that specific action, at position 221. The follow-up interaction can be computed by rewriting the original interaction via a process that is described graphically on the right of Fig.6.3:

- In this example, executing action $l_3!m_4$ at position 221 simply consists in deleting the corresponding node from the interaction so that it may not be executed again. This is represented on Fig.6.3 by crossing-out in red the node at position 221. In practice, this consists in replacing $l_3!m_4$ at position 221 by the empty interaction $\varnothing$.

- However this replacement of the node at position 221 is not sufficient to obtain the follow-up interaction. As we have seen in Chap.5, we may need to clean-up the interaction term via the process of pruning. Pruning is applied to all the left cousins of the node at position 221 (the executed action) if their

common ancestor is a *seq* constructor. Here, the root node is a *seq*. Therefore we must replace $i_{|1}$ by $\texttt{prn}(i_{|1}, \theta(i_{|p}))$. As explained in Chap.5, the process of pruning removes from the interaction model any contradiction that could occur. In this example, if l_3 emits m_4 then it forfeits its ability to receive m_2 because otherwise, this would contradict the top to bottom order of the diagram which corresponds to a weak sequencing between the reception of m_2 and the emission of m_4 on l_3. If it were to occur, the reception of m_2 should precede the emission of m_4. As a result it cannot occur at all if m_4 is emitted at first. Let us recall that the pruning cleans up the interaction term so as to minimally remove those contradictions. A naive clean-up would be to simply remove the loop. However it is not necessary to do so because the reception of m_2 is contained within an alternative, and, in that alternative, the empty branch can be chosen to solve the contradiction. Operating this minimal pruning, which corresponds to forcing the choice of the alternative still allows the repetition of the emission of m_1 by l_1 and so on. If we were to choose the native pruning, by removing the loop in its entirety we would obtain an under-approximated follow-up interaction because it would be an interaction that do not specifies all but only some of the continuations of the executions starting by the emission of m_4 at position 221.

The process we have described allows one to rewrite the original interaction term into the specific follow-up interaction that corresponds to the execution of an action at a specific position. The transformation that is illustrated on Fig.6.3 may be denoted by $i \xrightarrow{l_3!m_4@221} i'$, which signifies that the execution of action $l_3!m_4$ at position 221 within the interaction i results in i' as a follow-up interaction.

We can generalize this notation such that for any interaction i, the fact that an action a at position p can be executed which yields to the follow-up interaction i' can be denoted by $i \xrightarrow{a@p} i'$.

The formalisation of this process (for the computation of the follow-up interaction) takes the form of an inductive function `exe` that we define in Def.6.6. This function takes as arguments an interaction i and one of its frontier position $p \in front(i)$ and returns an i' that characterizes all the continuations of the executions of i that begin with the execution of the action $i_{|p}$ that is at the specific position p within i. `exe` in effects isolates and implements the "small-step" from our operational semantics.

Definition 6.6: Interaction Execution

$\texttt{exe} : \mathbb{I}_\Omega \times \{1,2\}^* \to \mathbb{I}_\Omega$ is the function s.t. for any $i \in \mathbb{I}_\Omega$ and $p \in \texttt{frt}(i)$:

- $\texttt{exe}(i,p) =$ **match** (i,p) **with**

$\mid (act, \epsilon)$	$\to$	$\varnothing$
$\mid (alt(i_1,i_2), 1.p_1)$	$\to$	$\texttt{exe}(i_1,p_1)$
$\mid (alt(i_1,i_2), 2.p_2)$	$\to$	$\texttt{exe}(i_2,p_2)$
$\mid (strict(i_1,i_2), 1.p_1)$	$\to$	$strict(\texttt{exe}(i_1,p_1), i_2)$
$\mid (strict(i_1,i_2), 2.p_2)$	$\to$	$\texttt{exe}(i_2,p_2)$
$\mid (seq(i_1,i_2), 1.p_1)$	$\to$	$seq(\texttt{exe}(i_1,p_1), i_2)$
$\mid (seq(i_1,i_2), 2.p_2)$	$\to$	$seq(\texttt{prn}(i_1, \theta(i_{2\mid p_2})), \texttt{exe}(i_2,p_2))$
$\mid (par(i_1,i_2), 1.p_1)$	$\to$	$par(\texttt{exe}(i_1,p_1), i_2)$
$\mid (par(i_1,i_2), 2.p_2)$	$\to$	$par(i_1, \texttt{exe}(i_2,p_2))$
$\mid (loop_S(i_1), 1.p_1)$	$\to$	$strict(\texttt{exe}(i_1,p_1), i)$
$\mid (loop_H(i_1), 1.p_1)$	$\to$	$seq(\texttt{exe}(i_1,p_1), i)$
$\mid (loop_W(i_1), 1.p_1)$	$\to$	$seq(\texttt{prn}(i, \theta(i_{1\mid p_1})), seq(\texttt{exe}(i_1,p_1), i))$
$\mid (loop_P(i_1), 1.p_1)$	$\to$	$par(\texttt{exe}(i_1,p_1), i)$

$\texttt{exe}$ is defined inductively on both the structure of the interaction i and the position $p = d_1...d_n \in \{1,2\}^n$. The execution of $\texttt{exe}(i,p)$ traverses recursively the syntactic structure of i guided by the path defined by the position p, that is, from $\texttt{exe}(i_{\mid\epsilon}, d_1...d_n)$ (root node), ..., up to $\texttt{exe}(i_{\mid p}, \epsilon)$ (target action leaf to execute).

- $\texttt{exe}(i_{\mid p}, \epsilon) = \varnothing$ constitutes the stopping criterion and i' is then constructed when the algorithm goes back up through the syntactic structure of i. Assigning $\varnothing$ to $\texttt{exe}(i_{\mid p}, \epsilon)$ ensures that the action $i_{\mid p}$ is removed in the construction of i'.

- When a *par* node is encountered during the upward traversal, i.e. for $j \in [1,n]$, $i_{\mid d_1...d_j} = par(i_{\mid d_1...d_j.1}, i_{\mid d_1...d_j.2})$ then $\texttt{exe}(i_{\mid d_1...d_j}, d_{j+1}...d_n)$ is simply:

  - either $par(\texttt{exe}(i_{\mid d_1...d_j.1}, d_{j+2}...d_n), i_{\mid d_1...d_j.2})$ if $d_{j+1} = 1$
  - or $par(i_{\mid d_1...d_j.1}, \texttt{exe}(i_{\mid d_1...d_j.2}, d_{j+2}...d_n))$ if $d_{j+1} = 2$

  Indeed, as *par* specifies parallel executions, there is no need for pruning.

- When an *alt* node is reached, using the same notations, we would have:

  - $\texttt{exe}(i_{\mid d_1...d_j}, d_{j+1}...d_n) = \texttt{exe}(i_{\mid d_1...d_{j+1}}, d_{j+2}...d_n)$

  Indeed, we can 'skip' the *alt* node itself and replace it directly with the interaction resulting from the execution of the chosen branch.

- When a *loop* is reached, i.e. $i_{\mid d_1...d_j} = loop_f(i_{\mid d_1...d_j.1})$ (with a mandatory $d_{j+1} = 1$), we have:

  - $\texttt{exe}(i_{|d_1...d_j}, d_{j+1}...d_n) = f(\texttt{exe}(i_{|d_1...d_{j+1}}, d_{j+2}...d_n), i_{|d_1...d_j})$

  Indeed, the execution is done on a copy of the loop content that precedes (with f operator) the loop $i_{|d_1...d_j}$ itself, that is, on an unfolding of the loop.

- When a *strict* is reached $\texttt{exe}(i_{|d_1...d_j}, d_{j+1}...d_n)$ is:

  - either $strict(\texttt{exe}(i_{|d_1...d_j.1}, d_{j+2}...d_n), i_{|d_1...d_j.2})$ if $d_{j+1} = 1$
  - or $\texttt{exe}(i_{|d_1...d_j.2}, d_{j+2}...d_n))$ if $d_{j+1} = 2$

  Indeed, the *strict* operator won't allow any action from the left branch to occur after an action on the right has occurred. As a result, when executing an action on its right (i.e. $d_{j+1} = 2$) we can simply ignore the whole left branch $i_{|d_1...d_j.1}$

- When a *seq* is reached $\texttt{exe}(i_{|d_1...d_j}, d_{j+1}...d_n)$ is:

  - either $seq(\texttt{exe}(i_{|d_1...d_j.1}, d_{j+2}...d_n), i_{|d_1...d_j.2})$ if $d_{j+1} = 1$
  - or $seq(\texttt{prn}(i_{|d_1...d_j.1}, \theta(i_{|p})), \texttt{exe}(i_{|d_1...d_j.2}, d_{j+2}...d_n))$ if $d_{j+1} = 2$

  Indeed, when executing an action on the right of a *seq* (i.e. when $d_{j+1} = 2$), we must prune the left cousin $i_{|d_1...d_j.1}$ w.r.t. the lifeline on which the action that is executed occurs.

Let us remark that the domain of definition of exe is defined by the precondition $p \in front(i)$. Its inductive definition, on the cases authorized by its precondition, guarantees that it is well-defined:

- If $i \in \mathbb{A}_\Omega$, p can only be ϵ (and vice-versa). In this case $\texttt{exe}(i, \epsilon) = (\varnothing, i)$ because the action i is executed and nothing remains to be executed.
- In any other case, p is either of the form $1.p_1$ or $2.p_2$.
  - If $p = 1.p_1$ then the action to be executed is in the left sub-interaction $i_{|1}$. Then the result of $\texttt{exe}(i, p)$ is a reconstruction of an interaction term using $\texttt{exe}(i_1, p_1)$ and i_2.
  - If $p = 2.p_2$ then the action to be executed is in the right sub-interaction $i_{|2}$. Then the result of $\texttt{exe}(i, p)$ is a reconstruction of an interaction term using i_1 and $\texttt{exe}(i_2, p_2)$.
- The most subtle case occurs when $p = 2.p_2$ and $i = seq(i_{|1}, i_{|2})$. The precondition $p \in front(i)$ implies that $i_{|p} \in \mathbb{A}_\Omega$ and that the left child $i_{|1}$ avoids $\theta(i_{|p})$. In this case, to construct $\texttt{exe}(i, 2.p_2)$, exe does not use $i_{|1}$ but rather its pruned version $\texttt{prn}(i_{|1}, \theta(i_{|p}))$ which eliminates all traces involving $\theta(i_{|p})$ while preserving all others.

6.1.5 Execution semantics

This process of identification and execution of frontier actions can be repeated on the follow-up interactions and their own follow-up interactions so as to identify all the executions that are specified by the original

interaction term. For each of those follow-up interactions, one can identify a new frontier and execute those actions and so on. This results in the construction of a tree which we call the "execution tree" and in which each path corresponds to parts of an execution of the original interaction.

On the diagram from Fig.6.4 such an execution tree is (partially) illustrated. This execution tree is that of the interaction from our previous examples (that from Fig.6.2 and Fig.6.3). Let us note that the transformation $i \xrightarrow{l_3!m_4@221} i'$ in which the interaction on the top part becomes the one underneath on its right (after transition $l_3!m_4@221$), corresponds to the execution illustrated on Fig.6.3. The three child interactions underneath the top one correspond to the executions of the 3 frontier actions of i (detailed in Fig.6.2).

Let us consider the path represented in the middle on Fig.6.4. We can see that it corresponds to the successive execution of three actions $l_1!m_1$, $l_3!m_4$ and $l_1?m_4$; the first in the original interaction, and the following in intermediate follow-up interactions. Let us then consider the interaction that remains after those three executions. We can see that is is the empty interaction $\varnothing$ drawn as a white square ($\square$). The empty interaction can express the empty behavior i.e. we have $\varnothing \downarrow$. As a result, we can conclude that the sequence of events constituted of the concatenation of those three events i.e. the trace $l_1!m_1.l_3!m_4.l_1?m_4$ constitutes a full execution of the system as specified by the original interaction and is therefore part of its trace semantics.

This observation can be generalized. Each path within the execution tree of an interaction that terminates in a node hosting an interaction that can express the empty behavior corresponds to an "accepted" trace of the original interaction. As a result, the execution tree represents the semantics of the original interaction. The principle of the execution semantics is in fact the same as that of the operational semantics from Chap.5, except that we explicit the positions of the actions that are executed.

For any interaction i, if it can express the empty behavior (which can be statically inferred from its structure via the "$\downarrow$" predicate) then the empty trace ϵ belongs to its semantics. If an action a at position p can be executed which yields to the follow-up interaction i', then any trace of the form $a.t$ where t is a trace accepted by i', is accepted by i.

In other words, starting from an initial interaction i_0, we may have a succession of transformations $i_0 \xrightarrow{a_1@p_1} i_1 \xrightarrow{a_2@p_2} \cdots \xrightarrow{a_n@p_n} i_n$ such that $i_n \downarrow$. We can then reconstitute a sequence $a_1.\cdots.a_n$ which is an accepted trace. By grouping all such paths together, we obtain a tree, called the execution tree, whose nodes are interactions and arcs can be labelled by couples (p, a) (where a is the action that is executed and p its position within the given interaction term) noted $a@p$. For a node i, child nodes are interactions i' obtained via the execution of any frontier action $a = i_{|p}$ with $p \in front(i)$. Any such child node i' corresponds to an interaction that accepts traces that are suffixes of traces accepted by i and which start with a (corresponding to a specific instance of a at a specific position p).

Let us also note that, given the existence of loops, execution trees can be infinite, and traces can be arbitrarily long. For instance, on Fig.6.4 the execution tree is only partially drawn (as indicated by the • • •) given that, in any case, the original interaction having a non-empty *loop* node, this tree is infinite.

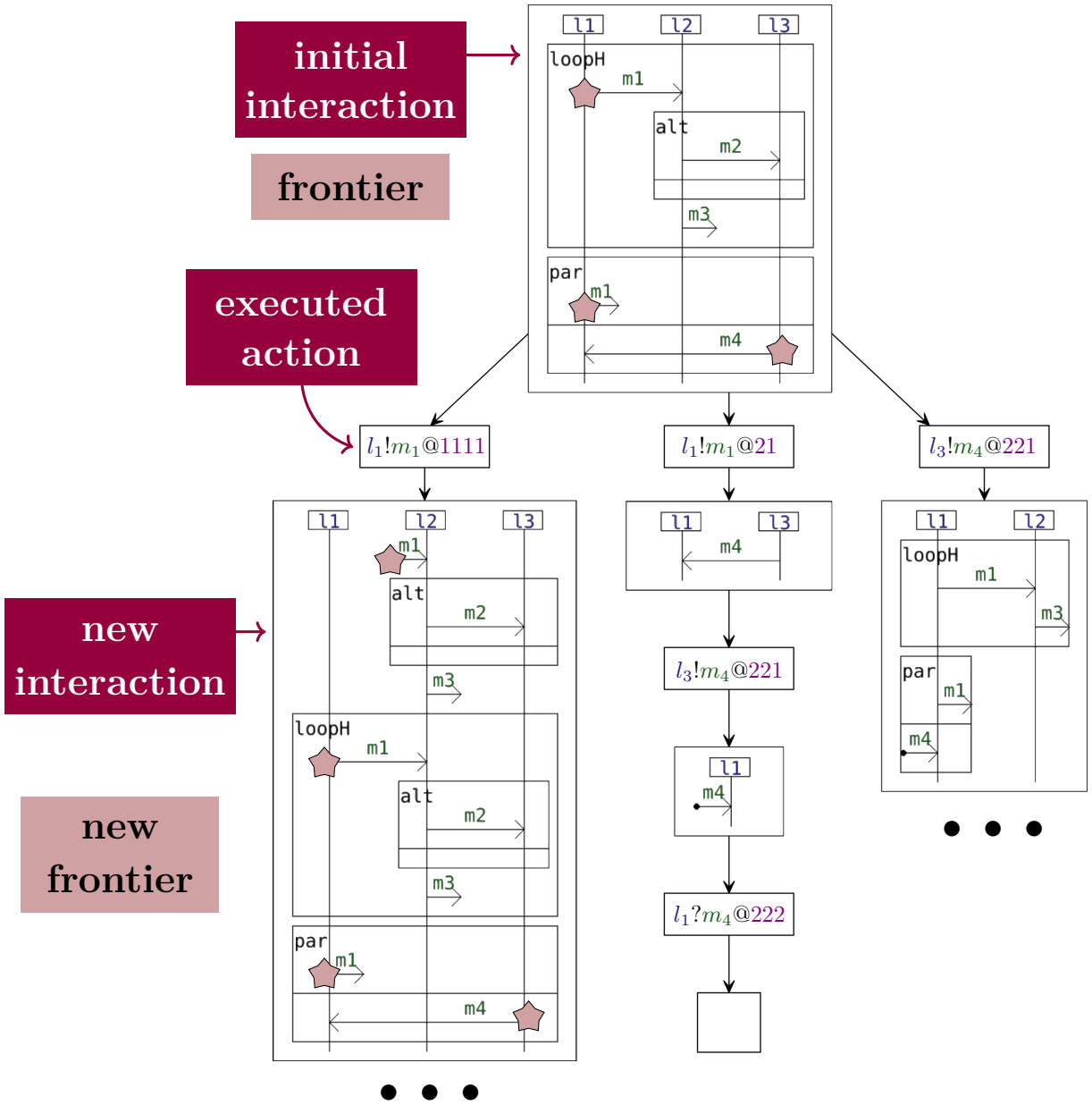


Figure 6.4: Description of the process for the execution semantics

Definition 6.7: Execution semantics

We define $\sigma_e : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ as:

$$\sigma_e(i) = empty(i) \cup \bigcup_{p \in \mathtt{frt}(i)} i_{|p}.\sigma_e(\mathtt{exe}(i,p))$$

with:

$$empty(i) = \begin{cases} \{\epsilon\} & \text{if } i \downarrow \\ \emptyset & \text{if } i \not\downarrow \end{cases}$$

In Def.6.7, we formally define our execution semantics σ_e, which can be obtained from the exploration of execution trees.

We have therefore formally defined our "execution semantics" σ_e which serves as an algorithmization of the operational semantics σ_o for the purposes of facilitating its application to Formal Verification (FV) techniques in the context of a tool implementation. So as to ensure its correctness, we will prove in the next section (Sec.6.2) that it is equivalent to the operational semantics i.e. that we have, for any interaction $i \in \mathbb{I}_\Omega$, $\sigma_e(i) = \sigma_o(i)$. Given that we have also proven, in Chap.5, that σ_o is equivalent to the foundational denotational-style semantics σ_d, we will then have proven that σ_e is equivalent to σ_d.

6.2 Proof of equivalence between σ_e and σ_o

The operational-style semantics σ_o and the execution semantics σ_e are very similar. Their main point of divergence concerns the use of the frontier and execution functions in σ_e, which substitute the execution relation $\rightarrow$ used in σ_o.

As a result, the crux of the proof will be to characterize "`exe`" and "`frt`" with regards to the execution relation $\rightarrow$. This is what we do in the next section (Sec.6.2.1). Then, in Sec.6.2.2 we use this characterization to prove the equivalence of both semantics.

Let us remark that this proof is machine-checked using the Coq proof assistant in [88].

6.2.1 Characterization of the execution function w.r.t. the execution relation

In Lem.6.4, we explain that for any predicate $i \xrightarrow{a} i'$, the execution of a in i to obtain i' in facts corresponds to the execution of a specific action a at a specific position p in the frontier of i. This corresponds to the fact that $i \xrightarrow{a} i'$ implies the existence of a position $p \in \texttt{frt}(i)$ such that:

- $i_{|p} = a$ meaning that within interaction i, the node at position p is a leaf node hosting an instance of action a (there can be several such instances)
- and $\texttt{exe}(i, p) = i'$ meaning that the follow-up interaction i' can be computed using "`exe`" and results from the execution of the specific instance of a at the specific position p

Lemma 6.4: Characterization (left side) of `exe` w.r.t. $\rightarrow$

For any interactions i and i' in $\mathbb{I}$, for any action $a \in \mathbb{A}_\Omega$, we have:

$$\left(i \xrightarrow{a} i' \right) \Rightarrow \left(\exists\, p \in \{1,2\}^*,\ (p \in \texttt{frt}(i)) \ \wedge\ (i_{|p} = a) \ \wedge\ (i' = \texttt{exe}(i,p)) \right)$$

Proof. Let us suppose that $i \xrightarrow{a} i'$ and let us reason by induction on the structure of i.

- If $i = \varnothing$ then it is not possible to have an i' such that $i \xrightarrow{a} i'$
- If $i = a \in \mathbb{A}_\Omega$ then:
  - On the one hand, the only possible i' such that $i \xrightarrow{a} i'$ is $\varnothing$ i.e. we have $a \xrightarrow{a} \varnothing$.

– On the other hand, we have $\epsilon \in \mathtt{frt}(a)$ and $a_{|\epsilon} = a$ and $\mathtt{exe}(a, \epsilon) = \varnothing$. Therefore the property holds.

- If $i = \mathit{strict}(i_1, i_2)$ then $i \xrightarrow{a} i'$ implies:

  – either that there exists i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $i' = \mathit{strict}(i'_1, i_2)$. We can then apply the induction hypothesis on sub-interaction i_1 to obtain the existence of a position $p_1 \in \mathtt{frt}(i_1)$ such that $i_{1|p_1} = a$ and $i'_1 = \mathtt{exe}(i_1, p_1)$. Then, by construction $1.p_1 \in \mathtt{frt}(i)$ and $i_{|1.p_1} = i_{1|p_1} = a$ and $\mathtt{exe}(i, 1.p_1) = \mathit{strict}(\mathtt{exe}(i_1, p_1), i_2) = \mathit{strict}(i'_1, i_2) = i'$. Therefore the property holds.

  – or that we have $i_1 \downarrow$ and that there exists i'_2 such that $i_2 \xrightarrow{a} i'_2$ and $i' = i'_2$. We can then apply the induction hypothesis on sub-interaction i_2 to obtain the existence of a position $p_2 \in \mathtt{frt}(i_2)$ such that $i_{2|p_2} = a$ and $i'_2 = \mathtt{exe}(i_2, p_2)$. Then, given that $i_1 \downarrow$, we have by construction that $2.p_2 \in \mathtt{frt}(i)$ and $i_{|2.p_2} = i_{2|p_2} = a$ and $\mathtt{exe}(i, 2.p_2) = \mathtt{exe}(i_2, p_2) = i'_2 = i'$. Therefore the property holds.

- If $i = \mathit{seq}(i_1, i_2)$ then $i \xrightarrow{a} i'$ implies:

  – either that there exists i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $i' = \mathit{seq}(i'_1, i_2)$. We can then apply the induction hypothesis on sub-interaction i_1 to obtain the existence of a position $p_1 \in \mathtt{frt}(i_1)$ such that $i_{1|p_1} = a$ and $i'_1 = \mathtt{exe}(i_1, p_1)$. Then, by construction $1.p_1 \in \mathtt{frt}(i)$ and $i_{|1.p_1} = i_{1|p_1} = a$ and $\mathtt{exe}(i, 1.p_1) = \mathit{seq}(\mathtt{exe}(i_1, p_1), i_2) = \mathit{seq}(i'_1, i_2) = i'$. Therefore the property holds.

  – or that we have $i_1 \downarrow^{*} \theta(a)$ and that there exists i'_2 such that $i_2 \xrightarrow{a} i'_2$ and $i' = \mathit{seq}(\mathtt{prn}(i_1, \theta(a)), i'_2)$. We can then apply the induction hypothesis on sub-interaction i_2 to obtain the existence of a position $p_2 \in \mathtt{frt}(i_2)$ such that $i_{2|p_2} = a$ and $i'_2 = \mathtt{exe}(i_2, p_2)$ and $i_{|2.p_2} = i_{2|p_2} = a$. Then, given that $i_1 \downarrow^{*} \theta(i_{|2.p_2})$, we have by construction that $2.p_2 \in \mathtt{frt}(i)$ and $i_{|2.p_2} = i_{2|p_2} = a$ and $\mathtt{exe}(i, 2.p_2) = \mathit{seq}(\mathtt{prn}(i_1, \theta(i_{2|p_2})), \mathtt{exe}(i_2, p_2)) = \mathit{seq}(\mathtt{prn}(i_1, \theta(a)), i'_2) = i'$. Therefore the property holds.

- If $i = \mathit{par}(i_1, i_2)$ then $i \xrightarrow{a} i'$ implies:

  – either that there exists i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $i' = \mathit{par}(i'_1, i_2)$. We can then apply the induction hypothesis on sub-interaction i_1 to obtain the existence of a position $p_1 \in \mathtt{frt}(i_1)$ such that $i_{1|p_1} = a$ and $i'_1 = \mathtt{exe}(i_1, p_1)$. Then, by construction $1.p_1 \in \mathtt{frt}(i)$ and $i_{|1.p_1} = i_{1|p_1} = a$ and $\mathtt{exe}(i, 1.p_1) = \mathit{par}(\mathtt{exe}(i_1, p_1), i_2) = \mathit{par}(i'_1, i_2) = i'$. Therefore the property holds.

  – either that there exists i'_2 such that $i_2 \xrightarrow{a} i'_2$ and $i' = \mathit{par}(i_1, i'_2)$. We can then apply the induction hypothesis on sub-interaction i_2 to obtain the existence of a position $p_2 \in \mathtt{frt}(i_2)$ such that $i_{2|p_2} = a$ and $i'_2 = \mathtt{exe}(i_2, p_2)$. Then, by construction $2.p_2 \in \mathtt{frt}(i)$ and $i_{|2.p_2} = i_{2|p_2} = a$ and $\mathtt{exe}(i, 2.p_2) = \mathit{par}(i_1, \mathtt{exe}(i_2, p_2)) = \mathit{par}(i_1, i'_2) = i'$. Therefore the property holds.

- If $i = \mathit{alt}(i_1, i_2)$ then $i \xrightarrow{a} i'$ implies:

  - either that there exists i'_1 such that $i_1 \xrightarrow{a} i'_1$ and $i' = i'_1$. We can then apply the induction hypothesis on sub-interaction i_1 to obtain the existence of a position $p_1 \in \texttt{frt}(i_1)$ such that $i_{1|p_1} = a$ and $i'_1 = \texttt{exe}(i_1, p_1)$. Then, by construction $1.p_1 \in \texttt{frt}(i)$ and $i_{|1.p_1} = i_{1|p_1} = a$ and $\texttt{exe}(i, 1.p_1) = \texttt{exe}(i_1, p_1) = i'_1 = i'$. Therefore the property holds.
  - either that there exists i'_2 such that $i_2 \xrightarrow{a} i'_2$ and $i' = i'_2$. We can then apply the induction hypothesis on sub-interaction i_2 to obtain the existence of a position $p_2 \in \texttt{frt}(i_2)$ such that $i_{2|p_2} = a$ and $i'_2 = \texttt{exe}(i_2, p_2)$. Then, by construction $2.p_2 \in \texttt{frt}(i)$ and $i_{|2.p_2} = i_{2|p_2} = a$ and $\texttt{exe}(i, 2.p_2) = \texttt{exe}(i_2, p_2) = i'_2 = i'$. Therefore the property holds.

- If $i = \mathit{loop}_k(i_1)$ with $k \in \{S, H, W, P\}$, then $i \xrightarrow{a} i'$ implies that there exists i'_1 such that $i_1 \xrightarrow{a} i'_1$ and i' is either $\mathit{strict}(i'_1, i)$ or $\mathit{seq}(i'_1, i)$ or $\mathit{seq}(\texttt{prn}(i, \theta(a)), \mathit{seq}(i'_1, i))$ (as per Lem.6.3) or $\mathit{par}(i'_1, i)$. In any case, we can apply the induction hypothesis on sub-interaction i_1 to obtain the existence of a position $p_1 \in \texttt{frt}(i_1)$ such that $i_{1|p_1} = a$ and $i'_1 = \texttt{exe}(i_1, p_1)$. Then, by construction $1.p_1 \in \texttt{frt}(i)$ and $i_{|1.p_1} = i_{1|p_1} = a$ and $\texttt{exe}(i, 1.p_1) = i'$. Therefore the property holds.

□

In Lem.6.5, we explain that for any frontier position p and corresponding action a within the frontier of an interaction i (i.e. $p \in \texttt{frt}(i)$ and $a = i_{|p}$), then $\texttt{exe}(i, p)$ indeed corresponds to a follow-up interaction of i after the execution of a according to the transition relation $\rightarrow$. This equates to implying that $i \xrightarrow{a} \texttt{exe}(i, p)$.

Lemma 6.5: Characterization (right side) of exe w.r.t. $\rightarrow$

For any interaction $i \in \mathbb{I}$, for any position $p \in \{1, 2\}^*$ and for any action $a \in \mathbb{A}_\Omega$, we have:

$$\Big(\ (p \in \texttt{frt}(i)) \ \wedge \ (i_{|p} = a) \ \Big) \Rightarrow \Big(\ i \xrightarrow{a} \texttt{exe}(i, p) \ \Big)$$

Proof. Let us suppose that $p \in \texttt{frt}(i)$ and that $i_{|p} = a$. Let us then reason by induction on the structure of i:

- If $i = \varnothing$ then we have $\texttt{frt}(\varnothing) = \emptyset$ so it is not possible to have a $p \in \texttt{frt}(i)$.
- If $i = a \in \mathbb{A}_\Omega$ then we have $\texttt{frt}(a) = \{\epsilon\}$ and p must be ϵ, which satisfies $a_{|\epsilon} = a$. Then:
  - On the one hand, we have $a \xrightarrow{a} \varnothing$.
  - On the other hand, $\texttt{exe}(a, \epsilon) = \varnothing$. Therefore the property holds.
- If $i = \mathit{strict}(i_1, i_2)$ then $p \in \texttt{frt}(i)$ implies:
  - either that p is of the form $1.p_1$ and $p_1 \in \texttt{frt}(i_1)$. Also we have $i_{1|p_1} = i_{|1.p_1} = a$. We can then apply the induction hypothesis on sub-interaction i_1 to get that we have $i_1 \xrightarrow{a} \texttt{exe}(i_1, p_1)$. This implies, as per the definition of the execution relation, that $\mathit{strict}(i_1, i_2) \xrightarrow{a} \mathit{strict}(\texttt{exe}(i_1, p_1), i_2)$.

Then, given that $\texttt{exe}(i, 1.p_1) = \mathit{strict}(\texttt{exe}(i_1, p_1), i_2)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

– or that we have $i_1 \downarrow$, that p is of the form $2.p_2$ and that $p_2 \in \texttt{frt}(i_2)$. Also we have $i_{2|p_2} = i_{|2.p_2} = a$. We can then apply the induction hypothesis on sub-interaction i_2 to get that we have $i_2 \xrightarrow{a} \texttt{exe}(i_2, p_2)$. Given that we have $i_1 \downarrow$, this implies, as per the definition of the execution relation, that $\mathit{strict}(i_1, i_2) \xrightarrow{a} \texttt{exe}(i_2, p_2)$. Then, given that $\texttt{exe}(i, 2.p_2) = \texttt{exe}(i_2, p_2)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

- If $i = \mathit{seq}(i_1, i_2)$ then $p \in \texttt{frt}(i)$ implies:

  – either that p is of the form $1.p_1$ and $p_1 \in \texttt{frt}(i_1)$. Also we have $i_{1|p_1} = i_{|1.p_1} = a$. We can then apply the induction hypothesis on sub-interaction i_1 to get that we have $i_1 \xrightarrow{a} \texttt{exe}(i_1, p_1)$. This implies, as per the definition of the execution relation, that $\mathit{seq}(i_1, i_2) \xrightarrow{a} \mathit{seq}(\texttt{exe}(i_1, p_1), i_2)$. Then, given that $\texttt{exe}(i, 1.p_1) = \mathit{seq}(\texttt{exe}(i_1, p_1), i_2)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

  – or that we have $i_1 \downarrow^{*} \theta(i_{|p})$, that p is of the form $2.p_2$ and that $p_2 \in \texttt{frt}(i_2)$. Also we have $i_{2|p_2} = i_{|2.p_2} = a$. We can then apply the induction hypothesis on sub-interaction i_2 to get that we have $i_2 \xrightarrow{a} \texttt{exe}(i_2, p_2)$. Given that we have $i_1 \downarrow \theta(a)$, this implies, as per the definition of the execution relation, that $\mathit{seq}(i_1, i_2) \xrightarrow{a} \mathit{seq}(\texttt{prn}(i_1, \theta(a)), \texttt{exe}(i_2, p_2))$. Then, given that $\texttt{exe}(i, 2.p_2) = \mathit{seq}(\texttt{prn}(i_1, \theta(i_{2|p_2})), \texttt{exe}(i_2, p_2))$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

- If $i = \mathit{par}(i_1, i_2)$ then $p \in \texttt{frt}(i)$ implies:

  – either that p is of the form $1.p_1$ and $p_1 \in \texttt{frt}(i_1)$. Also we have $i_{1|p_1} = i_{|1.p_1} = a$. We can then apply the induction hypothesis on sub-interaction i_1 to get that we have $i_1 \xrightarrow{a} \texttt{exe}(i_1, p_1)$. This implies, as per the definition of the execution relation, that $\mathit{par}(i_1, i_2) \xrightarrow{a} \mathit{par}(\texttt{exe}(i_1, p_1), i_2)$. Then, given that $\texttt{exe}(i, 1.p_1) = \mathit{par}(\texttt{exe}(i_1, p_1), i_2)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

  – either that p is of the form $2.p_2$ and $p_2 \in \texttt{frt}(i_2)$. Also we have $i_{2|p_2} = i_{|2.p_2} = a$. We can then apply the induction hypothesis on sub-interaction i_2 to get that we have $i_2 \xrightarrow{a} \texttt{exe}(i_2, p_2)$. This implies, as per the definition of the execution relation, that $\mathit{par}(i_1, i_2) \xrightarrow{a} \mathit{par}(i_1, \texttt{exe}(i_2, p_2))$. Then, given that $\texttt{exe}(i, 2.p_2) = \mathit{par}(i_1, \texttt{exe}(i_2, p_2))$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

- If $i = \mathit{alt}(i_1, i_2)$ then $p \in \texttt{frt}(i)$ implies:

  – either that p is of the form $1.p_1$ and $p_1 \in \texttt{frt}(i_1)$. Also we have $i_{1|p_1} = i_{|1.p_1} = a$. We can then apply the induction hypothesis on sub-interaction i_1 to get that we have $i_1 \xrightarrow{a} \texttt{exe}(i_1, p_1)$. This

implies, as per the definition of the execution relation, that $alt(i_1, i_2) \xrightarrow{a} \texttt{exe}(i_1, p_1)$. Then, given that $\texttt{exe}(i, 1.p_1) = \texttt{exe}(i_1, p_1)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

– or that p is of the form $2.p_2$ and $p_2 \in \texttt{frt}(i_2)$. Also we have $i_{2|p_2} = i_{|2.p_2} = a$. We can then apply the induction hypothesis on sub-interaction i_2 to get that we have $i_2 \xrightarrow{a} \texttt{exe}(i_2, p_2)$. This implies, as per the definition of the execution relation, that $alt(i_1, i_2) \xrightarrow{a} \texttt{exe}(i_2, p_2)$. Then, given that $\texttt{exe}(i, 2.p_2) = \texttt{exe}(i_2, p_2)$ this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

- If $i = loop_k(i_1)$ with $k \in \{S, H, W, P\}$ then $p \in \texttt{frt}(i)$ implies that p is of the form $1.p_1$ and $p_1 \in \texttt{frt}(i_1)$. Also we have $i_{1|p_1} = i_{|1.p_1} = a$. We can apply the induction hypothesis on sub-interaction i_1 to get that we have $i_1 \xrightarrow{a} \texttt{exe}(i_1, p_1)$. Let us then denote by i'_1 the interaction $\texttt{exe}(i_1, p_1)$. This then implies, as per the definition of the execution relation, that $loop_k(i_1) \xrightarrow{a} i'$ with i' being either $strict(i'_1, i)$ or $seq(i'_1, i)$ or $seq(\texttt{prn}(i, \theta(a)), seq(i'_1, i))$ (as per Lem.6.3) or $par(i'_1, i)$ Then, given that $\texttt{exe}(i, 1.p_1)$ is exactly defined as i'_1, this equates to $i \xrightarrow{a} \texttt{exe}(i, p)$. Therefore the property holds.

□

6.2.2 Conclusion

Thanks to the previous characterization of $front$ and χ w.r.t. $\rightarrow$, the actual proof is quite short. In this section, we will:

- in Th.6.1 prove the inclusion of σ_e in σ_o
- in Th.6.2 prove the inclusion of σ_o in σ_e
- conclude on the equivalence of the semantics with Th.6.3

Theorem 6.1: Inclusion of σ_e in σ_o

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_e(i) \supset \sigma_o(i)$$

Proof. Let us consider $i \in \mathbb{I}_\Omega$ and $t \in \sigma_d(i)$ and let us reason by induction on the trace t.

- If $t = \epsilon$, the fact that $t = \epsilon \in \sigma_e(i)$ implies that $empty(i) = \{\epsilon\}$ and therefore that $i \downarrow$. Then, by definition of σ_o, this means that $\epsilon \in \sigma_o(i)$.
- If $t = a.t'$ then, the fact that $a.t' \in \sigma_e(i)$ implies the existence of a frontier position $p \in \texttt{frt}(i)$ such that $i_{|p} = a$ and $t \in \sigma_e(\texttt{exe}(i, p))$.
  – On the one hand, given that trace t' is strictly smaller than t, we can apply the induction hypothesis which implies that $t' \in \sigma_o(\texttt{exe}(i, p))$
  – On the other hand, we can apply Lem.6.5 to get that $i \xrightarrow{a} \texttt{exe}(i, p)$

– Those two facts combined imply that $t = a.t' \in \sigma_o(i)$

□

Theorem 6.2: Inclusion of σ_o in σ_e

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_o(i) \supset \sigma_e(i)$$

Proof. Let us consider $i \in \mathbb{I}_\Omega$ and $t \in \sigma_d(i)$ and let us reason by induction on the (size of the) trace t.

- If $t = \epsilon$, the fact that $t = \epsilon \in \sigma_o(i)$ implies that $i \downarrow$. Then, this implies that $empty(i) = \{\epsilon\}$ and therefore that $\epsilon \in \sigma_e(i)$.
- If $t = a.t'$ then, the fact that $a.t' \in \sigma_o(i)$ implies the existence of an action a and an interaction i' such that $i \xrightarrow{a} i'$ and $t' \in \sigma_o(i')$.
  - On the one hand, we can apply the induction hypothesis on t', which implies that $t' \in \sigma_e(i')$
  - On the other hand, we can apply Lem.6.4 to reveal the existence of a position $p \in \texttt{frt}(i)$ such that $i_{|p} = a$ and $i' = \texttt{exe}(i, p)$
  - As a result we have $p \in \texttt{frt}(i)$ and $t' \in \sigma_e(\texttt{exe}(i, p))$. Therefore, by definition of σ_e we have $t = a.t' = i_{|p}.t' \in \sigma_e(i)$

□

We conclude with Th.6.3 that the three semantics that we have defined on our IL, namely σ_d, σ_o and σ_e are all equivalent.

Theorem 6.3: Equivalence of all three semantics

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_d(i) = \sigma_o(i) = \sigma_e(i)$$

Proof. Immediately implies by Th.5.3, Th.6.1 and Th.6.2. □

6.3 Execution semantics with simplifications

The operational semantics σ_o and its algorithmization σ_e are defined in such a manner that the nodes within an interaction term that host the actions that are executed are progressively (at each small-step) replaced by the empty interaction $\varnothing$. Also, when executing actions that are found within loops, the interaction term is complexified with the addition of new nodes that correspond to a duplication of the sub-interaction underneath the loop that is instanciated. This results in an accumulation of useless nodes which increases

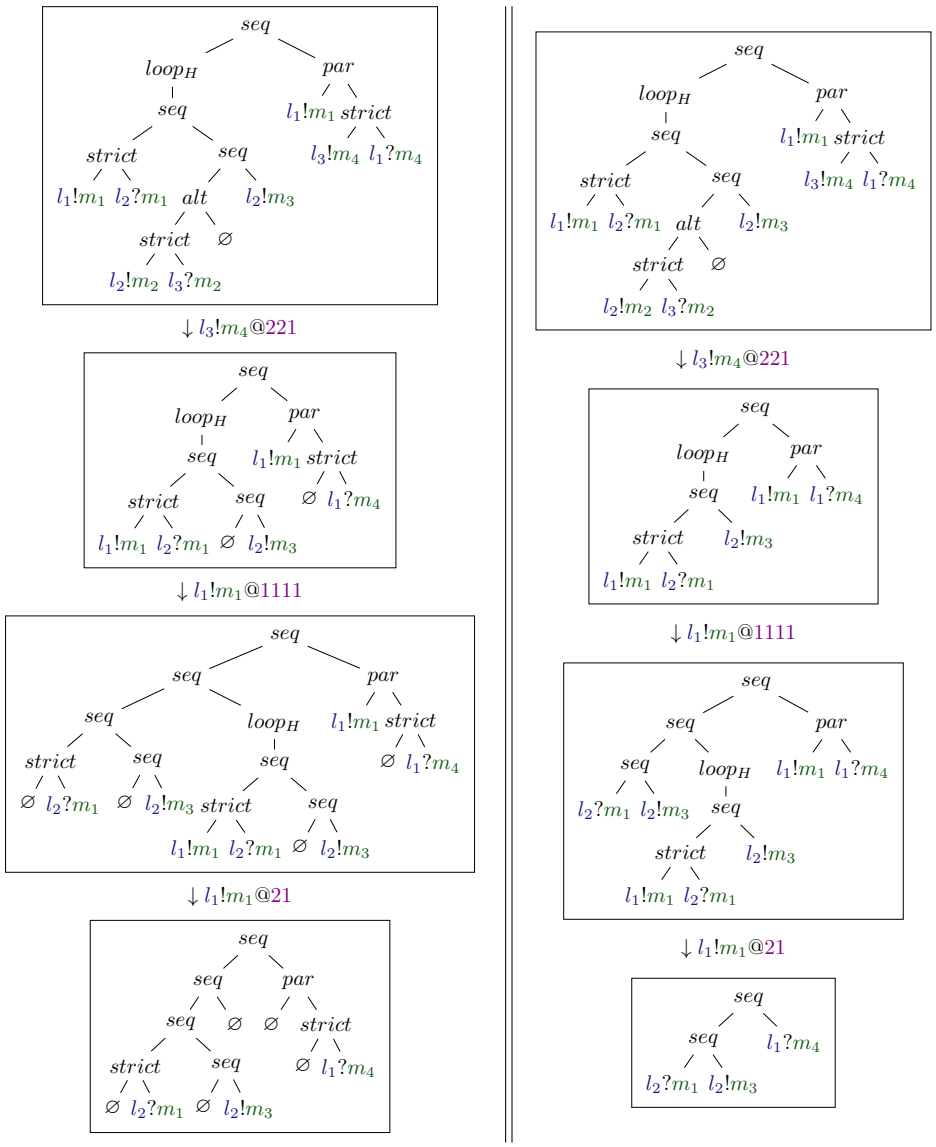


Figure 6.5: Example of a succession of small-step executions without (left) and with (right) simplifications

with each application of a small-step. In other words, and informally, interactions may become less and less "canonic" during their executions, especially when they contain loops.

Let us indeed consider the illustration from Fig.6.5, which details two possible successions of transformations starting from the same initial interaction which is the example of Fig.6.3.

The example on the left of Fig.6.5 consists in the exploration using σ_e of a path in the execution tree of the initial interaction, which reveals that $l_3!m_4.l_1!m_1.l_1!m_1$ is a prefix of an (at least one) accepted trace. We can see however that at each execution of a small-step, redundant $\varnothing$ nodes are added, and, in the interaction term that is reached at the end of the path, many such occurrences of redundant $\varnothing$ nodes are present.

Removing those redundant $\varnothing$ (among other potential simplifications) would allow an implementation of the semantics to work with simplified, more compact interaction terms. Working with simpler terms would in turn improve overall performances given that it both reduces the memory footprint and the length of the traversals for computing inductive predicates on the terms.

On the example on the right of Fig.6.5 is represented the execution of the same (prefix of) trace from

the same initial interaction. However, in this case, we have used some simplifications when defining each intermediate interaction term. The resulting term at the end of the path is here simpler and more compact than the one reached in the example on the left.

In the following we define two manners of defining a (proven correct) execution semantics which incorporate such simplifications. Doing so involves results from equational logic and term rewriting which we have covered in Chap.4.

6.3.1 Normalizing intermediate interaction terms

In Chap.4 we have defined a process to normalize interaction terms. Let us recall that, for any interaction i we may denote by $R_{\prec}(i)$ its unique normal form through the rewriting process (given the three phases $R^1_{\mathbb{I}}$, $R^2_{\mathbb{I}}$ and $R^3_{\mathbb{I}}$ for simplifications and the total rewrite order $\prec_{\mathbb{I}}$ for selecting a unique represent in the AC-equivalence class).

A first obvious method for incorporating simplifications into the execution semantics is to normalize each intermediate interaction term. Doing so leads to defining the semantics which we denote by σ_n (for "normalizing") from Def.6.8

Definition 6.8: Normalizing execution semantics

We define $\sigma_n : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ and (the tool function) $\texttt{sem} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ as:

$$\sigma_n \ : \ i \mapsto \texttt{sem}(R(i)) \qquad \text{and} \qquad \texttt{sem} \ : \ i \mapsto \mathit{empty}(i) \cup \bigcup_{p \in \texttt{frt}(i)} i_{|p}.\sigma_n(\texttt{exe}(i,p))$$

with *empty* from Def.6.7 and R from Chap.4

The "normalizing" execution semantics that is thus defined is correct w.r.t. the previous semantics, which we state in Th.6.4.

Theorem 6.4: Correctness of the normalizing execution semantics

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_n(i) = \sigma_e(i)$$

Proof. Let us proceed by induction on a member trace t:

- if $t = \epsilon$ then $\epsilon \in \sigma_n(i)$ and $\epsilon \in \sigma_e(i)$ respectively equate $R_{\prec}(i)\downarrow$ and $i\downarrow$. By Lem.5.4 those predicates then respectively equate that $\epsilon \in \sigma_d(R_{\prec}(i))$ and $\epsilon \in \sigma_d(i)$. Then, given that i and $R_{\prec}(i)$ are in the same equivalence class, we have that $\sigma_d(R_{\prec}(i)) = \sigma_d(i)$. Hence both predicates are equivalent.
- if $t = a.t'$ then:
  - $\subset$ if $a.t' \in \sigma_n(i)$ there exists i' s.t. $R_{\prec}(i) \xrightarrow{a} i'$ and $t' \in \sigma_n(i')$ then:

- at first we can apply the induction hypothesis so that $t' \in \sigma_e(i')$
- then, given that $R_\prec(i) \xrightarrow{a} i'$ and $t' \in \sigma_e(i')$ we have by definition that $a.t' \in \sigma_e(R_\prec(i))$
- finally, as per Th.6.3 and because $R_\prec(i) \approx_{E_\mathbb{I}} i$ this means that $a.t' \in \sigma_e(i)$

$\supset$ if $a.t' \in \sigma_e(i)$ there exists i' s.t. $i \xrightarrow{a} i'$ and $t' \in \sigma_e(i')$ then:

- at first we can apply the induction hypothesis so that $t' \in \sigma_n(i')$
- the fact that $i \xrightarrow{a} i'$ imply that $R_\prec(i) \xrightarrow{a} R_\prec(i')$ and the fact that $t' \in \sigma_n(i') = \sigma_n(R_\prec(i'))$ imply that $a.t' \in \sigma_n(R_\prec(i)) = \sigma_n(i)$

□

However, we might argue that this algorithmicized semantics may have high overhead costs due to the fact that, in between each step of execution, we normalize the follow-up interaction which involves trying to find redexes and which can be costly.

As a result, in the next section, we propose another execution semantics with simplifications, which, instead of simplifying a-posteriori the i' such that $i \xrightarrow{a} i'$ merges the simplification with the process of reconstructing i' itself. As we will see this can be very simply done by introducing modifications in the inductive definitions of the pruning and execution functions.

6.3.2 Simplifying while computing the intermediate interaction terms

As mentioned previously, we can define variants of the prune and execution functions which introduce term simplification in their inductive definitions.

Let us consider the example from Fig.6.6 and compare what is represented here with what we have seen on Fig.6.3. Both figures represent the execution of the same action at the same position in the same interaction term. However, while Fig.6.3 illustrates the process without term simplification, Fig.6.6 illustrates a version with term simplification. We can see that the simplification steps illustrated on Fig.6.6 yield a simpler and more compact interaction term.

As mentioned previously, we formalize this process by defining a $\texttt{prn}_\approx$ and a $\texttt{exe}_\approx$ function.

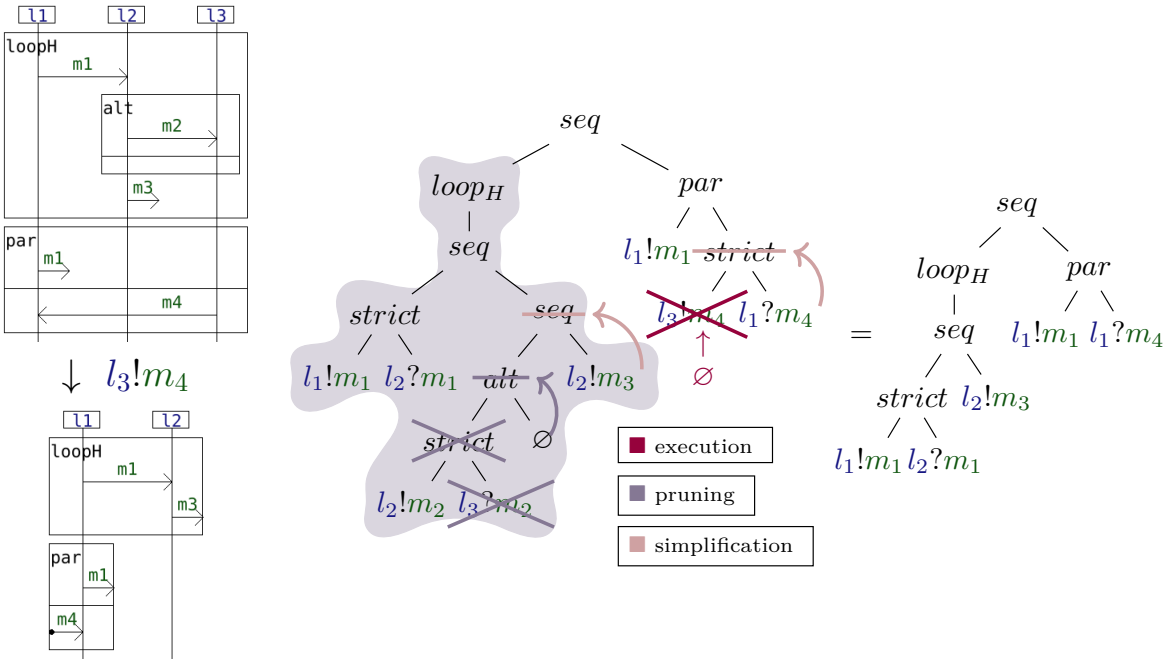


Figure 6.6: Illustration of the execution of an action at a specific position with simplifications

Definition 6.9: Pruning with simplifications

$\texttt{prn}_{\approx} : \mathbb{I}_\Omega \times L \to \mathbb{I}_\Omega$ is s.t. for any $(i,l) \in \mathbb{I}_\Omega \times L$ verifying $\downarrow^{*} (i,l)$:

- **provided** $\downarrow^{*} (i,l)$ **then** $\texttt{prn}_{\approx}(i,l) =$ **match** i **with**

$$
\begin{array}{lll}
\mid \varnothing & \to & \varnothing \\
\mid a \in \mathbb{A}_\Omega & \to & a \\
\mid alt(i_1,i_2) & \to & \left\{ \begin{array}{ll}
\texttt{prn}_{\approx}(i_2,l) & \textbf{if } \downarrow^{*} (i_2,l) \wedge \neg \downarrow^{*} (i_1,l) \\
\texttt{prn}_{\approx}(i_1,l) & \textbf{if } \downarrow^{*} (i_1,l) \wedge \neg \downarrow^{*} (i_2,l) \\
alt(\texttt{prn}_{\approx}(i_1,l), \texttt{prn}_{\approx}(i_2,l)) & \textbf{if } \downarrow^{*} (i_1,l) \wedge \downarrow^{*} (i_2,l)
\end{array} \right. \\
\mid f(i_1,i_2) & \to & \left\{ \begin{array}{ll}
\texttt{prn}_{\approx}(i_2,l) & \textbf{if } \texttt{prn}_{\approx}(i_1,l) = \varnothing \\
\texttt{prn}_{\approx}(i_1,l) & \textbf{if } \texttt{prn}_{\approx}(i_2,l) = \varnothing \\
f(\texttt{prn}_{\approx}(i_1,l), \texttt{prn}_{\approx}(i_2,l)) & \textbf{else}
\end{array} \right. \quad \textbf{for } f \in \{strict, seq, par\} \\
\mid loop_k(i_1) & \to & \left\{ \begin{array}{ll}
loop_k(\texttt{prn}_{\approx}(i_1,l)) & \textbf{if } \downarrow^{*} (i_1,l) \textbf{ and } \texttt{prn}_{\approx}(i_1,l) \neq \varnothing \\
\varnothing & \textbf{else}
\end{array} \right. \quad \textbf{for } k \in \{S,H,W,P\}
\end{array}
$$

The new $\texttt{prn}_{\approx}$ function from Def.6.9 operates the pruning process with simplifications. We characterize this new $\texttt{prn}_{\approx}$ function w.r.t. $\texttt{prn}$ in Lem.6.6. An interaction term $\texttt{prn}_{\approx}(i,l)$ resulting from the application of this new pruning process being equivalent according to $\approx_{E_{\mathbb{I}}}$ to $\texttt{prn}(i,l)$.

Lemma 6.6: Characterization of pruning with simplifications

For any interaction $i \in \mathbb{I}_\Omega$ and any $l \in L$ such that $i \downarrow^* l$ we have:

$$\texttt{prn}(i,l) \approx_{E_\mathbb{I}} \texttt{prn}_\approx(i,l)$$

Proof. Let us reason by induction on the term structure of i

- if $i = \varnothing$ then $\texttt{prn}(i,l) = \varnothing$ and $\texttt{prn}_\approx(i,l) = \varnothing$ then $\approx_{E_\mathbb{I}}$ being reflexive, the property holds
- if $i = a \in \mathbb{A}_\Omega$ which implies that $\theta(a) \neq l$ then $\texttt{prn}(i,l) = a$ and $\texttt{prn}_\approx(i,l) = a$ and then $\approx_{E_\mathbb{I}}$ being reflexive, the property holds
- if $i = alt(i_1, i_2)$ then:
  - if $\downarrow^* (i_2, l) \ \wedge \ \neg \downarrow^* (i_1, l)$ then $\texttt{prn}(i,l) = \texttt{prn}(i_2,l)$ and $\texttt{prn}_\approx(i,l) = \texttt{prn}_\approx(i_2,l)$. We can then apply the induction hypothesis on i_2 which implies that $\texttt{prn}(i_2,l) \approx_{E_\mathbb{I}} \texttt{prn}_\approx(i_2,l)$ and therefore the property holds
  - if $\downarrow^* (i_1, l) \ \wedge \ \neg \downarrow^* (i_2, l)$ then $\texttt{prn}(i,l) = \texttt{prn}(i_1,l)$ and $\texttt{prn}_\approx(i,l) = \texttt{prn}_\approx(i_1,l)$. We can then apply the induction hypothesis on i_1 which implies that $\texttt{prn}(i_1,l) \approx_{E_\mathbb{I}} \texttt{prn}_\approx(i_1,l)$ and therefore the property holds
  - if $\downarrow^* (i_1, l) \wedge \downarrow^* (i_2, l)$ then $\texttt{prn}(i,l) = alt(\texttt{prn}(i_1,l), \texttt{prn}(i_2,l))$ and $\texttt{prn}_\approx(i,l) = alt(\texttt{prn}_\approx(i_1,l), \texttt{prn}_\approx(i_2,l))$. We can then apply the induction hypothesis on both i_1 and i_2, and, given that $\approx_{E_\mathbb{I}}$ is $\mathcal{F}$-compatible (congruence), the property holds
- if $i = f(i_1, i_2)$ with $f \in \{strict, seq, par\}$ then we have $\texttt{prn}(i,l) = f(\texttt{prn}(i_1,l), \texttt{prn}(i_2,l))$ and, as per the induction hypothesis on i_1 and i_2 we have both $\texttt{prn}(i_1,l) \approx_{E_\mathbb{I}} \texttt{prn}_\approx(i_1,l)$ and $\texttt{prn}(i_2,l) \approx_{E_\mathbb{I}} \texttt{prn}_\approx(i_2,l)$ and then:
  - if $\texttt{prn}_\approx(i_1,l) = \varnothing$ then:

$$\begin{array}{lll} \texttt{prn}_\approx(i,l) & = & \texttt{prn}_\approx(i_2,l) \\ & \approx_{E_\mathbb{I}} & f(\varnothing, \texttt{prn}_\approx(i_2,l)) \qquad \text{(as per Simpl-Left)} \\ & \approx_{E_\mathbb{I}} & f(\texttt{prn}(i_1,l), \texttt{prn}(i_2,l)) \\ & \approx_{E_\mathbb{I}} & \texttt{prn}(i,l) \end{array}$$

    hence the property holds

– if $\texttt{prn}_{\approx}(i_2, l) = \varnothing$ then:

$$\begin{array}{rcll} \texttt{prn}_{\approx}(i, l) & = & \texttt{prn}_{\approx}(i_1, l) & \\ & \approx_{E_{\mathbb{I}}} & f(\texttt{prn}_{\approx}(i_1, l), \varnothing) & \text{(as per Simpl-Right)} \\ & \approx_{E_{\mathbb{I}}} & f(\texttt{prn}(i_1, l), \texttt{prn}(i_2, l)) & \\ & \approx_{E_{\mathbb{I}}} & \texttt{prn}(i, l) & \end{array}$$

hence the property holds

– else the property is implied by the fact that $\approx_{E_{\mathbb{I}}}$ is $\mathcal{F}$-compatible (congruence)

- if $i = loop_k(i_1)$ with $k \in \{S, H, W, P\}$, we have, as per the induction hypothesis that $\texttt{prn}(i_1, l) \approx_{E_{\mathbb{I}}} \texttt{prn}_{\approx}(i_1, l)$ and then:

– if $\downarrow^{*} (i_1, l)$ then $\texttt{prn}(i, l) = loop_k(\texttt{prn}(i_1, l))$ and:

* if $\texttt{prn}_{\approx}(i_1, l) \neq \varnothing$ then $\texttt{prn}_{\approx}(i) = loop_k(\texttt{prn}_{\approx}(i_1, l))$ and the property holds because $\approx_{E_{\mathbb{I}}}$ is a congruence

* if $\texttt{prn}_{\approx}(i_1, l) = \varnothing$ then:

$$\begin{array}{rcll} \texttt{prn}_{\approx}(i, l) & = & \varnothing & \\ & \approx_{E_{\mathbb{I}}} & loop_k(\varnothing) & \text{(as per Loop-Simpl)} \\ & \approx_{E_{\mathbb{I}}} & loop_k(\texttt{prn}(i_1, l)) & \\ & \approx_{E_{\mathbb{I}}} & \texttt{prn}(i, l) & \end{array}$$

– else then $\texttt{prn}(i, l) = \varnothing = \texttt{prn}_{\approx}(i, l)$

□

As a result, pruning with simplification that we defined in Def.6.9 is equivalent (in the sens of $\approx_{E_{\mathbb{I}}}$) to the original pruning.

In the same spirit, we define in Def.6.10 a variant of the execution function $\texttt{exe}_{\approx}$ that incorporates simplification steps. We then prove with Lem.6.7 that this new execution with simplifications is equivalent (in the sens of $\approx_{E_{\mathbb{I}}}$) to the original.

Definition 6.10: Execution with simplifications

$\texttt{exe}_{\approx} : \mathbb{I}_\Omega \times \{1,2\}^* \to \mathbb{I}_\Omega$ is the function s.t. for any $i \in \mathbb{I}_\Omega$ and $p \in \texttt{frt}(i)$:

- $\texttt{exe}_{\approx}(i,p) =$ **match** (i,p) **with**

$$
\begin{array}{lll}
\mid (act, \epsilon) & \to & \varnothing \\
\mid (alt(i_1,i_2), 1.p_1) & \to & \texttt{exe}_{\approx}(i_1,p_1) \\
\mid (alt(i_1,i_2), 2.p_2) & \to & \texttt{exe}_{\approx}(i_2,p_2) \\
\mid (strict(i_1,i_2), 1.p_1) & \to & \begin{cases} strict(\texttt{exe}_{\approx}(i_1,p_1), i_2) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i_2 & \textbf{else} \end{cases} \\
\mid (strict(i_1,i_2), 2.p_2) & \to & \texttt{exe}_{\approx}(i_2,p_2) \\
\mid (seq(i_1,i_2), 1.p_1) & \to & \begin{cases} seq(\texttt{exe}_{\approx}(i_1,p_1), i_2) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i_2 & \textbf{else} \end{cases} \\
\mid (seq(i_1,i_2), 2.p_2) & \to & \begin{cases} \texttt{exe}_{\approx}(i_2,p_2) & \textbf{if } \texttt{prn}(i_1,\theta(i_{2|p_2})) = \varnothing \\ \texttt{prn}(i_1,\theta(i_{2|p_2})) & \textbf{if } \texttt{exe}_{\approx}(i_2,p_2) = \varnothing \\ seq(\texttt{prn}(i_1,\theta(i_{2|p_2})), \texttt{exe}_{\approx}(i_2,p_2)) & \textbf{if both are not } \varnothing \end{cases} \\
\mid (par(i_1,i_2), 1.p_1) & \to & \begin{cases} par(\texttt{exe}_{\approx}(i_1,p_1), i_2) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i_2 & \textbf{else} \end{cases} \\
\mid (par(i_1,i_2), 2.p_2) & \to & \begin{cases} par(i_1, \texttt{exe}_{\approx}(i_2,p_2)) & \textbf{if } \texttt{exe}_{\approx}(i_2,p_2) \neq \varnothing \\ i_1 & \textbf{else} \end{cases} \\
\mid (loop_S(i_1), 1.p_1) & \to & \begin{cases} strict(\texttt{exe}_{\approx}(i_1,p_1), i) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i & \textbf{else} \end{cases} \\
\mid (loop_H(i_1), 1.p_1) & \to & \begin{cases} seq(\texttt{exe}_{\approx}(i_1,p_1), i) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i & \textbf{else} \end{cases} \\
\mid (loop_W(i_1), 1.p_1) & \to & \begin{cases} seq(\texttt{prn}(i,\theta(i_{1|p_1})), seq(\texttt{exe}_{\approx}(i_1,p_1), i)) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1)\neq\varnothing \textbf{ and } \texttt{prn}(i,\theta(i_{1|p_1}))\neq\varnothing \\ seq(\texttt{exe}_{\approx}(i_1,p_1), i) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1)\neq\varnothing \textbf{ and } \texttt{prn}(i,\theta(i_{1|p_1}))=\varnothing \\ seq(\texttt{prn}(i,\theta(i_{1|p_1})), i) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1)=\varnothing \textbf{ and } \texttt{prn}(i,\theta(i_{1|p_1}))\neq\varnothing \\ i & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1)=\texttt{prn}(i,\theta(i_{1|p_1}))=\varnothing \end{cases} \\
\mid (loop_P(i_1), 1.p_1) & \to & \begin{cases} par(\texttt{exe}_{\approx}(i_1,p_1), i) & \textbf{if } \texttt{exe}_{\approx}(i_1,p_1) \neq \varnothing \\ i & \textbf{else} \end{cases}
\end{array}
$$

Lemma 6.7: Characterization of execution with simplifications

For any interaction $i \in \mathbb{I}_\Omega$ and any frontier position $p \in front(i)$ we have:

$$\texttt{exe}(i,p) \approx_{E_\mathbb{I}} \texttt{exe}_{\approx}(i,p)$$

Proof. As for the proof of Lem.6.6 we can proceed by induction. For the case where $i = seq(i_1,i_2)$ and

$p = 2.p_2$ and for the case $i = loop_W(i_1)$ and $p = 1.p_1$ we apply Lem.6.6 related to pruning. □

Finally, we define the execution semantics with simplifications in Def.6.11. This new semantics makes use of the $\texttt{prn}_{\approx}$ and $\texttt{exe}_{\approx}$ functions to incorporate simplification during the computation of intermediate interactions. We then prove in Th.6.5 that this new variant of the execution semantics, which we denote by $\underset{e}{\sigma_{\approx}}$ is equivalent to σ_e and therefore to all the previously defined semantics.

Definition 6.11: Execution semantics with simplifications

We define $\underset{e}{\sigma_{\approx}} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_\Omega)$ as:

$$\underset{e}{\sigma_{\approx}}(i) = empty(i) \cup \bigcup_{p \in front(i)} i_{|p}.\underset{e}{\sigma_{\approx}}(\texttt{exe}_{\approx}(i,p))$$

with *empty* from Def.6.7

Theorem 6.5: Correctness of the simplifying execution semantics

For any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_e(i) = \underset{e}{\sigma_{\approx}}(i)$$

Proof. Let us prove both inclusions:

$\subseteq$ Let us consider $t \in \sigma_e(i)$ and let us reason by induction on the trace t:

- if $t = \epsilon$ then $t \in \sigma_e(i)$ implies that $i \downarrow$, which implies that $t = \epsilon \in \underset{e}{\sigma_{\approx}}(i)$
- if $t = a.t'$ then $t \in \sigma_e(i)$ implies the existence of $p \in front(i)$ such that $i_{|p} = a$ and $t' \in \sigma_e(\texttt{exe}(i,p))$. Then, we have:
  - as per Th.6.3, $t' \in \sigma_d(\texttt{exe}(i,p))$
  - as per Lem.6.7, $\texttt{exe}(i,p) \approx_{E_\mathbb{I}} \texttt{exe}_{\approx}(i,p)$
  - as per Lem.4.30, $\sigma_d(\texttt{exe}(i,p)) = \sigma_d(\texttt{exe}_{\approx}(i,p))$
  - as per Th.6.3, $t' \in \sigma_e(\texttt{exe}_{\approx}(i,p))$

  We can apply the induction hypothesis on t', which implies that $t' \in \underset{e}{\sigma_{\approx}}(\texttt{exe}_{\approx}(i,p))$ and therefore $t = a.t' \in \underset{e}{\sigma_{\approx}}(i)$

$\supseteq$ Let us consider $t \in \underset{e}{\sigma_{\approx}}(i)$ and let us reason by induction on the trace t:

- if $t = \epsilon$ then $t \in \underset{e}{\sigma_{\approx}}(i)$ implies that $i \downarrow$, which implies that $t = \epsilon \in \sigma_e(i)$
- if $t = a.t'$ then $t \in \underset{e}{\sigma_{\approx}}(i)$ implies the existence of $p \in front(i)$ such that $i_{|p} = a$ and $t' \in \underset{e}{\sigma_{\approx}}(\texttt{exe}_{\approx}(i,p))$. We can apply the induction hypothesis on t' which implies that $t' \in \sigma_e(\texttt{exe}_{\approx}(i,p))$ Then, we have:

* as per Th.6.3, $t' \in \sigma_d(\texttt{exe}_{\approx}(i,p))$
* as per Lem.6.7, $\texttt{exe}_{\approx}(i,p) \approx_{E_{\mathbb{I}}} \texttt{exe}(i,p)$
* as per Lem.4.30, $\sigma_d(\texttt{exe}_{\approx}(i,p)) = \sigma_d(\texttt{exe}(i,p))$
* as per Th.6.3, $t' \in \sigma_e(\texttt{exe}(i,p))$

Therefore $t = a.t' \in \sigma_e(i)$

□

We have therefore defined a "simplifying" execution semantics which guarantees that we won't have useless $\varnothing$ nodes that accumulate during execution. This $\sigma_{\underset{e}{\approx}}$ semantics makes use of simplifications that are incorporated in the inductive definitions of the prune and execution functions. We argue that this semantics might have less overhead costs than the "normalizing" execution semantics σ_n from the previous section. However, unlike σ_n, which guarantees that each intermediate term is a normal form, $\sigma_{\underset{e}{\approx}}$ offers no such guarantee.

Conclusion

In this chapter we have redefined the operational semantics from Chap.5 in the style of functional programming languages and with the added benefit of being able to separate concerns between the evaluation of which actions are immediately executable (the frontier of execution) and the computation the the follow-up interaction issued from the execution of a specific action at a specific position (the execution function). Separating those concerns allows to reduce overhead whenever we do not need to compute all follow-up interactions. For instance we may only want to compute follow-ups resulting from executions of frontier actions that match a certain action (there can be several occurrences at distinct positions). We have proven this algorithmicized "execution semantics" to be equivalent to the previously defined trace semantics. In addition, to further reduce potential overheads, we defined two variants of the execution semantics: a "normalizing execution semantics" and a "simplifying execution semantics" which respectively use results from equational logic and term rewriting from Chap.4 in order to guarantee working with intermediate interaction terms (that are computed during the exploration of the execution tree) that are kept simple and compact.

Given that all the previously defined trace semantics (σ_d from Chap.4, σ_o from Chap.5 and σ_e, σ_n and $\sigma_e{\approx}$ from this chapter) have been proven to be equivalent, in the remainder of this manuscript, we will simply use the notation σ to refer to the trace semantics of interactions.

Let us remark that the semantics that is implemented in the HIBOU tool presented in Chap.12 is $\sigma_{\underset{e}{\approx}}$. However, in the remainder of this manuscript, we will rather use notations related to σ_o (and sometimes σ_d) when presenting the formal verification algorithms that we have build on top of the semantics of interactions.

In the next chapter we extend the notion of the semantics of interactions to objects that are not traces but "multi-traces".

Chapter 7

Multi-trace semantics

Contents

In this chapter we define multi-traces, which are sets of local traces, each defined on a subset of lifelines. We then explain how one can extend the semantics of interactions to those objects.

The plan of this chapter is as follows:

- in Sec.7.1 we define multi-traces and algebraic operators on multi-traces,
- in Sec.7.2 we explain how we can relate traces to multi-traces via projection,
- in Sec.7.3 we define several semantics of "accepted" multi-traces,
- in Sec.7.4 we define various notions of prefixes of multi-traces and extend the multi-trace semantics to those notions of prefixes.

7.1 Multi-traces up to a partition

7.1.1 Defining multi-traces

Preliminaries

For any set X, we will note $\mathrm{Part}(X)$ the set of its partitions. A partition $C \in \mathrm{Part}(X)$ is defined as a collection $C = (c_k)_{k\in[1,n]}$ of cardinal $n \in \mathbb{N}^+$ such that:

- we have $\bigcup_{k\in[1,n]} c_k = X$
- and for any k and j in $[1,n]$ with $k \neq j$, we have $c_k \cap c_j = \emptyset$

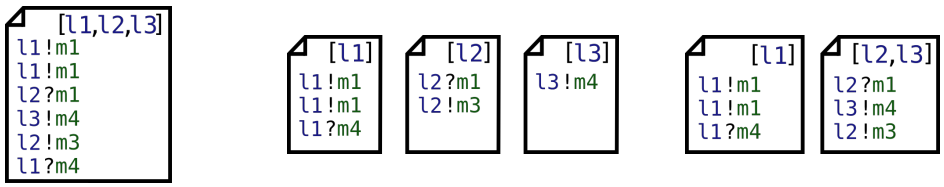
We may denote by $\check{X}$ the discrete partition $\check{X} = (\{x\})_{x\in X}$.

In the following, let us consider a given signature $\Omega = (L, M)$. In Sec.4.1, we have defined $\mathbb{T}_\Omega$ as the set of "traces" which are as sequences of actions from $\mathbb{A}_\Omega$. Those traces are therefore global traces [93], which order events occurring on every lifeline globally.

Generalized multi-traces

For the sake of practicity, in the following, for any subset of lifelines $c \in \mathcal{P}(L)$, we will denote by $\mathbb{A}_{\Omega|c}$ the set of actions occurring on c i.e. we have $\mathbb{A}_{\Omega|c} = \{a \in \mathbb{A}_\Omega \mid \theta(a) \in c\}$. We can then denote by $\mathbb{T}_{\Omega|c}$ the set $\mathbb{A}^*_{\Omega|c}$ of sequences of such actions.

We can then define multi-traces up to a partition C for any partition $C \in \mathrm{Part}(L)$, as collections of component traces $(t_c)_{c\in C}$ such that for any $c \in C$, we have $t_c \in \mathbb{T}_{\Omega|c}$. Such a multi-trace can be understood as a set of component traces, each defined on a "co-localization" of several lifelines. Within each such co-localization, events can be reordered even though they may occur on different lifelines. We can hence assume, for instance, that sub-systems within the same co-localization share a common clock.



global trace
up to the trivial partition
$C = (L) = (\{l_1, l_2, l_3\})$

(classical) multi-trace
up to the discrete partition
$C = (\{l\})_{l \in L} = (\{l_1\}, \{l_2\}, \{l_3\})$

(generalized) multi-trace
up to a specific partition
$C = (\{l_1\}, \{l_2, l_3\})$

Figure 7.1: Explicitation of various notions of traces and multi-traces

Definition 7.1: Multi-traces up to a partition

For any signature $\Omega = (L, M)$ and for any partition $C \in \mathrm{Part}(L)$, we define:

$$\mathbb{T}_{\Omega|C} = \prod_{c \in C} \mathbb{T}_{\Omega|c}$$

With, for any $c \subset L$:

$$\mathbb{T}_{\Omega|c} = \mathbb{A}^*_{\Omega|c} \quad \text{and} \quad \mathbb{A}_{\Omega|c} = \{a \in \mathbb{A}_\Omega \mid \theta(a) \in c\}$$

Inclusion of global traces & classical multi-traces in the new definition

The definition of generalized multi-traces in Def.7.1 encompasses that of both global traces (from Chap.4) and "classical" multi-traces, defined such that each lifeline has its own component in the multi-trace as in [94].

If we consider the trivial partition $C = (L)$ of L, then we have $\mathbb{T}_{\Omega|(L)} = \mathbb{T}_\Omega$ and we recognise the definition of global traces. Indeed, a global trace is a generalized multi-trace in which there is a single co-localization which includes all the lifelines.

If we consider the discrete partition $\check{L} = (\{l\})_{l \in L}$ of L, then we recognize in $\mathbb{T}_{\Omega|\check{L}}$ the definition of classical multi-traces (as per [94]). Indeed, every co-localization is reduced to a singleton, and reciprocally, every lifeline is associated with its own trace component.

In Fig.7.1 are represented three multi-traces that are defined up to different partitions. All three multi-traces describe the same execution of a given distributed system. However the level of detail regarding that description may vary in so far as the information regarding the ordering between events is of unequal quality:

- The multi-trace which has the most information is the one on the left, which corresponds to a global trace. Indeed, in that case, all lifelines are co-localized and as a result, the information regarding the order in which all events occurred globally is present. This multi-trace corresponds to the case in which the partition according to which it is defined is the trivial partition.

- The multi-trace which has the least amount of information about the execution of the distributed system is the one in the middle. This case corresponds to the one of classical multi-traces in which there

is no information regarding the relative order in which events occurring on different lifelines occurred during the execution. This multi-trace corresponds to the case in which the partition according to which it is defined is the discrete partition.

- The multi-trace on the right is an intermediate case in which some information is gathered and some is lost. It corresponds to a case in which the partition is neither the trivial partition nor the discrete partition.

Notations

For any partition C of L, we will use the greek letter μ to designate a multi-trace from $\mathbb{T}_{\Omega|C}$. And for any $c \in C$, we will denote by $\mu_{|c}$ the trace component of μ corresponding to the co-localization c. We will also use a substitution notation $\mu[t]_c$ to designate the multi-trace μ in which the component for co-localization c has been replaced by $t \in \mathbb{T}_{\Omega|c}$. Additionally, we may denote by ϵ_C the empty multi-trace $(\epsilon)_{c\in C} \in \mathbb{T}_{\Omega|C}$.

Given a lifeline $l \in L$, there exists a unique co-localization $c \in C$ such that $l \in c$. We denote by $\theta_C(l)$ this unique co-localization. We extend this notation to any action a so that $\theta_C(a) = \theta_C(\theta(a))$ to designate the unique co-localization in which action a occurs.

7.1.2 The $\mathcal{F}$-algebra of multi-traces up to a partition

In Chap.4, we have seen that the set of sets of traces $\mathcal{P}(\mathbb{T}_\Omega)$, when fitted with the following set of operation symbols $\mathbb{A}_\Omega \cup \{\{\epsilon\}, \cup, ;, ,, ;_*, ||, \overset{*}{,}, \overset{*}{;_*}, \overset{*}{,_*}, \overset{*}{||}\}$ has the structure of a $\mathcal{F}$-algebra w.r.t. the term algebra $\mathbb{I}_\Omega$ of interactions. By analogy, we can define similar operations on the set of sets of multi-traces up to a partition $\mathcal{P}(\mathbb{T}_{\Omega|C})$.

In the following, given a partition $C \in \text{Part}(L)$ we introduce algebraic operators on $\mathcal{P}(\mathbb{T}_{\Omega|C})$ so as to define the $\mathcal{F}$-algebra of multi-traces.

Concatenation of multi-traces

At first, we need to define some finer notion of concatenation (compared to the classical ".") that acts on multi-traces. We do so in Def.7.2.

Definition 7.2: Appending actions on multi-traces

We define the:

- left concatenation law $\overset{\rightarrow}{\odot_C} : \mathbb{A}_\Omega \times \mathbb{T}_{\Omega|C} \to \mathbb{T}_{\Omega|C}$
- and the right concatenation law $\overset{\leftarrow}{\odot_C} : \mathbb{T}_{\Omega|C} \times \mathbb{A}_\Omega \to \mathbb{T}_{\Omega|C}$

such that for any action $a \in \mathbb{A}$ and any $\mu \in \mathbb{T}_{\Omega|C}$ we have:

$$a \overset{\rightarrow}{\odot_C} \mu = \mu[a.\mu_{|\theta_C(a)}]_{\theta_C(a)} \qquad \text{and} \qquad \mu \overset{\leftarrow}{\odot_C} a = \mu[\mu_{|\theta_C(a)}.a]_{\theta_C(a)}$$

Those concatenation laws respectively correspond to appending action a either to the left or to the right of the trace component $\mu_{|\theta_C(a)}$ corresponding to the co-localization on which a occurs. Let us remark that we then have $|\mu \overleftarrow{\odot_C} a| = |a \overrightarrow{\odot_C} \mu| = |\mu| + 1$. Also, $\overrightarrow{\odot_C}$ and $\overleftarrow{\odot_C}$ display some interesting properties which are similar (although weaker) than the ones of the classical "." concatenation.

We may extend $\overrightarrow{\odot_C}$ and $\overleftarrow{\odot_C}$ to sets of multi-traces such that for any action $a \in \mathbb{A}_\Omega$ and any set T of multi-traces in $\mathcal{P}(\mathbb{T}_{\Omega|C})$ we have:

$$a \overrightarrow{\odot_C} T = \{a \overrightarrow{\odot_C} \mu \mid \mu \in T\} \qquad \text{and} \qquad T \overleftarrow{\odot_C} a = \{\mu \overleftarrow{\odot_C} a \mid \mu \in T\}$$

Using the left and right concatenation laws, we define inductively the concatenation of two multi-traces.

Definition 7.3: Concatenation of multi-traces

We define $\odot_C : \mathbb{T}_{\Omega|C} \times \mathbb{T}_{\Omega|C} \to \mathbb{T}_{\Omega|C}$ so that for any two multi-traces μ_1 and μ_2:

- if $\mu_2 = \epsilon_C$ then $\mu_1 \odot_C \mu_2 = \mu_1$
- if $\mu_2 = a \overrightarrow{\odot_C} \mu_2'$ then $\mu_1 \odot_C \mu_2 = (\mu_1 \overleftarrow{\odot_C} a) \odot_C \mu_2'$

We then extend it to sets of multi-traces with $\odot_C : \mathcal{P}(\mathbb{T}_{\Omega|C}) \times \mathcal{P}(\mathbb{T}_{\Omega|C}) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ so that for any two sets of multi-traces T_1 and T_2:

$$T_1 \odot_C T_2 = \bigcup_{\substack{\mu_1 \in T_1 \\ \mu_2 \in T_2}} \{\mu_1 \odot_C \mu_2\}$$

The $\odot_C$ operator is trivially associative so that for any three multi-traces μ_1, μ_2 and μ_3, we have $\mu_1 \odot_C (\mu_2 \odot_C \mu_3) = (\mu_1 \odot_C \mu_2) \odot_C \mu_3$, which we may simply denote by $\mu_1 \odot_C \mu_2 \odot_C \mu_3$.

Strict sequencing for multi-traces

In Def.7.4, we define a strict sequencing operator $\fatsemi_C$ on multi-trace via the application of the strict sequencing operator ; on each local trace component.

Definition 7.4: Strict sequencing for multi-traces

We define $\fatsemi_C : \mathbb{T}_{\Omega|C} \times \mathbb{T}_{\Omega|C} \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ so that for any two of multi-traces μ_1 and μ_2:

$$\mu_1 \fatsemi_C \mu_2 = \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} \in (\mu_{1|c}\ ;\ \mu_{2|c})\}$$

We then extend it to sets of multi-traces with $\fatsemi_C : \mathcal{P}(\mathbb{T}_{\Omega|C}) \times \mathcal{P}(\mathbb{T}_{\Omega|C}) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ such that for any two sets of multi-traces T_1 and T_2:

$$T_1 \fatsemi_C T_2 = \bigcup_{\substack{\mu_1 \in T_1 \\ \mu_2 \in T_2}} (\mu_1 \fatsemi_C \mu_2)$$

We can then show that the operator that is thus defined describes the same notion of concatenation than the $\odot_C$ operator. However, while $\odot_C$ is defined inductively in the world of multi-traces via the successive appending and removing of actions using $\overrightarrow{\odot}_C$ and $\overleftarrow{\odot}_C$, the $⦂_C$ operator is defined by reconstructing individually every local component of the multi-trace using ";".

Lemma 7.1: Strict sequencing on multi-traces is multi-trace concatenation

For any two multi-traces μ_1 and μ_2:

$$\mu_1 ⦂_C \mu_2 = \{\mu_1 \odot_C \mu_2\} \qquad \text{i.e.} \qquad (\mu_{1|c} \ ; \ \mu_{2|c})_{c\in C} = \mu_1 \odot_C \mu_2$$

Proof. Let us reason by induction on μ_2:

- for $\mu_2 = \epsilon_C$ we have:

$$\begin{aligned}(\mu_{1|c} \ ; \ \epsilon_{C|c})_{c\in C} &= (\mu_{1|c} \ ; \ \epsilon)_{c\in C} \\ &= (\mu_{1|c})_{c\in C} \\ &= \mu_1 \\ &= \mu_1 \odot_C \epsilon_C\end{aligned}$$

- for $\mu_2 = a \overrightarrow{\odot}_C \mu_2' = a \overrightarrow{\odot}_C (\epsilon_C \odot_C \mu_2') = (a \overrightarrow{\odot}_C \epsilon_C) \odot_C \mu_2' = \epsilon_C[a]_{\theta_C(a)} \odot_C \mu_2'$ we have:

$$\begin{aligned}(\mu_{1|c} \ ; \ (\epsilon_C[a]_{\theta_C(a)} \odot_C \mu_2')_{|c})_{c\in C} &= ((\mu_{1|c} \ ; \ (\epsilon_C[a]_{\theta_C(a)})_{|c}) \ ; \ \mu'_{2|c})_{c\in C} \\ &= (\mu_{1|c} \ ; \ (\epsilon_C[a]_{\theta_C(a)})_{|c})_{c\in C} \odot_C (\mu'_{2|c})_{c\in C} \\ &= ((\mu_1[\mu_{1|\theta_C(a)}.a]_{\theta_C(a)})_{|c})_{c\in C} \odot_C \mu_2' \\ &= \mu_1[\mu_{1|\theta_C(a)}.a]_{\theta_C(a)} \odot_C \mu_2' \\ &= (\mu_1 \overleftarrow{\odot}_C a) \odot_C \mu_2' \\ &= \mu_1 \odot_C (a \overrightarrow{\odot}_C \mu_2')\end{aligned}$$

□

As a result, even though they are not defined in the same manner, both the $\odot_C$ and $⦂_C$ operators describe the same notion of concatenating multi-traces. In the following we may therefore use them interchangeably. The notation $⦂_C$ can be used to keep in line with the notations from Chap.4 which are inspired from [75] and to clearly relate this symbol to the notion of strict sequencing. The notation $\odot_C$ can be used more trivially to manipulate multi-traces in a more general context.

However, having at disposal both definitions i.e. both manners of understanding multi-trace concatenation (from a global or a local point of view) can have some practical uses:

- We may notably use the inductive definition of $\odot_C$ in proofs or to define prefixes of multi-traces.
- The definition of $⦂_C$, which makes use of ; can be used to relate both operators.

Interleaving on multi-traces

We also extend interleaving to multi-traces with $⫴_C$ corresponding to the application of the interleaving $||$ on each local component.

Definition 7.5: Interleaving for multi-traces

We define $⫴_C : \mathbb{T}_{\Omega|C} \times \mathbb{T}_{\Omega|C} \to \mathbb{T}_{\Omega|C}$ so that for any two of multi-traces μ_1 and μ_2:

$$\mu_1 ⫴_C \mu_2 = \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} \in (\mu_{1|c} \,||\, \mu_{2|c})\}$$

We then extend it to sets of multi-traces with $⫴_C : \mathcal{P}(\mathbb{T}_{\Omega|C}) \times \mathcal{P}(\mathbb{T}_{\Omega|C}) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ such that for any two sets of multi-traces T_1 and T_2:

$$T_1 ⫴_C T_2 = \bigcup_{\substack{\mu_1 \in T_1 \\ \mu_2 \in T_2}} (\mu_1 ⫴_C \mu_2)$$

It is notable that we can characterize $⫴_C$ as is done in Lem.7.2.

Lemma 7.2: Characterization of interleaving on multi-traces

For any μ_1 and μ_2 in $\mathbb{T}_{\Omega|C}$ and any a_1 and a_2 in $\mathbb{A}_\Omega$:

$$\begin{aligned}
\epsilon_C ⫴_C \mu_2 &= \{\mu_2\} \\
\mu_1 ⫴_C \epsilon_C &= \{\mu_1\} \\
(a_1 \overrightarrow{\odot}_C \mu_1) ⫴_C (a_2 \overrightarrow{\odot}_C \mu_2) &= \left(\begin{array}{l} \{a_1 \overrightarrow{\odot}_C \mu \mid \mu \in \mu_1 ⫴_C (a_2 \overrightarrow{\odot}_C \mu_2)\} \\ \cup\, \{a_2 \overrightarrow{\odot}_C \mu \mid \mu \in (a_1 \overrightarrow{\odot}_C \mu_1) ⫴_C \mu_2\} \end{array} \right)
\end{aligned}$$

Proof. Let us prove each point:

- we have that:

$$\begin{aligned}
\epsilon_C ⫴_C \mu_2 &= \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} \in ((\epsilon_C)_{|c} \,||\, \mu_{2|c})\} \\
&= \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} \in (\epsilon \,||\, \mu_{2|c})\} \\
&= \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} = \mu_{2|c}\} \\
&= \{\mu \in \mathbb{T}_{\Omega|C} \mid \mu = \mu_2\} = \{\mu_2\}
\end{aligned}$$

- likewise $\mu_1 ⫴_C \epsilon_C = \{\mu_1\}$

- we have:

$$(a_1 \overrightarrow{\odot}_C \mu_1) ⫴_C (a_2 \overrightarrow{\odot}_C \mu_2) = \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\, c \in C,\ \mu_{|c} \in ((a_1 \overrightarrow{\odot}_C \mu_1)_{|c} \,||\, (a_2 \overrightarrow{\odot}_C \mu_2)_{|c})\}$$

then:

– if $\theta_C(a_1) = \theta_C(a_2) = c_a$:

$$
\begin{array}{rcl}
T & = & (a_1 \overset{\rightarrow}{\odot_C} \mu_1) \between_C (a_2 \overset{\rightarrow}{\odot_C} \mu_2) \\
& = & \left\{ \mu \in \mathbb{T}_{\Omega|C} \;\middle|\; \begin{array}{ll} \forall\ c \in C \setminus \{c_a\}, & \mu_{|c} \in \mu_{1|c} \mid\mid \mu_{2|c} \\ & \mu_{|c_a} \in a_1.\mu_{1|c_a} \mid\mid a_2.\mu_{2|c_a} \end{array} \right\} \\
& = & \left\{ \mu \in \mathbb{T}_{\Omega|C} \;\middle|\; \begin{array}{ll} \forall\ c \in C \setminus \{c_a\}, & \mu_{|c} \in \mu_{1|c} \mid\mid \mu_{2|c} \\ & \mu_{|c_a} \in \left(\begin{array}{l} \{a_1.t \mid t \in \mu_{1|c_a} \mid\mid a_2.\mu_{2|c_a}\} \\ \cup\, \{a_2.t \mid t \in a_1.\mu_{1|c_a} \mid\mid \mu_{2|c_a}\} \end{array} \right) \end{array} \right\} \\
& = & \left\{ \mu \in \mathbb{T}_{\Omega|C} \;\middle|\; \forall\ c \in C, \mu_{|c} \in \left(\begin{array}{l} \{(a_1 \overset{\rightarrow}{\odot_C} \mu)_c \mid \mu \in \mu_1 \ \between\ a_2 \overset{\rightarrow}{\odot_C} \mu_2\} \\ \cup\, \{(a_2 \overset{\rightarrow}{\odot_C} \mu)_c \mid \mu \in a_1 \overset{\rightarrow}{\odot_C} \mu_1 \ \between\ \mu_2\} \end{array} \right) \right\} \\
& = & \left(\begin{array}{l} \{a_1 \overset{\rightarrow}{\odot_C} \mu \mid \mu \in \mu_1 \between_C (a_2 \overset{\rightarrow}{\odot_C} \mu_2)\} \\ \cup\, \{a_2 \overset{\rightarrow}{\odot_C} \mu \mid \mu \in (a_1 \overset{\rightarrow}{\odot_C} \mu_1) \between_C \mu_2\} \end{array} \right)
\end{array}
$$

– the case for $\theta_C(a_1) \neq \theta_C(a_2)$ can be proven similarly

□

Weak sequencing on multi-traces

We also define weak sequencing $\circledast_C$ on multi-traces via the application of the weak sequencing $;_*$ on each local trace component.

Definition 7.6: Weak sequencing for multi-traces

We define $\circledast_C : \mathbb{T}_{\Omega|C} \times \mathbb{T}_{\Omega|C} \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ so that for any two of multi-traces μ_1 and μ_2:

$$\mu_1 \circledast_C \mu_2 = \{\mu \in \mathbb{T}_{\Omega|C} \mid \forall\ c \in C,\ \mu_{|c} \in (\mu_{1|c} ;_*\ \mu_{2|c})\}$$

We then extend it to sets of multi-traces with $\circledast_C : \mathcal{P}(\mathbb{T}_{\Omega|C}) \times \mathcal{P}(\mathbb{T}_{\Omega|C}) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ such that for any two sets of multi-traces T_1 and T_2:

$$T_1 \circledast_C T_2 = \bigcup_{\substack{\mu_1 \in T_1 \\ \mu_2 \in T_2}} (\mu_1 \circledast_C \mu_2)$$

However we do not provide any characterization for this operator.

Closures of scheduling operators on multi-traces

By analogy to the results from Chap.4, let us also define the three Kleene closures for the $\fatsemi_C$, $\circledast_C$ and $\between_C$ operators.

Definition 7.7: Kleene closures for multi-traces

For any scheduling operator $\circledcirc \in \{⨾_C,\ ⊛_C,\ ⫴_C\}$, we define, for any set of multi-traces $T \in \mathcal{P}(\mathbb{T}_{\Omega|C})$:

$$\begin{aligned} T^{\circledcirc 0} &= \{\epsilon_C\} \\ T^{\circledcirc j} &= T \circledcirc T^{\circledcirc (j-1)} \quad \textbf{for } j > 0 \\ T^{\circledcirc *} &= \bigcup_{j\in\mathbb{N}} T^{\circledcirc j} \end{aligned}$$

We have seen in Chap.4 that the Head-First closure of weak sequencing is uniquely defined. As a result, let us also define an analogue of this operator on multi-traces.

Definition 7.8: Weak HF-closure for multi-traces

We define $⊛^{\curvearrowleft}_C : \mathcal{P}(\mathbb{T}_{\Omega|C}) \times \mathcal{P}(\mathbb{T}_{\Omega|C}) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ s.t. for any two sets of multi-traces T_1 and T_2:

$$T_1 ⊛^{\curvearrowleft}_C T_2 = \left\{ \mu \in T_1 ⊛_C T_2 \;\middle|\; (\mu = a \overrightarrow{\odot}_C \mu') \Rightarrow \left(\exists\, \mu_1 \in \mathbb{T}_{\Omega|C} \text{ s.t. } \begin{array}{l} (a \overrightarrow{\odot}_C \mu_1 \in T_1) \\ \wedge\ (\mu' \in \{\mu_1\} ⊛_C T_2) \end{array} \right) \right\}$$

We then define $⊛^{\curvearrowleft *}_C$ as the K-closure of $⊛^{\curvearrowleft}_C$.

Algebraic considerations

Similarly to the operators on traces from Chap.4, it follows that the previously defined operators on multi-traces present some interesting algebraic properties. For instance, those properties include, among others, that: $⨾_C$ and $⊛_C$ are associative or that $⫴_C$ is associative commutative.

We will not enter into the details here but, in fact, all of the properties defined in Chap.4 for operators on traces also apply to their respective counterparts on multi-traces.

The set of sets of multi-traces $\mathcal{P}(\mathbb{T}_{\Omega|C})$ then admits the structure of a $\mathcal{F}$-algebra as defined in Def.7.9.

Definition 7.9: Semantic domain $\mathcal{P}(\mathbb{T}_{\Omega|C})$ as a $\mathcal{F}$-algebra

$\mathcal{A}_C = (\mathcal{P}(\mathbb{T}_{\Omega|C}),\ \mathcal{F}^{\mathcal{A}_C}$ (with $\mathcal{F}^{\mathcal{A}_C} = \{f^{\mathcal{A}_C} \mid f \in \mathcal{F}\}$) is the $\mathcal{F}$-algebra of carrier $\mathcal{P}(\mathbb{T}_{\Omega|C})$ and the following interpretations of the operation symbols in $\mathcal{F}$:

$$\begin{array}{lcl} \varnothing^{\mathcal{A}_C} &=& \{\epsilon_C\} \\ a^{\mathcal{A}_C} &=& \{a \overrightarrow{\odot}_C \epsilon_C\} \end{array} \qquad \begin{array}{lcl} strict^{\mathcal{A}_C} &=& ⨾_C \\ seq^{\mathcal{A}_C} &=& ⊛_C \\ par^{\mathcal{A}_C} &=& ⫴_C \\ alt^{\mathcal{A}_C} &=& \cup \end{array} \qquad \begin{array}{lcl} loop_S^{\mathcal{A}_C} &=& ⨾_{C*} \\ loop_H^{\mathcal{A}_C} &=& ⊛^{\curvearrowleft}_{C*} \\ loop_W^{\mathcal{A}_C} &=& ⊛_{C*} \\ loop_P^{\mathcal{A}_C} &=& ⫵_{C*} \end{array}$$

The case of the trivial partition

In the case of the trivial partition $C = \{L\}$ i.e. when we have a single co-localization which gathers all lifelines then $\mathcal{A}_C$ is isomorph w.r.t. $\mathcal{A}$ i.e. we have a bijective homomorphism between the two $\mathcal{F}$-algebras. For all intents and purposes manipulating multi-traces defined on the trivial partition is then equivalent to manipulating global traces.

The case of the discrete partition

In the case of the discrete partition $\check{L}$ of lifelines, the $\circledast_{\check{L}}$ and $⦂_{\check{L}}$ operators behave identically as stated in Lem.7.3.

Lemma 7.3: Equivalence of multi-trace weak and strict sequencing on the discrete partition

For any signature $\Omega = (L, M)$, for any μ_1 and μ_2 in $\mathbb{T}_{\Omega|\check{L}}$ we have:

$$\mu_1 \circledast_{\check{L}} \mu_2 = \mu_1 ⦂_{\check{L}} \mu_2$$

Proof. Indeed:

- in any given co-localization $c = \{l\} \in \check{L}$, for any t_1 and t_2 from $\mathbb{T}_{\Omega|\{l\}}$ we have $t_1 ;_* t_2 = t_1 ; t_2$ given that it is not possible to interleave any actions using weak sequence because they all occur on the same lifeline.
- which implies that:

$$\begin{aligned} \mu_1 \circledast \mu_2 &= \{\mu \in \mathbb{T}_{\Omega|\check{L}} \mid \forall\, l \in L,\ \mu_{|\{l\}} \in (\mu_{1|\{l\}} ;_* \mu_{2|\{l\}})\} \\ &= \{\mu \in \mathbb{T}_{\Omega|\check{L}} \mid \forall\, l \in L,\ \mu_{|\{l\}} \in (\mu_{1|\{l\}} ; \mu_{2|\{l\}})\} \\ &= \mu_1 ⦂ \mu_2 \end{aligned}$$

□

This last result can be extended to conclude that the strict and weak K-closures and the weak HF-closure also behave identically on the discrete partition (Lem.7.4).

Lemma 7.4: Equivalent repetition operators on the discrete partition

For any signature $\Omega = (L, M)$, for any set of multi-traces $T \in \mathbb{T}_{\Omega|\check{L}}$ up to the discrete partition:

$$T^{⦂_{\check{L}}*} = T^{\circledast^{\curvearrowleft}_{\check{L}}*} = T^{\circledast_{\check{L}}*}$$

Proof. Implied by Lem.7.3. □

This means that weak and strict sequencing are indistinguishable when observing multi-traces defined up to the discrete partition.

7.2 Projecting traces

The $\mathcal{F}$-algebras $\mathcal{A}$ of sets of traces and the $\mathcal{F}$-algebras $\mathcal{A}_C$ of sets of multi-traces up to a partition $C \in \mathrm{Part}(L)$ have similar algebraic structures.

In this section we discuss the possibility of relating those two $\mathcal{F}$-algebras via a projection operator which projects traces into multi-traces. We will see that depending on which is the partition C, this projection may or may not preserve algebraic structures.

7.2.1 The projection function

In Def.7.10 we define a $proj_C$ function for projecting global traces into multi-traces defined up to a partition C.

Definition 7.10: Trace Projection

For any signature $\Omega = (L, M)$ and any partition $C \in \mathrm{Part}(L)$, we define $\mathbf{proj}_C : \mathbb{T}_\Omega \to \mathbb{T}_{\Omega|C}$ such that:

- $\mathbf{proj}_C(\epsilon) = \epsilon_C$ the empty multi-trace $\epsilon_C = (\epsilon)_{c \in C}$
- for any $t \in \mathbb{T}_\Omega$ and any $a \in \mathbb{A}_\Omega$, if $\mathbf{proj}_C(t) = \mu$ then $\mathbf{proj}_C(a.t) = a \vec{\odot} \mu$
  i.e. given $c = \theta_C(a)$ the unique element of C such that $\theta(a) \in c$, we have $\mathbf{proj}_C(a.t) = \mu[a.\mu_{|c}]_c$

Let us consider the examples from Fig.7.1. For instance, let us consider the global trace on the left of Fig.7.1. In this example, we have $L = \{l_1, l_2, l_3\}$. If we project it on the discrete partition $C = (\{l_1\}, \{l_2\}, \{l_3\})$, we obtain the multi-trace in the middle of Fig.7.1. If we project it on the partition $C = (\{l_1\}, \{l_2, l_3\})$, we obtain the multi-trace in the right of Fig.7.1.

For any partition $C \in \mathrm{Par}(L)$, the projection function $\mathbf{proj}_C$ is always surjective but not necessarily injective. Indeed, several global traces can be projected into the same multi-trace. Let us for instance consider, given $L = \{l_1, l_2\}$ and $C = (\{l_1\}, \{l_2\})$, the global traces $t_\alpha = l_1!m.l_2!m$ and $t_\beta = l_2!m.l_1!m$. We then have $\mathbf{proj}_C(t_\alpha) = \mathbf{proj}_C(t_\beta) = (l_1!m,\ l_2!m)$. Given that multi-traces do not specify any order between events occurring on different co-localizations, the global order $l_1!m < l_2!m$ specified in t_α and the global order $l_2!m < l_1!m$ specified in t_β is not preserved after the projection as a multi-trace of $\mathbb{T}_{\Omega|C}$.

We extend the projection function to sets of traces in Def.7.11.

Definition 7.11: Extensions of trace projection

We extend $\mathbf{proj}_C$ to sets of traces as $\mathbf{proj}_C : \mathcal{P}(\mathbb{T}_\Omega) \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ such that for any set T of traces:

$$\mathbf{proj}_C(T) = \{\mathbf{proj}_C(t) \mid t \in T\}$$

7.2.2 Preservation of algebraic structures

As we hinted earlier, the projection function $\mathbf{proj}_C$ preserves some (but not all) algebraic structure of $\mathcal{A}$ into $\mathcal{A}_C$. In the following we explain which structures are preserved and which are not.

As stated in Lem.7.5, strict sequencing and interleaving are preserved by projection.

Lemma 7.5: Projection preserves strict sequencing & interleaving

For any traces t_1 and t_2:

$$\mathbf{proj}_C(t_1;t_2) = \mathbf{proj}_C(t_1) \fatsemi \mathbf{proj}_C(t_2) \qquad \mathbf{proj}_C(t_1||t_2) = \mathbf{proj}_C(t_1) \fatbslash\!\!\fatbslash \mathbf{proj}_C(t_2)$$

And hence, for any sets of traces T_1 and T_2:

$$\mathbf{proj}_C(T_1;T_2) = \mathbf{proj}_C(T_1) \fatsemi \mathbf{proj}_C(T_2) \qquad \mathbf{proj}_C(T_1||T_2) = \mathbf{proj}_C(T_1) \fatbslash\!\!\fatbslash \mathbf{proj}_C(T_2)$$

Proof. Let us consider traces t_1 and t_2.

◯ Let us start by the strict sequencing. We reason by induction on the trace t_1:

- when $t_1 = \epsilon$ we have:

$$\begin{aligned} \mathbf{proj}_C(\epsilon;t_2) &= \mathbf{proj}_C(t_2) \\ &= \epsilon_C \fatsemi \mathbf{proj}_C(t_2) \\ &= \mathbf{proj}_C(\epsilon) \fatsemi \mathbf{proj}_C(t_2) \end{aligned}$$

- when $t_1 = a.t_1'$ we have:

$$\begin{aligned} \mathbf{proj}_C((a.t_1');t_2) &= \mathbf{proj}_C(\{a.t \mid t \in t_1';t_2\}) \\ &= \{\mathbf{proj}_C(a.t) \mid t \in t_1';t_2\} \\ &= \{a \overset{\rightarrow}{\odot} \mathbf{proj}_C(t) \mid t \in t_1';t_2\} \\ &= a \overset{\rightarrow}{\odot} \mathbf{proj}_C(t_1';t_2) \\ &= a \overset{\rightarrow}{\odot} (\mathbf{proj}_C(t_1') \fatsemi \mathbf{proj}_C(t_2)) && \text{induction hypothesis} \\ &= (a \overset{\rightarrow}{\odot} \mathbf{proj}_C(t_1')) \fatsemi \mathbf{proj}_C(t_2) \\ &= \mathbf{proj}_C(a.t_1') \fatsemi \mathbf{proj}_C(t_2) \end{aligned}$$

◯ Let us now consider the interleaving and let us reason by induction on both t_1 and t_2:

- when $t_1 = \epsilon$ we have:

$$\begin{aligned} \mathbf{proj}_C(\epsilon||t_2) &= \mathbf{proj}_C(t_2) \\ &= \epsilon_C \fatbslash\!\!\fatbslash \mathbf{proj}_C(t_2) \\ &= \mathbf{proj}_C(\epsilon) \fatbslash\!\!\fatbslash \mathbf{proj}_C(t_2) \end{aligned}$$

- when $t_2 = \epsilon$ we have:

$$\begin{aligned}\mathbf{proj}_C(t_1||\epsilon) &= \mathbf{proj}_C(t_1)\\ &= \mathbf{proj}_C(t_1) \between \epsilon_C\\ &= \mathbf{proj}_C(t_1) \between \mathbf{proj}_C(\epsilon)\end{aligned}$$

- when $t_1 = a_1.t_1'$ and $t_2 = a_2.t_2'$ we have:

$$\begin{aligned}\mathbf{proj}_C((a_1.t_1')||(a_2.t_2')) &= \mathbf{proj}_C(\{a_1.t \mid t \in t_1'||(a_2.t_2')\} \cup \{a_2.t \mid t \in (a_1.t_1')||t_2'\})\\ &= a_1 \vec{\odot}_C \mathbf{proj}_C(t_1'||(a_2.t_2')) \cup a_2 \vec{\odot}_C \mathbf{proj}_C((a_1.t_1')||t_2')\\ \text{(induction)} \quad &= a_1 \vec{\odot}_C (\mathbf{proj}_C(t_1') \between \mathbf{proj}_C(a_2.t_2')) \cup a_2 \vec{\odot}_C (\mathbf{proj}_C(a_1.t_1') \between \mathbf{proj}_C(t_2'))\\ &= \left(\begin{array}{l} a_1 \vec{\odot}_C (\mathbf{proj}_C(t_1') \between (a_2 \vec{\odot}_C \mathbf{proj}_C(t_2')))\\ \cup\ a_2 \vec{\odot}_C ((a_1 \vec{\odot}_C \mathbf{proj}_C(t_1')) \between \mathbf{proj}_C(t_2'))\end{array}\right)\\ \text{(Lem.7.2)} \quad &= (a_1 \vec{\odot}_C \mathbf{proj}_C(t_1')) \between (a_2 \vec{\odot}_C \mathbf{proj}_C(t_2'))\\ &= \mathbf{proj}_C(a_1.t_1') \between \mathbf{proj}_C(a_2.t_2')\end{aligned}$$

□

Given that the algebraic structures of strict sequencing and interleaving are preserved by projection, it comes that repetitions of those structures with their Kleene closures are also preserved by projection. We state this in Lem.7.6.

Lemma 7.6: Projection preserves strict and interleaving K-closures

For any set of traces T:

$$\mathbf{proj}_C(T^{;*}) = \mathbf{proj}_C(T)^{\dot{;}*} \qquad \mathbf{proj}_C(T^{||*}) = \mathbf{proj}_C(T)^{\between *}$$

Proof. Implied by Lem.7.5 □

However, weak sequencing is not preserved. It suffices to consider the following example, with $L = \{l_1, l_2, l_3\}$ and the partition $C = (\{l_1, l_2\},\ \{l_3\})$. Here we have:

$$\begin{aligned}\mathbf{proj}_C(\{l_1!m_1.l_3?m_1\};_* \{l_3!m_2.l_2?m_2\}) &= \mathbf{proj}_C(\{l_1!m_1.l_3?m_1.l_3!m_2.l_2?m_2\})\\ &= \left\{\left(\begin{array}{l} l_1!m_1.l_2?m_2,\\ l_3?m_1.l_3!m_2\end{array}\right)\right\}\end{aligned}$$

However:

$$\begin{aligned}\mathbf{proj}_C(\{l_1!m_1.l_3?m_1\}) \circledast_C \mathbf{proj}_C(\{l_3!m_2.l_2?m_2\}) &= \left\{\left(\begin{array}{l} l_1!m_1,\\ l_3?m_1\end{array}\right)\right\} \circledast_C \left\{\left(\begin{array}{l} l_2?m_2,\\ l_3!m_2\end{array}\right)\right\}\\ &= \left\{\left(\begin{array}{l} l_1!m_1.l_2?m_2,\\ l_3?m_1.l_3!m_2\end{array}\right), \left(\begin{array}{l} l_2?m_2.l_1!m_1,\\ l_3?m_1.l_3!m_2\end{array}\right)\right\}\end{aligned}$$

In the world of interactions, this example corresponds to the diagram given on Fig.7.2. We have indeed that the semantics of this interaction is the weak sequencing of $l_1!m_1.l_3?m_1$ which is the passing of message m_1 followed by $l_3!m_2.l_2?m_2$ which is the passing of message m_2. On co-localization $\{l_1, l_2\}$ we are therefore supposed to observe at first $l_1!m_1$ and then $l_2?m_2$. Observing the reception of m_2 i.e. $l_2?m_2$ at first is impossible because its emission must occur after the reception of m_1. However, when projecting onto multi-traces, the information pertaining to this chain of causality is lost, which is why in the second computation, using $\circledast_C$, the interleaving between $l_1!m_1$ and $l_2?m_2$ is authorized.

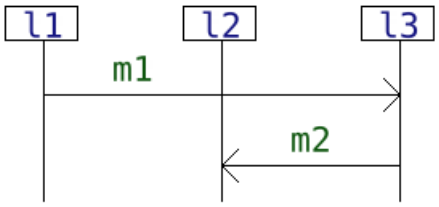


Figure 7.2: Counter example demonstrating that projection do not preserve weak sequencing

We generalize those remarks on the inclusion of projected weak sequences in Lem.7.7.

Lemma 7.7: Inclusion of projected weak sequences

For any traces t_1 and t_2, any sets of traces T_1, T_2 and T:

$$\mathbf{proj}_C(t_1 ;_* t_2) \subset \mathbf{proj}_C(t_1) \circledast \mathbf{proj}_C(t_2) \qquad \mathbf{proj}_C(T_1 ;_* T_2) \subset \mathbf{proj}_C(T_1) \circledast \mathbf{proj}_C(T_2)$$

$$\mathbf{proj}_C(T^{;_**}) \subset \mathbf{proj}_C(T)^{\circledast *} \qquad \mathbf{proj}_C(T^{;_*^{\urcorner}*}) = \mathbf{proj}_C(T)^{\circledast^{\urcorner} *}$$

Proof. We can reason in terms of which interleavings are allowed or not allowed. More formally, we can proceed by induction on traces t_1 and t_2 similarly as what we did for Lem.7.5. □

We can conclude that, in the general case of any partition $C \in \mathrm{Part}(L)$, the projection function $\mathbf{proj}_C$ is not a homomorphism between $\mathcal{A}$ and $\mathcal{A}_C$ because weak sequencing (and therefore its K-closure and HF-closure) is not preserved.

However, as we have seen with Lem.7.3 and Lem.7.4, in the case of the discrete partition, all the problematic operators related to weak sequencing behave like their strict counterpart. As a result, and in that case, $proj_{\check{L}}$ on the discrete partition is a homomorphism between $\mathcal{A}$ and $\mathcal{A}_{\check{L}}$. Also, and more trivially, in the case of the trivial partition $\{L\}$, $proj_{\{L\}}$ is also a homomorphism between $\mathcal{A}$ and $\mathcal{A}_{\{L\}}$. Those two remarks are illustrated on Fig.7.3.

7.3 Semantics of accepted multi-traces

In this section we define two different semantics of "accepted" multi-traces: an algebraic multi-trace semantics defined as a homomorphism between $\mathbb{I}_\Omega$ and $\mathcal{P}(\mathbb{T}_{\Omega|C})$ and a projected multi-trace semantics defined as the composition of $\mathbf{proj}_C$ and σ. We then explain how those two semantics can be related.

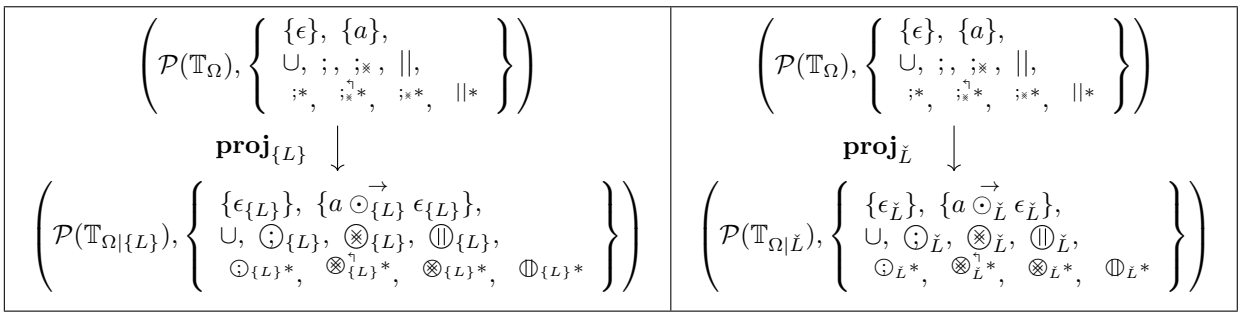


Figure 7.3: Extreme cases where $\mathbf{proj}_C$ is a homomorphism

7.3.1 Algebraic multi-trace semantics

In Chap.4 we defined the trace semantics σ in denotational-style semantics as a homomorphism between the term algebra $\mathbb{I}_\Omega$ and the $\mathcal{F}$-algebra $\mathcal{A}$ of sets of traces. In the same manner, we can define a multi-trace semantics $\circledcirc_C$ in denotational-style as a homomorphism between the term algebra $\mathbb{I}_\Omega$ and the $\mathcal{F}$-algebra $\mathcal{A}_C$ of sets of multi-traces.

Definition 7.12: Algebraic multi-trace semantics

$\circledcirc_C : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is defined as the unique $\mathcal{F}$-homomorphism between $\mathcal{T}_\mathcal{F} = \mathbb{I}_\Omega$ and $\mathcal{A}_C = (\mathcal{P}(\mathbb{T}_{\Omega|C}),\ \mathcal{F}^{\mathcal{A}_C})$

In the same manner as σ is the initial homomorphism between $\mathbb{I}_\Omega$ and $\mathcal{A}$, $\circledcirc_C$ is the initial homomorphism between $\mathbb{I}_\Omega$ and $\mathcal{A}_C$, as illustrated on Fig.7.4.

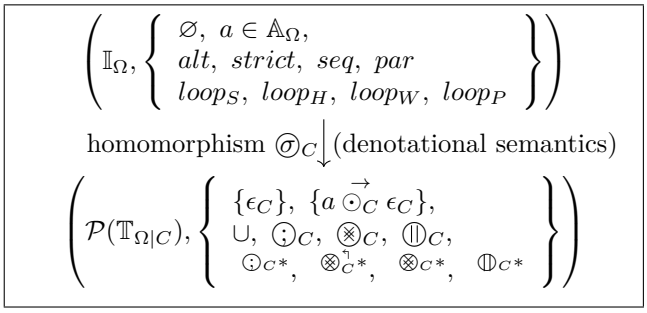


Figure 7.4: Algebraic multi-trace semantics

7.3.2 Multi-trace semantics by projection

By contrast, we can define another semantics of multi-traces, as the projection of the trace semantics. We define this semantics by projection in Def.7.13.

Definition 7.13: Projected multi-trace semantics

$\sigma_{|C} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is such that for any $i \in \mathbb{I}_\Omega$:

$$\sigma_{|C}(i) = \mathbf{proj}_C(\sigma(i))$$

This "projected" multi-trace semantics benefits from results that we have proven for the trace semantics. We notably have the property stated in Lem.7.8.

Lemma 7.8: A characterization of projected multi-trace semantics

For any interaction $i \in \mathbb{I}_\Omega$ we have that:

$$\forall\ (a, \mu', i'), \left(\begin{array}{c} (\mu' \in \sigma_{|C}(i')) \\ \wedge\ (i \xrightarrow{a} i') \end{array} \right) \Rightarrow (a \overrightarrow{\odot}_C \mu' \in \sigma_{|C}(i))$$

And:

$$\forall\ (\mu), \left(\begin{array}{c} (\mu \in \sigma_{|C}(i)) \\ \wedge\ (\mu \neq \epsilon_C) \end{array} \right) \Rightarrow \left(\begin{array}{cc} & (\mu = a \overrightarrow{\odot}_C \mu') \\ \exists\ (a, \mu', i'), & \wedge\ (i \xrightarrow{a} i') \\ & \wedge\ (\mu' \in \sigma_{|C}(i')) \end{array} \right)$$

Proof. Let us prove both properties:

- The fact that $\mu' \in \sigma_{|C}(i')$ implies the existence of a trace $t' \in \sigma(i')$ such that $\mathbf{proj}_C(t') = \mu'$. Then it comes that $i \xrightarrow{a} i'$ and $t' \in \sigma(i')$ implies $a.t' \in \sigma(i)$. Hence $\mathbf{proj}_C(a.t') = a \overrightarrow{\odot}_C \mu' \in \sigma_{|C}(i)$
- The fact that $\mu \in \sigma_{|C}(i)$ implies the existence of a trace $t \in \sigma(i)$ such that $\mathbf{proj}_C(t) = \mu$. Then, given that $\mu \neq \epsilon_C$ we have $t \neq \epsilon$ and hence there exists a and t' such that $t = a.t' \in \sigma(i)$. Therefore there exists i' such that $i \xrightarrow{a} i'$ and $t' \in \sigma(i')$. Then, let us denote by μ' the projection of t' i.e. $\mu' = \mathbf{proj}_C(t')$. We then have $\mathbf{proj}_C(t) = \mu = \mathbf{proj}_C(a.t') = a \overrightarrow{\odot}_C \mu'$ and $\mathbf{proj}_C(t') = \mu' \in \sigma_{|C}(i')$

□

7.3.3 Relating the algebraic and projected multi-trace semantics

We have seen in Sec.7.2.2 that the projection operator do not preserve the algebraic structure related to weak sequencing. In practice, we have that $@\sigma_C$ allows interleavings between actions that would otherwise be forbidden in $\sigma_{|C}$, as can be inferred from the counter-example given in Sec.7.2.2.

We therefore characterize, in the general case of any partition C, the relationship between $\sigma_{|C}$ and $@\sigma_C$ as an inclusion. We do so in Lem.7.9. The counter-example from Sec.7.2.2 proves that the reciprocate inclusion does not hold.

Lemma 7.9: Inclusion of the projected multi-trace semantics into the algebraic one

For any $i \in \mathbb{I}_\Omega$ we have:

$$\sigma_{|C}(i) \subset @\sigma_C(i)$$

Proof. We can simply use Lem.7.5, Lem.7.7 and Lem.7.6. Let us however detail the proof to illustrate the difference between the weak sequence and the other operators.

Let us use the denotational formulation of σ and reason by induction on the structure of interactions:

- for the empty interaction: $\sigma_{|C}(\varnothing) = \mathbf{proj}_C(\{\epsilon\}) = \{\epsilon_C\} = @_C(\varnothing)$
- for any action a: $\sigma_{|C}(a) = \mathbf{proj}_C(\{a\}) = \{a \overrightarrow{\odot}_C \epsilon_C\} = @_C(a)$
- for any $(f, \diamond, \circledcirc) \in \{(strict, ;, ⨾), (par, ||, ⫴), (alt, \cup, \cup)\}$ we have:

$$\begin{array}{rcll}
\sigma_{|C}(f(i_1,i_2)) & = & \mathbf{proj}_C(\sigma(f(i_1,i_2))) & \\
& = & \mathbf{proj}_C(\sigma(i_1) \diamond \sigma(i_2)) & \\
& = & \mathbf{proj}_C(\sigma(i_1)) \circledcirc \mathbf{proj}_C(\sigma(i_2)) & \text{as per Lem.7.5} \\
& = & \sigma_{|C}(i_1) \circledcirc \sigma_{|C}(i_2) & \\
& = & @_C(i_1) \circledcirc @_C(i_2) & \text{induction hypothesis} \\
& = & @_C(f(i_1,i_2)) &
\end{array}$$

- for the weak sequencing we have:

$$\begin{array}{rcll}
\sigma_{|C}(seq(i_1,i_2)) & = & \mathbf{proj}_C(\sigma(seq(i_1,i_2))) & \\
& = & \mathbf{proj}_C(\sigma(i_1) ;_* \sigma(i_2)) & \\
& \subset & \mathbf{proj}_C(\sigma(i_1)) \circledast \mathbf{proj}_C(\sigma(i_2)) & \text{as per Lem.7.7} \\
& \subset & \sigma_{|C}(i_1) \circledast \sigma_{|C}(i_2) & \\
& \subset & @_C(i_1) \circledast @_C(i_2) & \text{induction hypothesis} \\
& \subset & @_C(seq(i_1,i_2)) &
\end{array}$$

- for the repetition operators and constructors we can proceed similarly, using Lem.7.6 for the strict and interleaving K-closures and using Lem.7.7 for the weak K-closure and weak HF-closure

□

Pragmatically, the semantics that is "correct" is $\sigma_{|C}$. However, given that $@_C$ is a homomorphism while $\sigma_{|C}$ is not, it is easier to manipulate and find interesting results related to $@_C$. In particular, we might be interested in the two extreme cases of the trivial and discrete partitions, where, as we state in Th.7.1 both semantics coincide.

Theorem 7.1: When 2 multi-trace semantics coincide

We have that:

$$\sigma_{|\{L\}} = @_{\{L\}} \qquad \text{and} \qquad \sigma_{|\check{L}} = @_{\check{L}}$$

Proof. Using Lem.7.3 and Lem.7.4 to eliminate the operators related to weak sequencing. And then using Lem.7.5 and Lem.7.6. □

On Fig.7.5, we illustrate the relationships between the trace semantics σ, the projection function $\mathbf{proj}_C$ and the two multi-trace semantics $\sigma_{|C}$ and $@_C$. Let us recall that σ and $@_C$ are both homomorphisms i.e.

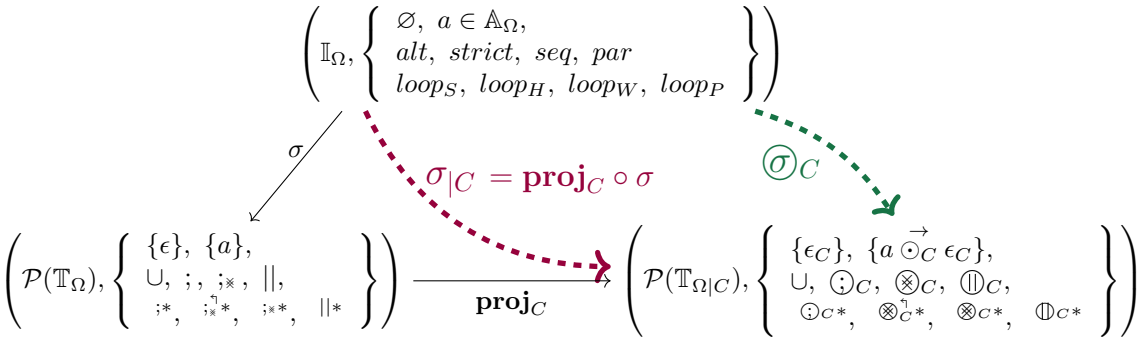


Figure 7.5: Algebraic and projective multi-trace semantics: the diagram commutes if $C = \{L\}$ or $C = \check{L}$

they preserve algebraic structures. However, $\mathbf{proj}_C$ and hence $\sigma_{|C}$ do not in the general case of any partition C. In the particular case of the trivial partition $C = \{L\}$ and the discrete partition $C = \check{L}$ however, $\mathbf{proj}_C$ is homomorphic, allowing us to have $\sigma_{|C} = \textcircled{\sigma}_C$ (as per Th.7.1) which translates into having the diagram from Fig.7.5 being commutative.

7.4 Prefixes and slices of multi-traces

During the process of recording the multi-trace, the observation of what occurs on each co-localization starts at a given time, which may not be exactly the same for each co-localization, and ends at a given time, which may also differ according to the co-localization. As a result, it may be so, that on different co-localizations, the observation starts and ends at different times. Moreover the span of time during which the observation occurs may not necessarily include the span of time during which observable events occur on a given co-localization. In practice those issues may arise due to the lack of a strict synchronization mechanism between local testers.

In any case, the occurrence of some events may not be recorded as a result. Indeed:

- if the observation starts too late on a given co-localization, some events occurring at the beginning of the execution on the corresponding sub-systems may not have been recorded
- if the observation ends too early on a given co-localization, some events occurring at the end of the execution on the corresponding sub-systems may not have been recorded

This notion of "partial" or "degraded" observation is illustrated on Fig.7.6 which represents different possible recordings of the same execution of a certain distributed system. In the context of this example, we have $\Omega = (L, M)$ with $L = \{l_1, l_2, l_3\}$ and $M = \{m_1, m_2, m_3, m_4\}$. We consider an execution of a distributed system constituted of the three sub-systems represented by lifelines l_1, l_2 and l_3. Lifelines l_2 and l_3 are grouped in the same co-localization and therefore events occurring on them can be reordered. As a result, the recorded multi-traces are expressed as elements of $\mathbb{T}_{\Omega|C}$ with $C = (\{l_1\}, \{l_2, l_3\})$.

In the right of Fig.7.6, four different recordings (as multi-traces) of the same execution of the example distributed system are illustrated. Those four examples are as follows:

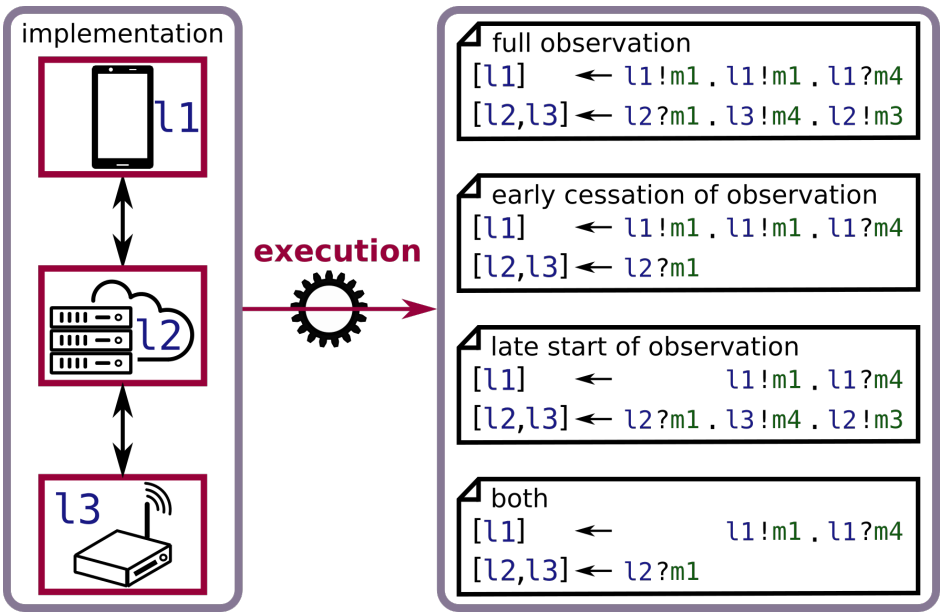


Figure 7.6: Recording multi-traces in different conditions of observability

- In ideal conditions, the full execution of the system is recorded, with no events missing. This is represented by the multi-trace at the top of Fig.7.6. On each of the two co-localization, three events were recorded.

- As explained earlier, it may be so that the observation or the recording of events may cease too early on any given co-localization. An example of such a case is given on the second multi-trace from the top in the right of Fig.7.6. We can see that the same execution of the distributed system have been observed but that two events have been missed at the end of the component trace corresponding to the co-localization $\{l_2, l_3\}$. The omission of those two events correspond to an early cessation of observation on the corresponding component.

- The third example from the top, in the right of Fig.7.6 corresponds to a case in which the same execution has been observed but the observation has begun too late on the component corresponding to lifeline l_1. Here, one event, at the beginning of the l_1 component has been missed.

- The bottom example of a multi-trace in the right of Fig.7.6 corresponds to a case in which we had both an early cessation of observation on component $\{l_2, l_3\}$ and a late start of observation on component $\{l_1\}$.

We can characterize the multi-traces obtained from recordings under degraded observability with regards to the multi-trace recorded under a full observability. This characterization consists in describing partially observed multi-traces as prefixes (in the sens of multi-traces) or slices of the fully observed multi-trace.

Let us at first introduce some preliminaries on the notion of prefixes, applied to globally defined traces.

7.4.1 Prefixes of global traces

The notion of prefix is tied to the existence of a concatenation law. In the case of global traces, the concatenation law is "." and we define prefixes as is classically done in Def.7.14.

Definition 7.14: Prefixes of traces

For any traces t, t_1 and t_2 from $\mathbb{T}_\Omega$ such that $t = t_1.t_2$ we may say that:

- t_1 is a prefix of μ
- t_2 is a suffix of μ

Given a set $T \subseteq \mathbb{T}_\Omega$ of traces, we may denote by $\overline{T}$ the prefix-closure of T i.e. the set:

$$\overline{T} = \{t_1 \in \mathbb{T}_\Omega \mid \exists\ t_2 \in \mathbb{T}_\Omega,\ \text{s.t. } t_1.t_2 \in T\}$$

We can remark that for any trace t both ϵ and t are simultaneously prefixes and suffixes of t. Also, for any set T of traces, we may say that T is prefix-closed if $T = \overline{T}$

7.4.2 Prefixes and slices of multi-traces

In the same manner we can define prefixes in the case of multi-traces given that we have defined a concatenation law $\odot_C$ on multi-traces of $\mathbb{T}_{\Omega|C}$ defined up to a certain partition C of lifelines.

By analogy with Def.7.14, we define in Def.7.15 the notions of prefixes and slices of multi-traces.

Definition 7.15: Prefixes and slices of multi-traces

For any multi-traces μ, μ_1, μ_2 and μ_3 such that $\mu = \mu_1 \odot \mu_2 \odot \mu_3$ we may say that:

- μ_1 is a prefix of μ
- μ_2 is a slice of μ
- μ_3 is a suffix of μ

We can remark that, for any multi-trace μ:

- ϵ_C is simultaneously a prefix, a slice and a suffix of μ
- μ is simultaneously a prefix, a slice and a suffix of μ

We then define in Def.7.16 the notions of the prefix-closure and slice-closure of sets of multi-traces. For any set T of multi-traces, the prefix-closure of T, denoted by $\overline{T}$ is the minimal set of multi-traces that contains all the prefixes of all the multi-traces from T. Likewise, the slice-closure of T, denoted by $\underline{\overline{T}}$ is the minimal set of multi-traces that contains all the slices of all the multi-traces from T.

Definition 7.16: Prefix and slice closure

Given a set $T \subset \mathbb{T}_{\Omega|C}$ of multi-traces, we may denote by:

- $\overline{T}$ the prefix-closure of T i.e. the set

$$\overline{T} = \{\mu_1 \in \mathbb{T}_{\Omega|C} \mid \exists\ \mu_2 \in \mathbb{T}_{\Omega|C},\ \text{st}\ \mu_1 \odot \mu_2 \in T\}$$

- $\underline{T}$ the slice-closure of T i.e. the set

$$\underline{T} = \{\mu_2 \in \mathbb{T}_{\Omega|C} \mid \exists\ \mu_1, \mu_3 \in \mathbb{T}_{\Omega|C},\ \text{st}\ \mu_1 \odot \mu_2 \odot \mu_3 \in T\}$$

For any set T of multi-traces, we may then say that:

- T is prefix-closed if $T = \overline{T}$
- T is slice-closed if $T = \underline{T}$

Some interesting properties of the previously defined notions of prefix-closure and slice-closure are given in Lem.7.10. Those properties are the idempotency of both closures, the fact that prefix-closure, when applied on a slice-closed set, is the identity, and, the fact that for any set of trace T, T is included in its prefix-closure which is itself a subset of its slice-closure.

Lemma 7.10: Properties of prefix-closure and slice-closure

For any $T \subset \mathbb{T}_{\Omega|C}$:

$\overline{\overline{T}} = \overline{T}$	idempotency of prefix-closure
$\underline{\underline{T}} = \underline{T}$	idempotency of slice-closure
if $T' = \underline{T}$ then $\overline{T'} = T'$	slice-closed sets are prefix-closed
if $T' = \overline{T}$ then $\underline{T'} = \underline{T}$	slice-closure of prefix-closure is slice-closure
$T \subseteq \overline{T} \subseteq \underline{T}$	

Proof. Trivial. □

7.4.3 Semantics of prefixes and slices of accepted multi-traces

Now that we have defined prefixes and slices of multi-traces, we can expand the semantics of interaction terms to include prefixes and/or slices of accepted multi-traces. We will also define an intermediate semantics which is that of the projections of prefixes of accepted global traces.

Definition 7.17: Semantics of interaction with degraded observability

For any interaction $i \in \mathbb{I}_\Omega$:

- $\sigma^\dagger_{|C} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is s.t. $\sigma^\dagger_{|C}(i) = \{\mathbf{proj}_C(t) \mid t \in \overline{\sigma(i)}\}$
- $\overline{\sigma_{|C}} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is s.t. $\overline{\sigma_{|C}}(i) = \overline{\sigma_{|C}(i)}$
- $\overline{\circledcirc_C} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is s.t. $\overline{\circledcirc_C}(i) = \overline{\circledcirc_C(i)}$
- and $\underline{\overline{\sigma_{|C}}} : \mathbb{I}_\Omega \to \mathcal{P}(\mathbb{T}_{\Omega|C})$ is s.t. $\underline{\overline{\sigma_{|C}}}(i) = \underline{\overline{\sigma_{|C}(i)}}$

While $\sigma_{|C}$ is a semantics of fully observed multi-traces, $\sigma^\dagger_{|C}$, $\overline{\sigma_{|C}}$ and $\underline{\overline{\sigma_{|C}}}$ describe varying degrees of partial observations. With $\sigma^\dagger_{|C}$, events that are expected to occur, globally, at the end of accepted behaviors might be missing. With $\overline{\sigma_{|C}}$, missing events might be those at the end of some local behaviors even though they may not be at the end of the behavior globally. Finally, $\underline{\overline{\sigma_{|C}}}$ describe a more general case of partial observation in which events might be missing either at the beginning or the end of any local behavior.

We of course have that, for any interaction $i \in \mathbb{I}_\Omega$:

$$\sigma_{|C}(i) \subseteq \sigma^\dagger_{|C}(i) \subseteq \overline{\sigma_{|C}}(i) \subseteq \underline{\overline{\sigma_{|C}}}(i)$$

In the second part of the thesis, we will detail analysis methods that allow one to identify and discriminate between elements of those new semantics.

Conclusion

In this chapter we have defined the notion of multi-traces. A multi-trace can be described as a distributed observation of a global trace, in which events occurring on different localization are not ordered. Concretely, a multi-trace takes the form of a finite set of locally defined traces, each of which corresponding to a local behavior. As a result, a multi-trace describes a global behavior as a set of locally observed behaviors. The set of sets of multi-traces can be fitted with algebraic operators so as to constitute a $\mathcal{F}$-algebra similarly as the set of sets of traces was in Chap.4. Multi-traces can also be obtained by projecting global traces. The projection operator displays some interesting properties, which we have described and proved. It is notable that it preserves some algebraic structures but not all. Then, we described various notions of prefixes of multi-traces which in fact correspond to partial observations, as we will see in the following chapter.

In this chapter, we also extended the notion of what is the semantics of interactions by defining various semantics in terms of accepted multi-traces and various prefixes of accepted multi-traces.

With this chapter we end the first part of the thesis which consisted in the definition of the Interaction Language (IL), from syntax to semantics.

In the second part of the thesis we will be interested in defining various algorithms to solve the membership

problem i.e. determining whether or not a multi-trace belongs to a certain semantics. Because those algorithms consist in analyzing multi-traces against interaction models we may call them multi-trace analysis algorithms.

Part II

Multi-trace analysis

Chapter 8

On observing and analyzing executions of distributed systems

During the execution of a Distributed System (DS), information may enter or exit any given sub-system when it communicates either with some other sub-systems or with an external actor. We may assume that we can observe this exchange of information on the interfaces of sub-systems. We may also assume that this exchange of information, on each such interface, takes the form of a sequence of discrete communication events. Any sub-system then yields a sequence of such events that were successively observed during the execution of the system. As a result, each sub-system yields a locally defined sequence of events, and, after having collected those:

- if it is possible to reorder events globally, for instance, if a global clock is shared within the DS, then all the observed events can be recomposed as a global sequence in which events appear in the order in which they occurred globally during the execution.
- in the case where no two sub-system share a common clock, we cannot reorder events at all.
- a third possibility is that some (at least two) of the sub-systems of the DS share a common clock. In that case those sub-systems form a co-localization and we can partially reorder events.

Let us then assume that we may transpose those communications events into actions, as defined in Chap.4. This entails abstracting sub-systems into lifelines and abstracting exchanged information into messages. Then, in all three of the cases described above, we can synthesize the observation of the execution of the DS as a multi-trace that is defined up to a certain partition of lifelines. In the first case, it is the trivial partition, in the second, the discrete partition, and in the third, any other partition.

As we have seen in the first part of the thesis, interaction models can be used to specify the behaviors of DSs in terms of which scenarios of communications may be expressed during an execution. More precisely, any such individual behavior corresponds to a multi-trace in the projected multi-trace semantics of the interaction model.

As a result, we can relate multi-traces collected from observations of concrete executions of a DS to multi-traces specified by interaction models. In particular we might be interested in checking whether or not a multi-trace observed on a DS belongs to a certain semantics of an interaction which is intended to specify the behavior of that same DS.

We illustrate this process of "checking", which we may call multi-trace analysis on Fig.8.1. On the top right is represented a concrete implementation of a DS (here with three sub-systems, each represented by a lifeline). An execution of that DS yields (through observation, collection, and transposition) the multi-trace represented underneath it. We then analyze this multi-trace against a specification of the DS written in the form of an interaction model. The analysis may yield a certain verdict, which may not necessarily be a binary verdict.

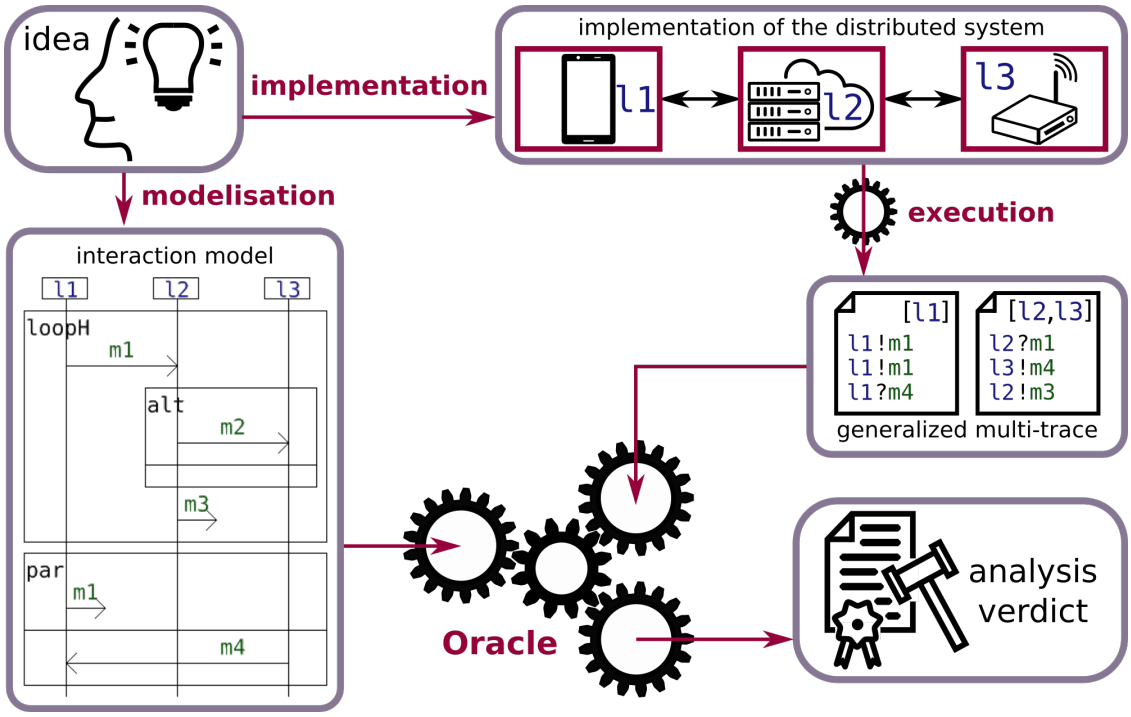


Figure 8.1: Analysing an observation of a DS execution against an interaction

We have seen that, in intention, behaviors specified by interaction models correspond to the projected multi-trace semantics $\sigma_{|C}$. Therefore, a first try at defining such an analysis process would be to check the membership of the multi-trace to the semantics $\sigma_{|C}$ of the specifying interaction model.

However, it might be so that the execution of the DS from which the multi-trace is collected is only partially observed. This results in having some missing events in the multi-trace. On each co-localization, the observation can either start too late or end too early or both. As a result, the multi-trace resulting from this partial observation of the execution can be a prefix or a slice of an accepted multi-trace.

If we assume full observation then we should check the inclusion into $\sigma_{|C}$. In cases where we may suppose a degraded observation then the execution might still be that of a specified behavior even though the multi-trace that was collected is not in the $\sigma_{|C}$ semantics. In that case we may rather consider the $\sigma^{\dagger}_{|C}$, $\overline{\sigma_{|C}}$ or $\underline{\sigma_{|C}}$ semantics depending on what it is that we want to check and what are our hypotheses concerning the

observation.

This problem of "belonging to a semantics" may be called the "membership problem". What we have described above is an application of that problem for a very particular case of testing the conformance of an implementation against a formal specification.

In this second part of the thesis we address the general problem of solving the membership problem for the various multi-trace semantics which we have defined in Chap.7. Let us remark that the membership problem for global traces is included in the latter given that we can equate global traces to multi-traces defined on a trivial partition of lifelines.

In this chapter we informally discuss how we can use the operational formulation of the trace semantics of interaction to recognize accepted multi-traces under various condition of full or partial observation. At the same time it may help in the understanding of partial observation and of the various notions of prefixes (and their distinctness) which we have defined.

The plan of this chapter is as follows:

- in Sec.8.1 we explain how the operational semantics defined in Chap.5 can be used to recognize accepted multi-traces,
- in Sec.8.2 we explain how we can characterize projections of prefixes of accepted global traces operationally,
- in Sec.8.3 we characterize prefixes in the sense of multi-traces and illustrate how they are not included in the previous notion of projections of prefixes,
- finally, in Sec.8.4 we provide an operational characterization of slices of multi-traces.

8.1 Recognizing accepted multi-traces

Let us consider the example from Fig.8.2. On the top left we have a globally defined trace ς which exactly corresponds to a given execution of a certain Distributed System (DS). With $\varsigma = l_1!m_1.l_2?m_1.l_2!m_2.l_1?m_2$, it describe the successive occurrence (globally ordered) of four events. On the top of Fig.8.2, we have an interaction model $i = seq(loop_H(\cdots), \cdots)$ which serves as a formal specification of that DS. Then, on the top right of Fig.8.2, we have a multi-trace μ which corresponds to a certain observation of the execution characterized by ς.

We then analyze this observed multi-trace μ w.r.t. the model i. This multi-trace is defined up to a partition $C = (\{l_1\}, \{l_2\})$ in this example (although it could be any partition). In the example on Fig.8.2 the multi-trace corresponds to a full observation of the behavior.

The key principle behind the analysis is to reconstruct a globally defined traces accepted by i that project onto μ by $\mathbf{proj}_C$. Constructing such a trace is made possible by taking elementary steps $(i, \mu) \rightsquigarrow (i', \mu')$ such that there exists an action a such that:

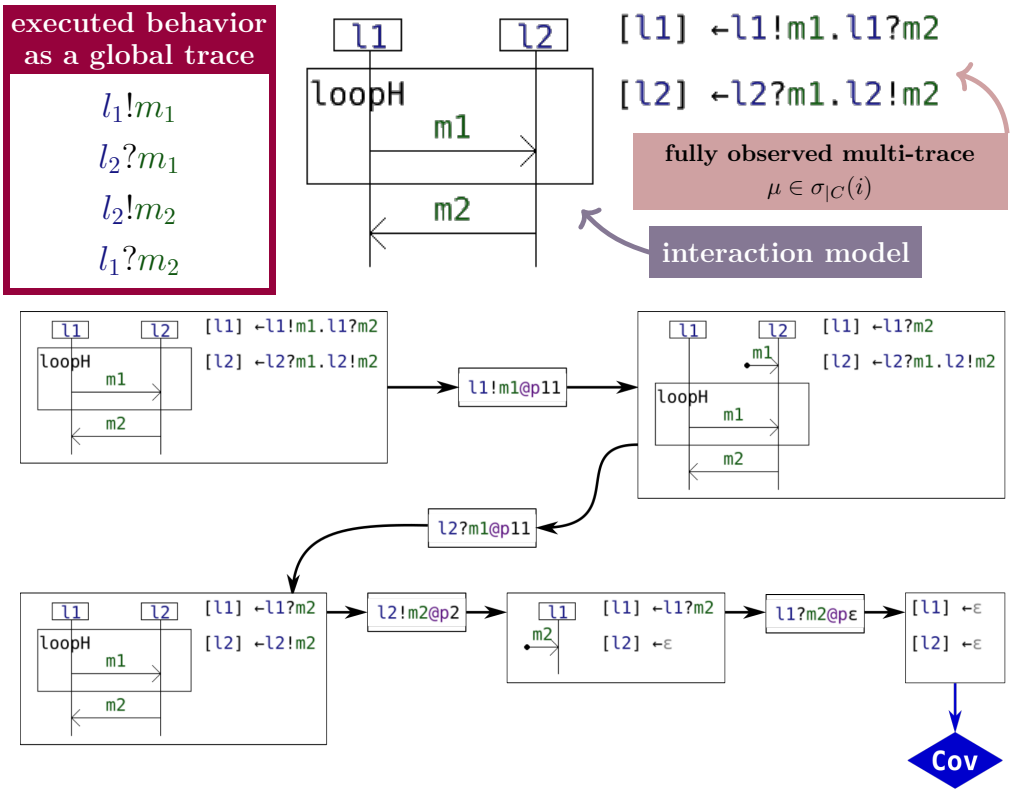


Figure 8.2: Analysing a multi-trace obtained from a full observation of an accepted behavior

- $i \xrightarrow{a} i'$ i.e. the execution of an action a allows one to transition from interaction i to interaction i'
- $\mu = a \vec{\odot}_C \mu'$ i.e. a is found at the head of a component $\theta_C(a)$ of μ and μ' is defined by removing this action from μ

This means that in any such step $(i, \mu) \rightsquigarrow (i', \mu')$ we simultaneously consume an action at the beginning of a component of the multi-trace μ and execute an instance of that action that is found in the interaction i.

As a result, the analysis may uncover a path in the execution tree of the interaction which allows the consumption of multi-trace μ. If, at the end of this path, we reach a (i_j, ϵ_C) in which the remaining interaction i_j can express the empty trace (which can be verified by the termination predicate from Chap.5), then this guarantees that $\mu \in \sigma_{|C}(i)$.

On the bottom part of Fig.8.2, we have represented such a path, which allows the consumption of the multi-trace μ. Here, the final pair that is reached contains the empty interaction which of course verifies that $\varnothing \downarrow$. Therefore we decorate this path with a local verdict *Cov* (in blue on Fig.8.2) which signifies that the multi-trace is an accepted multi-trace.

8.2 Recognizing projections of prefixes of accepted traces

Let us now consider the example analysis illustrated on Fig.8.3, which is a variation of the previous example. We observe here the same behavior of the same distributed system, which is also modelled by the same interaction term. However, in this example, the observation is degraded. We have indeed missed the

observation of the event which occurred last globally (as illustrated on the top left corner of Fig.8.3).

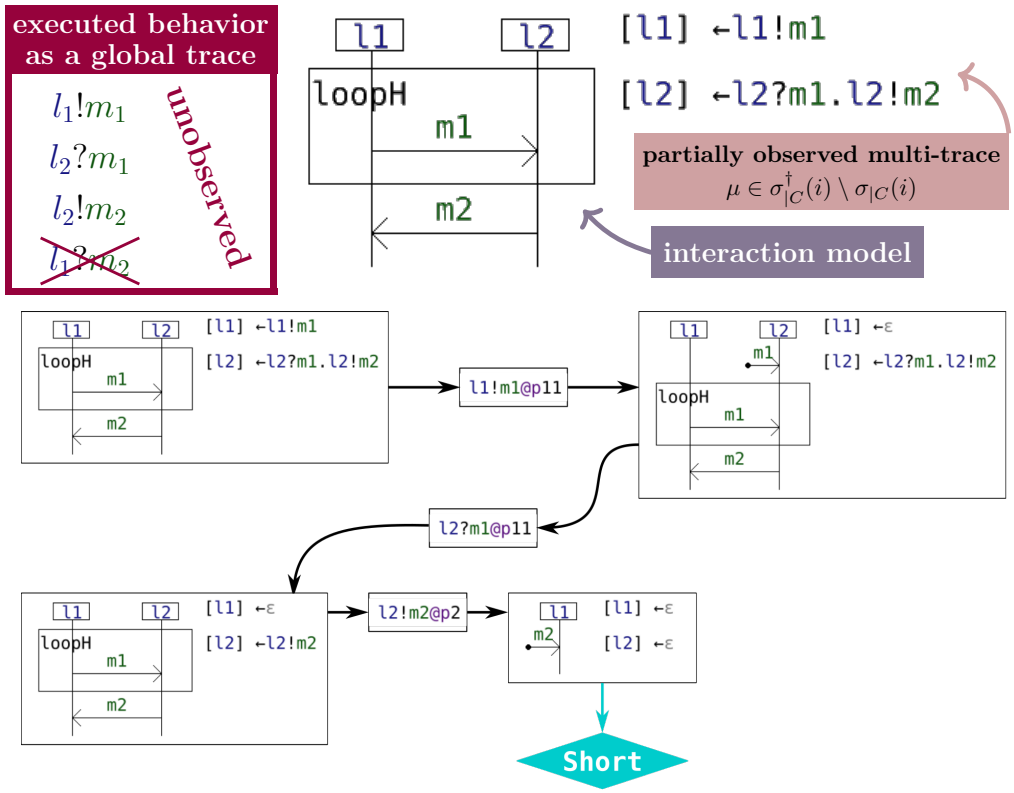


Figure 8.3: Analysis when the observation of an accepted behavior has ceased too early globally

As a result, the multi-trace that we analyze is partially observed. In this precise example, this multi-trace μ is in $\sigma^{\dagger}_{|C}(i) \setminus \sigma_{|C}(i)$ meaning that it is the projection (as a multi-trace) of a strict prefix of an accepted (global) trace but it is not itself an accepted multi-trace.

Elements of this set $\sigma^{\dagger}_{|C}(i) \setminus \sigma_{|C}(i)$ can be identified as illustrated on the bottom part of Fig.8.3. It suffices to find a path in the execution tree of the interaction which allows the consumption μ while reaching a (i_j, ϵ_C) in which the remaining interaction cannot express the empty trace i.e. that we have $i_j \not\downarrow$. On Fig.8.3 we decorated this path with a local verdict *Short* in cyan.

Finding such a path guarantees that $\mu \in \sigma^{\dagger}_{|C}(i)$ although it does not guarantees that $\mu \notin \sigma_{|C}(i)$ because there may be some other paths that consume μ while reaching an interaction that terminates (i.e. we may also be able to reach a *Cov* local verdict).

So as to guarantee that $\mu \in \sigma^{\dagger}_{|C}(i) \setminus \sigma_{|C}(i)$ one would have to ascertain that at least one path leads to *Short* and that no path leads to *Cov*.

8.3 Recognizing prefixes of accepted multi-traces

Let us now consider the example analysis illustrated on Fig.8.4, which, in the same manner, is also a variation of our example. Here, we also have a degraded observation but this degradation is somewhat different. Indeed, as illustrated on the top left corner of Fig.8.4 it is the $l_2!m_2$ event which observation has been missed. This event is not the last to have occurred globally but it is the last to have occurred on the

co-localization $\{l_2\}$. This may for instance be due to the local tester on co-localization $\{l_2\}$ having ceased its observation too early.

The multi-trace μ that we analyze is therefore partially observed but in this case, μ is in $\overline{\sigma_{|C}}(i) \setminus \sigma^{\dagger}_{|C}(i)$. Indeed, it is a prefix (in the sense of multi-traces) of an accepted multi-trace but it is not the projection of a prefix (in the sense of traces) of an accepted trace.

Elements of this set $\overline{\sigma_{|C}}(i) \setminus \sigma^{\dagger}_{|C}(i)$ can be identified as illustrated on the bottom part of Fig.8.4. This process of identification is similar to the ones we have described previously i.e. it relies on the simultaneous consumption of events from the multi-trace and their execution in the interaction model. However, once a component trace of the multi-trace has been entirely consumed, we allow the execution of some events on the corresponding co-localization so as to simulate the behavior of the unobserved lifelines. Those simulation steps can be used to fill-in the place left by events missing (because not observed) from the multi-trace during the exploration of the tree.

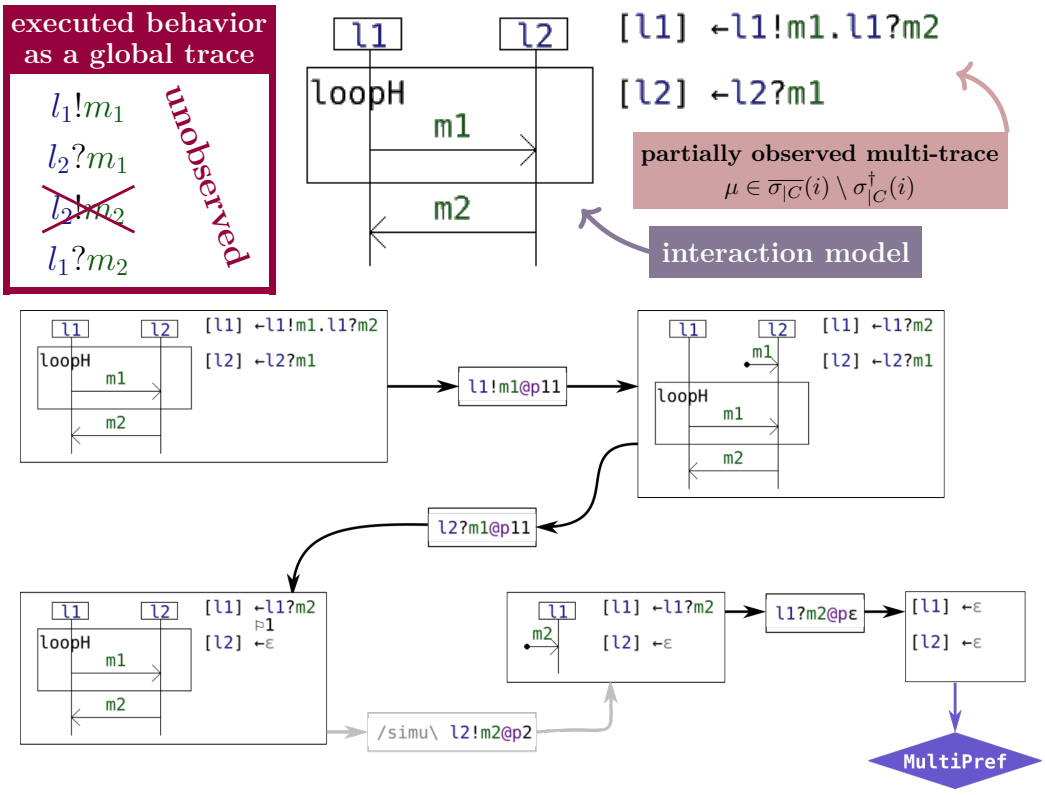


Figure 8.4: Analysis when the observation of an accepted behavior has ceased too early locally

Let us use the following notations to describe the example from Fig.8.4. We denote by (i_0, μ_0) the first vertex at the top left of the bottom part of Fig.8.4. We then have:

$$(i_0, \mu_0) \rightsquigarrow (i_1, \mu_1) \rightsquigarrow (i_2, \mu_2) \rightsquigarrow (i_3, \mu_3) \rightsquigarrow (\varnothing, \epsilon_C) \rightsquigarrow MultiPref$$

In this precise example, we can see that we manage to reach the (i_2, μ_2) vertex in which we have a multi-trace in which only $l_1?m_2$ remains. From this vertex we can see that it is not possible to consume the remaining event. However, the component trace on co-localization $\{l_2\}$ has been entirely emptied. This

means that observation have ceased on l_2, and, as a result, we may simulate its behavior. This allows us to execute action $l_2!m_2$ at position 2 which is the action that has not been observed (see top left corner of Fig.8.4).

Executing this action via simulation allows us to resume the consumption of the multi-trace. We then manage to reach the $(\varnothing, \epsilon_C)$ vertex in which the multi-trace is empty and the remaining interaction can express the empty trace. Given that we memorized that we needed some simulation steps to explore this path, we decorate this path not with a Cov local verdict, but with a dedicated $MultiPref$ local verdict.

Finding such a path guarantees that $\mu \in \overline{\sigma_{|C}}(i)$ although it does not guarantees that $\mu \notin \sigma^{\dagger}_{|C}(i)$ because there may be some other paths that may reach some other local verdicts (Cov or $Short$).

So as to guarantee that $\mu \in \overline{\sigma_{|C}}(i) \setminus \sigma^{\dagger}_{|C}(i)$ one would have to ascertain that at least one path leads to $MultiPref$ and that no path leads to either Cov or $Short$.

8.4 Recognizing slices of accepted multi-traces

Let us now consider our final example analysis illustrated on Fig.8.5. This time, events have been missed both at the end and at the beginning of some trace component of the multi-trace. Indeed, as illustrated on the top left corner of Fig.8.5 two events are missing:

- $l_1!m_1$ at the beginning of component $\mu_{|\{l_1\}}$
- $l_2!m_2$ at the end of component $\mu_{|\{l_2\}}$

In that case, we have both a local tester which ceased its observation too early (the one on co-localization $\{l_2\}$) and another local tester which started its observation too late (the one on co-localization $\{l_1\}$).

The multi-trace μ that we analyze is therefore partially observed with $\mu \in \underline{\overline{\sigma_{|C}}}(i) \setminus \overline{\sigma_{|C}}(i)$. Indeed, it is a slice (in the sense of multi-traces) of an accepted multi-trace but it is not a prefix (in the sense of multi-traces) of an accepted multi-trace.

Elements of this set $\underline{\overline{\sigma_{|C}}}(i) \setminus \overline{\sigma_{|C}}(i)$ can be identified as illustrated on the bottom part of Fig.8.5. This process of identification allows simulation steps both:

- after the entire consumption of any given trace component (as in the previous case)
- and before one has started consuming any event from a certain trace component

As in the previous case, those simulation steps can be used to fill-in the place left by events missing (unobserved) from the multi-trace during the exploration of the tree.

Let us use the same notations as before for describing the path described in Fig.8.5 i.e. we denote by (i_0, μ_0) the first vertex at the top left and we have $(i_0, \mu_0) \rightsquigarrow (i_1, \mu_1) \rightsquigarrow (i_2, \mu_2) \rightsquigarrow (i_3, \mu_3) \rightsquigarrow (\varnothing, \epsilon_C) \rightsquigarrow Slice$

At the start of the analysis, from the (i_0, μ_0) vertex, we can see that we cannot consume any event. Indeed frontier actions consists in $l_1!m_1$ and $l_2!m_2$ which are neither of the head actions of the multi-trace.

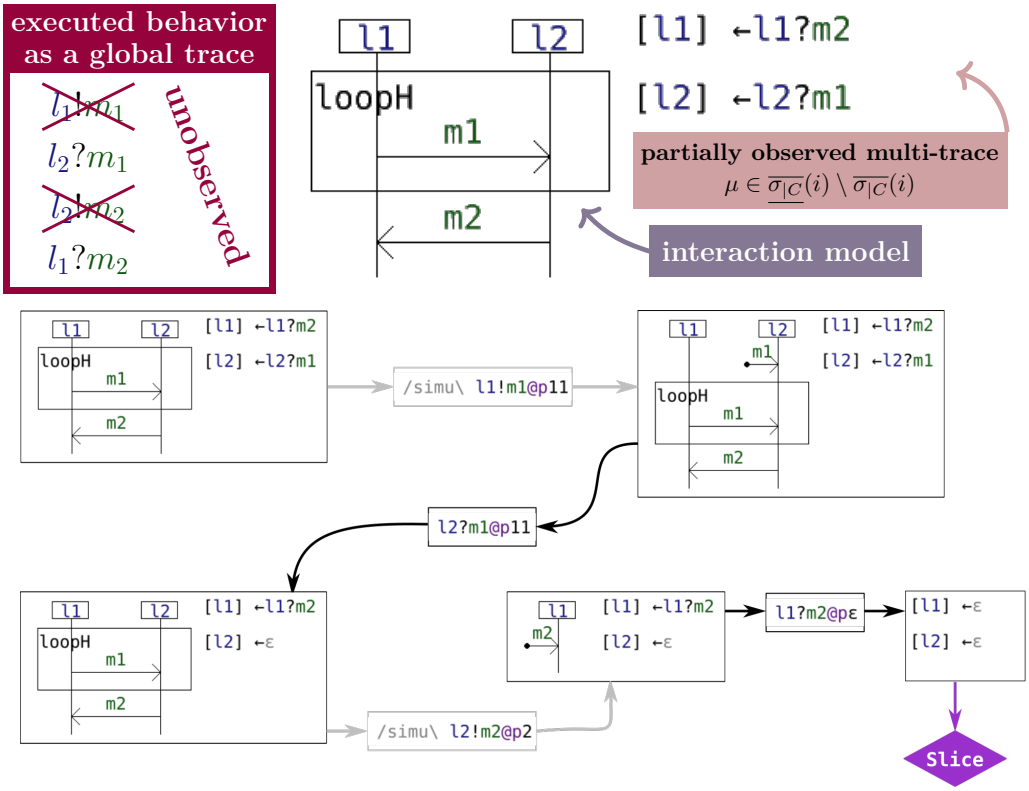


Figure 8.5: Analysis when observation has both started too late and ceased too early

However, given that we have not started the consumption on any of the two components, we allow simulation steps to be taken. Those simulation steps will act as a preliminary for the consumption of the multi-trace.

In the path described on Fig.8.5 we have chosen to simulate the execution of action $l_1!m_1$ at position 11. We can then see that this unlocked the further consumption of the multi-trace by allowing $l_2?m_1$ to be consumed.

Then, another simulation step is taken, this time as a postamble to the consumption of the component on $\{l_2\}$. This use of simulation step is the same as in the previous example.

Finally, we then manage to reach the $(\varnothing, \epsilon_C)$ vertex in which the multi-trace is empty and the remaining interaction can express the empty trace. Given that we memorized that we needed some simulation steps both before and after the consumption of some trace components, we decorate this path with a dedicated *Slice* local verdict.

Finding such a path guarantees that $\mu \in \overline{\sigma_{|C}}(i)$ although it does not guarantees that $\mu \notin \overline{\sigma_{|C}}(i)$ because there may be some other paths that may reach some other local verdicts (Cov or $Short$ or $MultiPref$).

So as to guarantee that $\mu \in \overline{\sigma_{|C}}(i) \setminus \overline{\sigma_{|C}}(i)$ one would have to ascertain that at least one path leads to *Slice* and that no path leads to either Cov or $Short$ or $MultiPref$.

Conclusion

In this chapter we have introduced the problematic of analyzing multi-traces against interaction models and proposed a solution in the form of analysis algorithms which key principle is to successively consume actions

from the multi-trace while executing them on the interaction model. We have also indirectly characterized the various notions of prefixes which we defined in Chap.7. While in Chap.7 we have define those notions of prefixes from an algebraic perspective i.e. as closures of sets, we provided in this chapter, an operational characterization of their nature, with some events (corresponding to atomic small-steps) not being observed during the execution. We have also provided a first sketch of a solution to recognize multi-traces that were observed under conditions of degraded observability. This sketch of solution relies on replacing missing events by simulation steps in which we explore what could have been expressed by the sub-systems that were not observed.

In this second part of the thesis we will define various algorithms so as to solve the membership problem for the various semantics which we have defined in Chap.7.

Chapter 9

Algorithm for recognizing accepted multi-traces

Contents

In this chapter we are interested in solving the membership problem for the $\sigma_{|C}$ semantics of accepted multi-traces. We define an algorithm which includes and extends those that we have proposed in our published papers [93, 94].

In [93] and [94], we respectively addressed global trace analysis and multi-trace analysis on the discrete partition.

In this chapter, we generalize the solutions from [93, 94] by providing an algorithm for recognizing accepted multi-traces defined up to any partition $C \in \text{Part}(L)$. For any multi-trace $\mu \in \mathbb{T}_{\Omega|C}$ and for any interaction $i \in \mathbb{I}_\Omega$, this algorithm is able to determine (in NP time at worst) whether or not $\mu \in \sigma_{|C}(i)$.

The plan of this chapter is as follows:

- in Sec.9.1 we define the algorithm,
- in Sec.9.2 we prove the correctness of the algorithm i.e. that it solves the membership problem for the $\sigma_{|C}$ semantics,
- in Sec.9.3 we discuss the complexity class of the underlying problem.

9.1 Definition of the algorithm

We define a process able to decide whether or not a multi-trace μ (defined up to a partition C) is accepted by an interaction i. We have seen in Chap.8 that such a process can be defined through the use of transitions $(i, \mu) \rightsquigarrow (i', \mu')$ which correspond to consuming an action at the head of a component of the multi-trace while executing it in the interaction model.

Starting from an initial vertex (i_0, μ_0), this allows us to navigate in a graph that is quite similar to the execution tree of i_0 (as seen in Chap.6) but, instead of having nodes that host interaction terms, the vertices of this graph host tuples that are composed of both an interaction term and a multi-trace i.e. objects of the form (i, μ). This graph is called the analysis graph. We propose an abstract illustration of such a graph Fig.9.1. An analysis graph that contains a vertex (i_0, μ_0) then contains a refinement of the execution tree of i_0. Indeed, we only allow a subset of the transitions $i \xrightarrow{a} i'$ enabled by the execution relation $\rightarrow$ which are those in which the executed action a matches a certain head of a certain component of the multi-trace μ.

Via the exploration of this analysis graph, we will uncover paths that lead towards one of two local verdicts which are Cov (for "covered") and $UnCov$ (for "uncovered"), as illustrated on Fig.9.1. The existence of a path towards Cov means that the multi-trace to analyze is accepted and as a result we may return a $Pass$ global verdict. The absence of any path towards Cov means that the multi-trace to analyze is not accepted and as a result we may return a $Fail$ global verdict.

Local verdicts are eventually reached by repeating steps of the form $(i, \mu) \rightsquigarrow (i', \mu')$ which we have described. Indeed, by doing so, we will eventually:

- either empty-out the multi-trace μ, by having removed all of its events one by one. In this case we will have identified a path of execution within the execution tree of the original interaction which allows

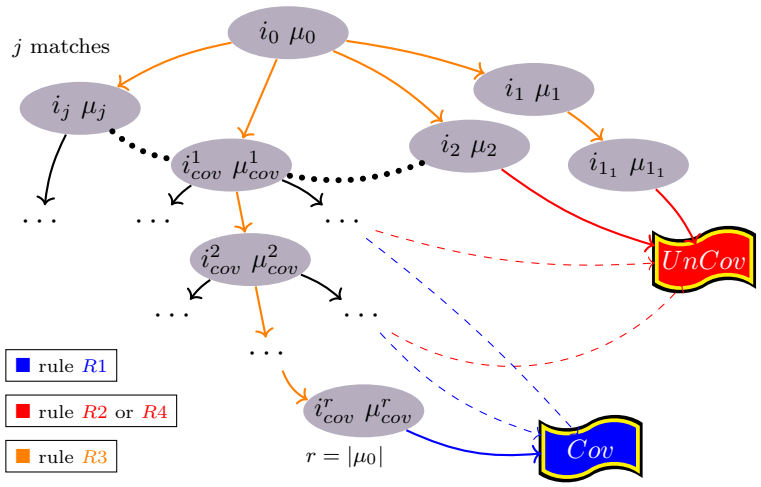


Figure 9.1: Principle of multi-trace analysis

the consumption of the multi-trace. Then:

- if the remaining interaction can express the empty trace, we can conclude that the multi-trace is accepted via the formulation of a Cov local verdict, which decorates the path leading to that vertex (as illustrated in the bottom of Fig.9.1)
- if the remaining interaction cannot express the empty trace, then it means that the multi-trace cannot be consumed in the manner described by the path leading to that vertex. From the point of view of that last vertex, the empty behavior cannot be expressed so the empty multi-trace is not an accepted multi-trace. As a result, we decorate this path with an $UnCov$ local verdict (as illustrated on the right of Fig.9.1)

- or be in a situation in which we cannot match any executable action in i with any head action from components of μ. In that case, we assign a local verdict $UnCov$ to the path leading to the vertex in which no matches have been found. As for the previous case, this $UnCov$ local verdict signifies that the multi-trace cannot be consumed in this manner. From the point of view of the last vertex in the path, the remaining multi-trace does not characterize any behavior that is accepted by the remaining interaction.

Given that a multi-trace has a finite number of actions, and that there may be only a finite number of matches between frontier actions of an interaction and a head action of a multi-trace, an analysis graph constructed in this manner is a finite graph.

As a result of the two last points, for any vertex (i, μ) there exists a finite number of outgoing paths, all of them are finite, and all such paths can be prolonged so as to eventually reach a local verdict Cov or $UnCov$. Hence, there exists at least one path between any (i, μ) and either Cov or $UnCov$, and, reciprocally, either Cov or $UnCov$ are reachable from any vertex (i, μ).

As explained previously, this allows us to define a global verdict for the analysis of the initial multi-trace μ_0 w.r.t. the initial interaction i_0. If there exists a path leading to Cov, the global verdict is $Pass$ given that

there exists a manner to reorder all the events from multi-trace μ_0 into a global trace $t_0 \in \sigma(i_0)$ accepted by i_0 and therefore $\mu_0 = \mathbf{proj}_C(t_0) \in \sigma_{|C}(i_0)$. If no such path exists then the global verdict is $Fail$.

Detailed description of Fig.9.1

Let us consider the illustration on Fig.9.1. Starting from vertex (i_0, μ_0), a number of paths can be explored. From (i_0, μ_0), there exists j outgoing transitions to other vertices, j being the number of matches between immediately executable actions (frontier actions) of i_0 and actions at the heads of components from μ. Exploration steps (which are not represented but implicitly designated by $\cdots$ in Fig.9.1) are then repeated for every one of those children. Ultimately, every path that is thus created leads back to one of the two local verdicts $UnCov$ or Cov (given the decreasing size of the multi-trace).

Paths starting from (i_0, μ_0) may have different lengths and different outcomes. This is explained by the fact that the graph explores how some executions of i_0 might (or might not) cover the behavior expressed by the multi-trace μ_0. It may be so that there exists several executions of i_0 that match μ_0. At the same time there might exists some that do not, and the fact that they do not match μ_0 can be made clear after an arbitrary number (bounded by the length of μ_0) of small-steps.

With regard to Cov, the fact that several paths might lead to it may be explained by the fact that several global traces can be projected into the same multi-trace (as explained in Chap.7). Therefore, when trying to reorder μ_0 into a global trace that satisfies i_0, we can find several of those.

In the example illustrated in Fig.9.1, there exists (at least one) such path $(i_0, \mu_0) \rightsquigarrow (i^1_{cov}, \mu^1_{cov}) \rightsquigarrow \cdots \rightsquigarrow (i^r_{cov}, \mu^r_{cov})$ that leads to Cov. Given that obtaining Cov requires to empty the initial multi-trace μ_0 we have that:

- the final remaining multi-trace is the empty multi-trace i.e. $\mu^r_{cov} = \epsilon_C$
- the length of the path (within the analysis graph) r is equal to that of the multi-trace $|\mu_0|$ (in number of actions)

The existence of this path then implies that the global verdict $Pass$ will be returned.

Let us also remark that all paths leading to Cov are of the same length (that of the initial multi-trace: $|\mu_0|$). However paths leading to $UnCov$ are of any length smaller or equal to $|\mu_0|$.

The process which allows the exploration of the analysis graph and the discovery of paths leading to local verdicts can be described by a set of four rules denoted by $R1$ through $R4$. On Fig.9.1, we have illustrated the application of those rules via differently colored arrows between the vertices of the graph (see the legend on the bottom-left corner). In the following section we define those four rules and detail their applications on some examples.

9.1.1 Presentation of the rules for exploring the analysis graph

In the previous section we have briefly explained the principle behind the multi-trace analysis algorithm. We will now detail each rule on which it is based.

Rule *R3* for the consumption of events from the multi-trace

Let us start by describing rule *R3* on an example. The application of the rule on that example is illustrated on Fig.9.2. For this example, we consider a signature $\Omega = (L, M)$ with $L = \{l_1, l_2, l_3\}$, $M = \{m_1, m_2, m_3, m_4\}$ and the discrete partition $\check{L} = (\{l\})_{l \in L}$.

We start the analysis on a vertex (the left-most vertex) that contains both an initial interaction $i_0 = seq(loop_H(\cdots), par(\cdots))$ and an initial multi-trace μ_0 such that $\mu_{|\{l_1\}} = l_1!m_1$, $\mu_{|\{l_2\}} = l_2?m_1.l_2!m_2$ and $\mu_{|\{l_3\}} = l_3?m_2$. The analysis is then that of μ_0 (the multi-trace to analyze) with regards to i_0 (the interaction which serves as a specification).

The first step in the analysis is to determine the frontier of the interaction i_0. In that case, we have three frontier actions:

- two instances of $l_1!m_1$ at positions 111 and 21
- and one instance of $l_3!m_4$ at position 221

Then, for each frontier action, we look for matching actions on the heads of the components of the multi-trace. Here, both instances of $l_1!m_1$ match the first action of component $\mu_{|\{l_1\}}$ of the multi-trace. However, $l_3!m_4$ has no match in the multi-trace.

Applying rule *R3* then consists in consuming in the multi-trace an event that is at the head of a component while executing an instance of that action in the interaction. Here we have two matches and hence we have two possible applications of rule *R3*, which opens-up two branches in the analysis graph. In the newly created vertices, we have a follow-up interaction resulting from the execution of a specific action and a multi-trace in which this action has been removed.

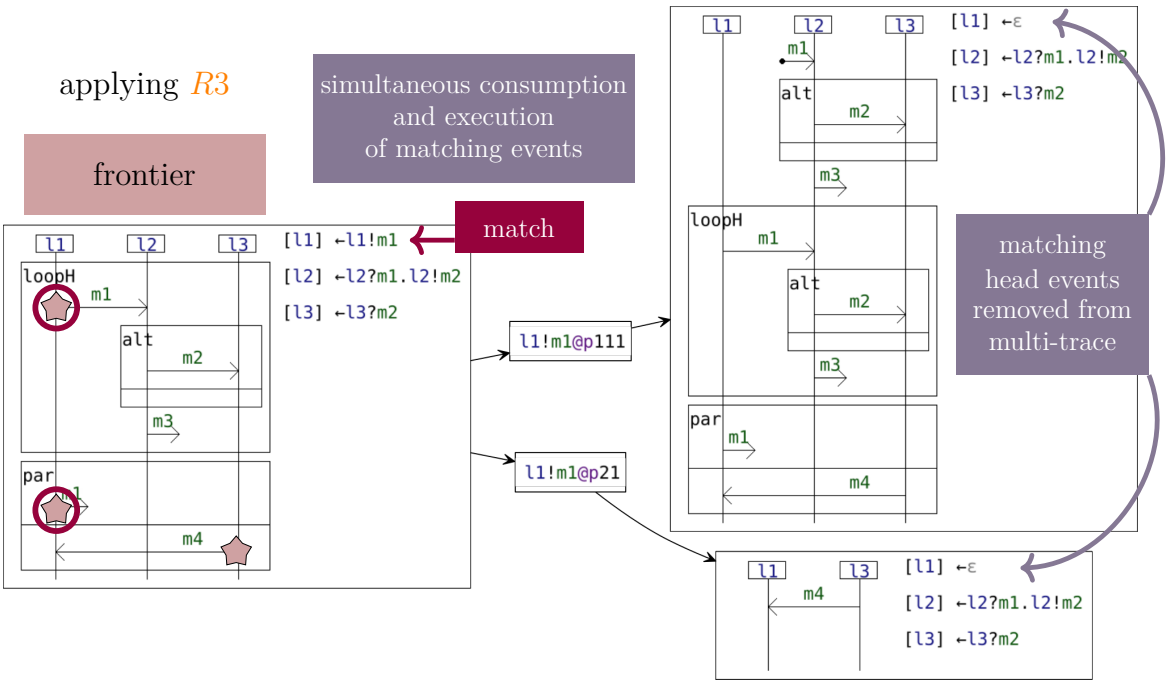


Figure 9.2: Example of application for rule *R3*

Rule $R3$ can be formalized as follows; for any interaction i and any multi-trace μ over a partition C of L:

$$(R3)\ \frac{i \qquad a \overrightarrow{\odot}_C \mu}{i' \qquad \mu}\ i \xrightarrow{a} i'$$

Every application of rule $R3$ reduces the size of the multi-trace by 1. As a result, repeating the application of rule $R3$ can eventually yield to either emptying the multi-trace or being in the impossibility to apply $R3$ anymore.

- What happens in the first case is then specified by either rule $R1$ or rule $R2$ that we will introduce in the following.
- The second case corresponds to the case in which the multi-trace μ is not empty but we cannot find any matches between frontier actions and the heads of trace components of μ. What happens then is dictated by rule $R4$.

Rule $R4$ when it is impossible to consume any event

As explained before, rule $R4$ specifies what happens during the analysis when the multi-trace is not empty and it is not possible to consume any event from it. Let us continue on the previous example by selecting one of the explored branches. We have represented this branch of the analysis graph on Fig.9.3. As the reader may have noticed, a previous application of rule $R3$ yielded a vertex (i, μ) in which we have the interaction $i = strict(l_3!m_4, l_1?m_4)$ and a multi-trace μ defined over the discrete partition of $L = \{l_1, l_2, l_3\}$.

The frontier of i contains a single instance of $l_3!m_4$ at position 1. However, on the component $\mu_{|\{l_3\}}$ of the multi-trace we have $\mu_{|\{l_3\}} = l_3?m_2$. As a result, in the remaining multi-trace to analyse there is no match for this action.

Given that there is no match at all, we can apply rule $R4$, which allows the transition $(i, \mu) \rightsquigarrow UnCov$, where $UnCov$ is a local verdict (signifying that the multi-trace cannot be consumed in this manner). From the point of view of the (i, μ) vertex, the remaining multi-trace does not characterize any behavior that is accepted by the remaining interaction.

Rule $R4$ can be formalized as follows; for any interaction i and any multi-trace μ over a partition C of L:

$$(R4)\ \frac{i \qquad \mu}{UnCov}\ \begin{cases} (\mu \neq \epsilon_C) \\ \wedge (\nexists\ i \xrightarrow{a} i' \text{ s.t. } \mu = a \overrightarrow{\odot}_C \mu') \end{cases}$$

Rules $R1$ & $R2$ when the multi-trace has been emptied

Finally, the two last rules, rules $R1$ and $R2$ dictate what happens once the multi-trace has been emptied. Let us consider the example on Fig.9.4 in which a previous application of rule $R3$ has yielded two new vertices in which the multi-trace has been emptied.

In this example, we have $L = \{l_1, l_2\}$ and C is the discrete partition of L. As can be seen on Fig.9.4, in both vertices on the right, the multi-trace, defined over two components $\{l_1\}$ and $\{l_2\}$, is empty.

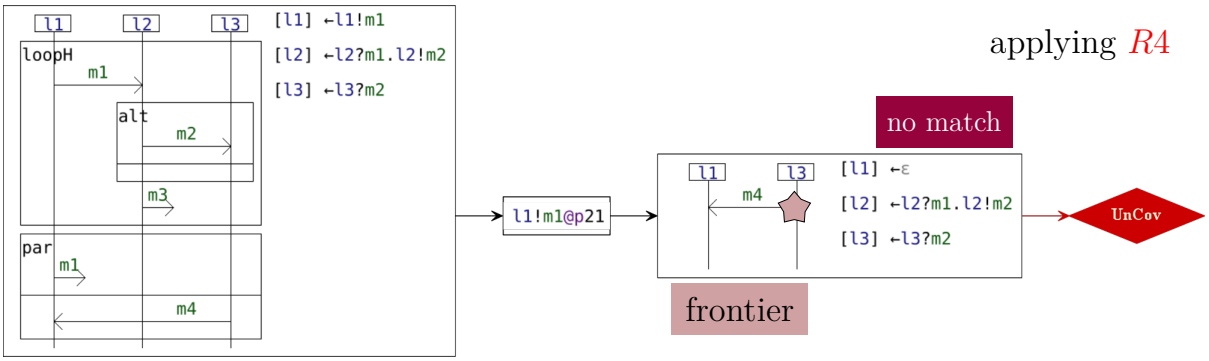


Figure 9.3: Example of application for rule $R4$

In the top-right vertex, the remaining interaction i can express the empty behavior. This can be statically inferred from its term structure using the termination predicate $i \downarrow$ from Chap.5. Here we have a loop which can be repeated zero times, and an alternative in which one branch is the empty interaction. Given that this interaction can express the empty trace, we can apply rule $R1$, which links local verdict Cov to this path (signifying that the multi-trace characterizes an accepted behavior).

In the bottom-right vertex, the remaining interaction i cannot express the empty behavior i.e. we have $i \not\downarrow$. As a result, the empty multi-trace is not accepted at this point. The application of rule $R2$ links the $UnCov$ local verdict to the path. The $UnCov$ verdict here has the same meaning as in the case of the application of rule $R4$. It signifies that, at this point, the remaining multi-trace is not accepted by the remaining interaction.

Rules $R1$ and $R2$ can be formalized as follows; for any interaction i and any partition C of L:

$$(R1)\ \frac{i \quad \epsilon_C}{Cov}\ i\downarrow \qquad\qquad (R2)\ \frac{i \quad \epsilon_C}{UnCov}\ i\not\downarrow$$

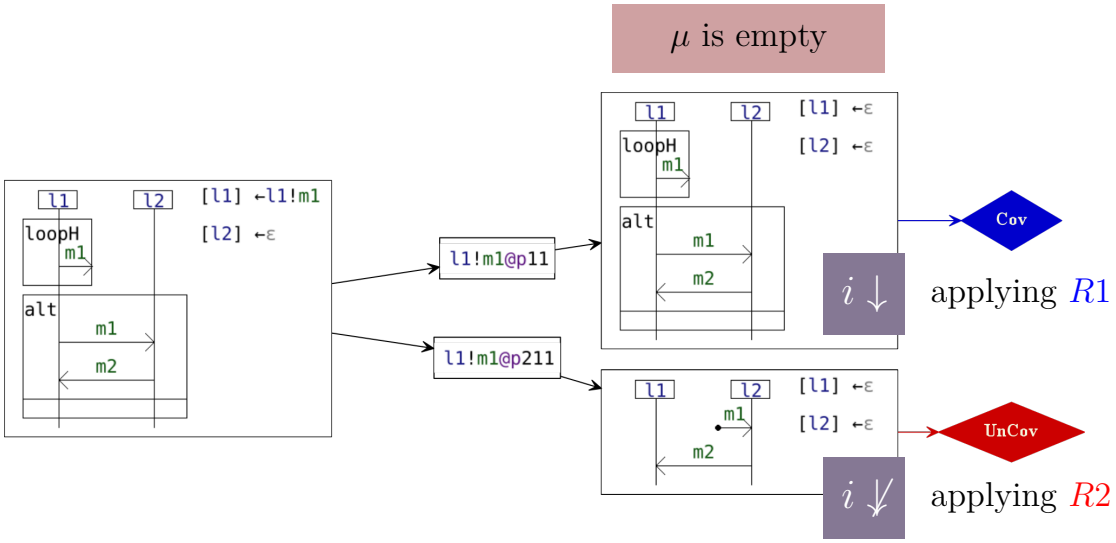


Figure 9.4: Example of application for rules $R1$ and $R2$

9.1.2 Formal Definition of the analysis process

As we have seen, the analysis of multi-trace analysis relies on four rules, denoted by $R1$, $R2$, $R3$ and $R4$. Those rules define a directed graph $\mathbb{G}$ in which vertices are either a tuple $(i, \mu) \in \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C}$ or a local verdict $v \in \{Cov, UnCov\}$. We note $\mathbb{V} = \{Cov, UnCov\} \cup (\mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C})$ the set of all such vertices. For x in $\{1, 2, 3, 4\}$, the rule $(Rx)\frac{v}{v'}$ *cond*, with $v \in \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C}$ and $v' \in \mathbb{V}$ specifies edges of the form $v \rightsquigarrow v'$ of that graph, provided that v satisfies condition *cond*.

The formal definition of the graph $\mathbb{G}$ and how to construct it is given on Def.9.1.

Definition 9.1: Rules of Multi-Trace Analysis

The analysis relation $\rightsquigarrow \subseteq \mathbb{V} \times \mathbb{V}$ is defined as:

$$(R1)\ \frac{i \qquad \epsilon_C}{Cov}\ i\downarrow \qquad\qquad (R2)\ \frac{i \qquad \epsilon_C}{UnCov}\ i\not\downarrow$$

$$(R3)\ \frac{i \qquad a \overrightarrow{\odot}_C \mu}{i' \qquad \mu}\ i \xrightarrow{a} i' \qquad\qquad (R4)\ \frac{i \qquad \mu}{UnCov} \begin{cases} (\mu \neq \epsilon_C) \\ \wedge (\nexists\ i \xrightarrow{a} i' \text{ s.t. } \mu = a \overrightarrow{\odot}_C \mu') \end{cases}$$

We can remark that vertices of the form (i, μ) are not sinks. Indeed:

- if μ is the empty multi-trace ϵ_C, given that we must either have $i\downarrow$ or $i\not\downarrow$, either $R1$ or $R2$ applies and therefore there exists an outgoing edge from any (i, ϵ_C)
- if $\mu \neq \epsilon_C$, one can either have or not have matches between frontier actions and multi-trace component heads. Hence, an outgoing edge exists according to either rule $R3$ or rule $R4$.

As a result, local verdicts $\{Cov, UnCov\}$ are the only two sinks of graph $\mathbb{G}$.

Rules $R1$, $R2$ and $R4$ specify edges from vertices of the form (i, μ) to local verdicts.

The rule $R3$ specifies edges $(i, \mu) \rightsquigarrow (i', \mu')$ such that there exists an action a that is both:

- immediately executable in i i.e. such that there exists a follow-up interaction i' s.t. $i \xrightarrow{a} i'$
- at the head of a component of μ i.e. such that the component on the co-localization $\theta_C(a)$ starts by a (meaning we have $\mu_{|\theta_C(a)} = a.t$), which also translates into having μ of the form $a \overrightarrow{\odot}_C \mu'$

Then, the vertex that is reached after applying $R3$ is (i', μ') as specified above i.e. such that i' is a follow-up in $i \xrightarrow{a} i'$ and μ' is such that $a \overrightarrow{\odot}_C \mu'$.

For applying $R3$, we must consider an action that is immediately executable in i. We have seen in Chap.6 that there is a finite number of such actions and that we can compute them as a "frontier of execution" $\texttt{frt}(i)$. As a result, for a vertex (i, μ), there are at most $|\texttt{frt}(i)|$ possible applications of the rule $R3$ with $|\texttt{frt}(i)|$ bounded by the number of occurrences of actions in i.

Let us then consider $|\mu|$ the number of actions occurring in a multi-trace μ, i.e. the sum of lengths of its component traces. Let us extend this notation to vertices, that is, $|(i, \mu)|$ defined as $|\mu|$, and $|Cov|$ and

$|UnCov|$ defined as -1. For any edge $v \rightsquigarrow v'$ of $\mathbb{G}$, we have $|v'| < |v|$ with $|v'| \geq -1$. Consequently, the successive application of the rules strictly decrements the size of vertices and from any vertex (i, μ), any maximal outgoing path is finite, and terminates in a local verdict in $\{Cov, UnCov\}$ (since (i, μ) are not sinks of $\mathbb{G}$). Thus, $\mathbb{G}$ is an acyclic graph. With the notation $v \overset{*}{\rightsquigarrow} v'$ to indicate that there is a path from v to v' in $\mathbb{G}$, we define multi-trace analysis in Def.9.2.

Definition 9.2: Multi-Trace Analysis

For any signature $\Omega = (L, M)$ and any partition $C \in \text{Part}(L)$ of lifelines, we define $\omega_C : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{Pass, Fail\}$ such that for any $i \in \mathbb{I}_\Omega$ and $\mu \in \mathbb{T}_{\Omega|C}$ we have:

- $\omega_C(i, \mu) = Pass$ iff there exists a path $(i, \mu) \overset{*}{\rightsquigarrow} Cov$
- $\omega_C(i, \mu) = Fail$ otherwise;
  i.e. for all path $(i, \mu) \overset{*}{\rightsquigarrow} v$ with $v \in \{Cov, UnCov\}$, then $v = UnCov$

The function ω_C is well-defined. Indeed, we established that any maximal path from a vertex (i_0, μ_0) has a maximum length of $|\mu| + 1$ and end on a local verdict (Cov or $UnCov$). Besides, each intermediate vertex (i, μ) between (i_0, μ_0) and a local verdict has a number of children bounded by the number of actions of i. Therefore, the number of vertices reachable from (i_0, μ_0) is finite

Application on an example

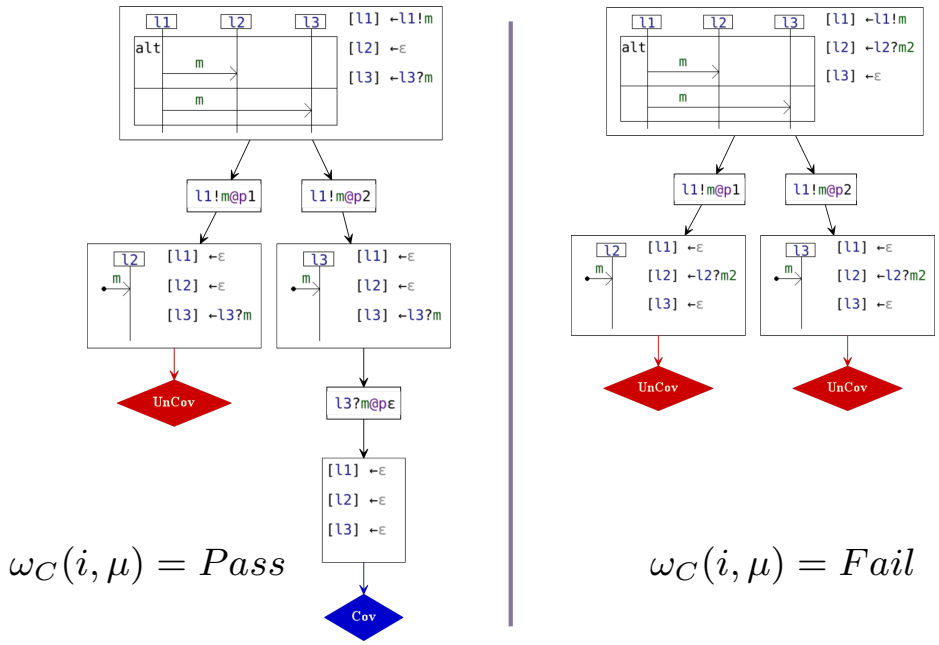


Figure 9.5: Multi-trace analysis on two examples respectively yielding a global $Pass$ and a global $Fail$

On Fig.9.5 are illustrated two examples of the analysis of a multi-trace. In both examples, we have a signature $\Omega = (L, M)$ with M that includes a certain message m and $L = \{l_1, l_2, l_3\}$ and we consider

the discrete partition C of L for the definition of multi-traces (i.e. they have three components on respectively $\{l_1\}$, $\{l_2\}$ and $\{l_3\}$. Also, in both examples, the multi-trace is analysed against the interaction $i = alt(strict(l_1!m, l_2?m), strict(l_1!m, l_3?m))$ i.e. an alternative between the passing of message m either between lifelines l_1 and l_2 or between l_1 and l_3. Then:

- in the first example, on the left of Fig.9.5, the multi-trace μ to analyse is such that $\mu_{|\{l_1\}} = l_1!m$, $\mu_{|\{l_2\}} = \epsilon$ and $\mu_{|\{l_1\}} = l_3?m$. As a result, we can intuitively guess that it is an accepted multi-trace which corresponds to the case in which the message m is passed from l_1 to l_3. The analysis graph starting from (i, μ) is drawn on the left part of Fig.9.5. As the reader can remark one of the paths leads to $UnCov$ and the other to Cov. As a result, the analysis yields the $Pass$ global verdict.

- in the second example, on the right of Fig.9.5, the multi-trace μ to analyse is such that $\mu_{|\{l_1\}} = l_1!m$, $\mu_{|\{l_2\}} = l_2?m_2$ and $\mu_{|\{l_1\}} = \epsilon$. As the reader may have noticed, in the multi-trace, l_1 emits m but it is an unexpected message m_2 that is received on l_2. The analysis graph starting from (i, μ) is drawn on the right part of Fig.9.5. In this case, all the paths starting from (i, μ) lead to the $UnCov$ local verdict. As a result, the analysis yields the $Fail$ global verdict.

Now that we have formally defined our process to analyze multi-trace, we will, in the next section, prove its correctness with regards to the semantics of interactions.

9.2 Proof of correctness of the multi-trace analysis algorithm

Let us consider a given signature $\Omega = (L, M)$ and a given partition $C \in \text{Part}(L)$ of lifelines.

In the following, we prove that the function ω_C in charge of analysing multi-traces defined over C w.r.t. interactions exactly captures the semantics of interactions. More precisely, we will prove that for any (i, μ) in $\mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C}$ we have that:

- $\omega_C(i, \mu) = Pass$ iff $\mu \in \sigma_{|C}(i)$

- and by extension, $\omega_C(i, \mu) = Fail$ iff $\mu \notin \sigma_{|C}(i)$

Let us remark that this proof has been machine-checked using the Coq proof assistant in [87] in the case of the discrete partition as in [94].

Given that $\sigma_{|C}(i)$ is the set of multi-traces obtained by projecting (with $\mathbf{proj}_C$) accepted global traces from $\sigma(i)$, it then suffices to prove that for any trace $\varsigma \in \mathbb{T}_\Omega$ we have $\omega_C(i, \mathbf{proj}_C(\varsigma)) = Pass$ iff $\varsigma \in \sigma(i)$.

This can be done in two steps. The first step is given by Th.9.1 and states that whenever a global trace ς is accepted by an interaction i then this means that its projection as a multi-trace must yield $Pass$ when analysed w.r.t. i.

Theorem 9.1: Accepted implies *Pass*

For any interaction $i \in \mathbb{I}_\Omega$ and any global trace $\varsigma \in \mathbb{T}_\Omega$:

$$\Big(\varsigma \in \sigma(i) \Big) \Rightarrow \Big(\omega_C(i, \mathbf{proj}_C(\varsigma)) = Pass \Big)$$

Proof. Let us reason by induction on (the size of) the global trace ς.

- If $\varsigma = \epsilon$ we have an interaction i such that $\epsilon \in \sigma(i)$. As per Lem.5.4, this implies that $i\downarrow$. As a result, if we start an analysis with (i, ϵ_C) we can immediately apply rule *R1* and we obtain $\omega_C(i, \epsilon_C) = Pass$. Then, given that $\mathbf{proj}_C(\epsilon) = \epsilon_C$ the property holds.
- Let us then consider ς of the form $a.\varsigma'$. We then denote by $\mu = \mathbf{proj}_C(\varsigma)$ and $\mu' = \mathbf{proj}_C(\varsigma')$ the projections and we have, by definition that $\mu = a \overrightarrow{\odot}_C \mu'$. The hypothesis is that we have an interaction i such that $a.\varsigma' \in \sigma(i)$. Then:
  - As per the operational formulation of the semantics of interactions, this implies the existence of a follow-up interaction $i' \in \mathbb{I}_\Omega$ such that $i \xrightarrow{a} i'$ and $\varsigma' \in \sigma(i')$. Also, given that we have $\mu = a \overrightarrow{\odot}_C \mu'$, we can apply rule *R3* from the node (i, μ) so that we have $(i, \mu) \rightsquigarrow (i', \mu')$.
  - Given that $|\varsigma'| < |\varsigma|$, we can apply the induction hypothesis (on a strictly smaller global trace) such that we obtain $\omega_C(i', \mu') = Pass$. This implies the existence of a path $(i', \mu') \overset{*}{\rightsquigarrow} Cov$ in the analysis graph.

  We then observe that $(i, \mu) \rightsquigarrow (i', \mu') \overset{*}{\rightsquigarrow} Cov$ and hence $(i, \mu) \overset{*}{\rightsquigarrow} Cov$ and therefore $\omega_C(i, \mu) = Pass$

□

The second step is given by Th.9.2 and states that whenever a multi-trace μ yields *Pass* when analyzed w.r.t. an interaction i, then this means that it is in fact the projection of a global trace accepted by i.

Theorem 9.2: *Pass* implies Accepted

For any interaction $i \in \mathbb{I}_\Omega$ and any multi-trace $\mu \in \mathbb{T}_{\Omega|C}$:

$$\Big(\omega_C(i, \mu) = Pass \Big) \Rightarrow \left(\exists\, \varsigma \in \mathbb{T}_\Omega \text{ s.t. } \begin{Bmatrix} (\mathbf{proj}_C(\varsigma) = \mu) \\ \wedge\ (\varsigma \in \sigma(i)) \end{Bmatrix} \right)$$

Proof. Let us reason by induction on the size of multi-trace μ, i.e. on $|\mu|$.

- If $|\mu| = 0$ then μ is the empty multi-trace ϵ_C. Let us then consider an interaction i such that $\omega_C(i, \epsilon_C) = Pass$. Since $\omega_C(i, \epsilon_C) = Pass$ given that is is not possible to apply rule *R3*, the only possible option to obtain *Pass* is that rule *R1* can apply. This then implies that $i\downarrow$. Then, as per Lem.5.4 this implies that $\epsilon \in \sigma(i)$, and given that $\mathbf{proj}_C(\epsilon) = \epsilon_C$, the property holds.

- Let us then consider μ of size $z+1$ i.e. $|\mu| = z+1$ with $z \geq 0$ and let us suppose that the property holds for any multi-trace of size z.

  Let us then consider an interaction i such that $\omega_C(i,\mu) = Pass$. Since $\omega_C(i,\mu) = Pass$, there exists a path $(i,\mu) \overset{*}{\rightsquigarrow} Cov$. Also, given that $|\mu| > 0$ we have $\mu \neq \epsilon_C$ therefore rule $R1$ cannot immediately apply. The only possibility to have a path towards Cov is that rule $R3$ applies. This consists in the existence of an action a, a multi-trace μ' and an interaction i' such that $i \xrightarrow{a} i'$ and $\mu = a \overset{\rightarrow}{\odot}_C \mu'$ so that we have a transition $(i,\mu) \rightsquigarrow (i',\mu')$ and then $(i',\mu') \overset{*}{\rightsquigarrow} Cov$.

  Given that applying rule $R3$ involves the consumption of action a in μ, we have $|\mu'| = |\mu| - 1 = z$.

  Therefore we can apply the induction hypothesis which implies the existence of a global trace $\varsigma' \in \mathbb{T}_\Omega$ such that $\mathbf{proj}_C(\varsigma') = \mu'$ and $\varsigma' \in \sigma(i')$.

  Then, given that we have both $i \xrightarrow{a} i'$ and $\varsigma' \in \sigma(i')$, by definition, $a.\varsigma' \in \sigma(i)$.

  Let us then denote $\varsigma = a.\varsigma'$.

  Then, by definition, we have $\mathbf{proj}_C(\varsigma) = \mathbf{proj}_C(a.\varsigma') = a \overset{\rightarrow}{\odot}_C \mu' = \mu$.

  We have therefore found $\varsigma \in \mathbb{T}_\Omega$ such that $\mathbf{proj}_C(\varsigma) = \mu$ and $\varsigma \in \sigma(i)$.

□

The two theorems Th.9.1 and Th.9.2 demonstrate that $\omega_C(i,\mu) = Pass$ characterizes the membership of a multi-trace μ to $\sigma_{|C}(i)$.

The computational cost of ω_C varies greatly depending on the initial (i,μ) couple. It can be reduced substantially in certain cases via the use of some heuristics to guide the exploration of the analysis graph and by stopping the analysis once a Cov local verdict is reached. Another technique which drastically reduce this cost will be detailed in Chap.10.

However, in the worst cases (and without the technique from Chap.10), the complexity costs remain high. In the following section we will discuss the complexity class of the underlying problem which is that of the membership of a multi-trace. We will demonstrate the NP-hardness of this membership problem through a reduction of the 1-in-3-SAT problem [118]. This discussion is inspired by [26, 60, 50].

9.3 Complexity class of the multi-trace analysis problem

In this section we will prove the NP-hardness of the problem of analysing multi-traces w.r.t. interactions in the worst case, that is when the partition upon which multi-traces are defined is the discrete partition $C = (\{l\})_{l \in L}$. To do so we will explain a process to reduce instances of a certain Boolean satisfiability problem, which is known to be NP-complete, into instances of the analysis of a multi-trace against an interaction.

- in Sec.9.3.1 we present this Boolean satisfiability problem which is called 1-in-3-SAT,
- in Sec.9.3.2 we present the reduction of a particular instance of 1-in-3-SAT into a particular instance of multi-trace analysis,

- in Sec.9.3.3 we generalize this process for any instance of 1-in-3-SAT and we conclude.

9.3.1 The 1-in-3-SAT problem

Boolean variables and formulae

A Boolean variable is a symbol or placeholder for a Boolean truth value i.e. either true, which we denote by $\top$ or false, which we denote by $\bot$. A Boolean expression also called a propositional logic formula or more simply a formula is an expression that is build inductively:

- from some Boolean variables from a set V of Boolean variables
- and using the following three truth-functional connectives:
  - the conjunction AND, denoted by $\wedge$ such that for any formulae ϕ_1 and ϕ_2, $\phi_1 \wedge \phi_2$ is also a formula
  - the disjunction OR, denoted by $\vee$ such that for any formulae ϕ_1 and ϕ_2, $\phi_1 \vee \phi_2$ is also a formula
  - the negation NOT, denoted by $\neg$ such that for any formula ϕ, $\neg\phi$ is also a formula. For any given variable $v \in V$ we may denote by $\overline{v}$ the negation $\neg v$ of v.

We may call a literal a formula that is reduced to a single variable $v \in V$ or its negation $\overline{v}$. In the first case a literal v may be called a positive literal and in the case, a literal $\overline{v}$ may be called a negative literal. As a result, the set of variables V is also the set of all positive literals. We may then denote by $\overline{V}$ the set $\{\overline{v} \mid v \in V\}$ of all negative literals.

Satisfying a Boolean formula

An interpretation of a set of Boolean variables V is a mapping $\rho : V \to \{\top, \bot\}$ that assigns a truth value to each variable. A Boolean satisfiability problem is a problem which consists in finding an interpretation of some Boolean variables that satisfies some conditions.

For instance, given a set $V = \{v_1, v_2\}$ containing two Boolean variables, and given a formula $\phi = v_1 \wedge v_2$, finding an interpretation $\rho : V \to \{\top, \bot\}$ of the variables, that satisfies ϕ, which we denote by $\rho \models \phi$ consists in finding values for v_1 and v_2 such that $v_1 \wedge v_2$ equates $\top$. The problem of finding ρ is a Boolean satisfiability problem. In this example, $\rho_0 = [v_1 \to \top,\ v_2 \to \top]$ is a solution such that $\rho \models \phi$. Given the existence of ρ_0, we say that ϕ is satisfiable. If such an interpretation (assignement of variables) do not exist, we can say that ϕ is unsatisfiable.

The 1-in-3-SAT problem

The 1-in-3-SAT problem [118] is a particular Boolean satisfiability problem in which the formula ϕ to satisfy has a particular form and in which the solution interpretation ρ must satisfy some additional properties.

Let us start by introducing some intermediate notions:

- we call a clause a formula ψ that is a disjunction of literals (or a single literal). This means that a clause is a formula ψ of the form $x_1 \vee \cdots \vee x_n$ with $n \in \mathbb{N}^+$ and $(x_j)_{j\in[1,n]}$ a collection of literals from $V \cup \overline{V}$
- we say that a formula ϕ is in Conjunctive Normal Form (or that ϕ is a CNF formula) if it is a conjunction of clauses (or a single clause). This means that there exists a family $(\psi_j)_{j\in[1,n]}$ of $n \in \mathbb{N}^+$ clauses such that $\phi = \psi_1 \wedge \cdots \wedge \psi_n$
- for any integer $k \in \mathbb{N}^+$, we say that a formula is k-CNF if it is a CNF formula in which each clause has exactly k literals

The 1-in-3 SAT problem then corresponds to finding an interpretation ρ of V that satisfy a 3-CNF formula ϕ and such that for any clause of ϕ, only one in the three literal is set to $\top$ by ρ.

In the following we will use the following notations:

- $V = \{v_1, \cdots, v_p\}$ meaning we consider $p \in \mathbb{N}^+$ to be the cardinal of the set V of variables and we index those variables with $[1, p]$
- $\phi = \psi_1 \wedge \cdots \wedge \psi_q$ denotes the 3-CNF formula of the 1-in-3-SAT problem meaning that we consider $q \in \mathbb{N}^+$ to be the number of clauses in ϕ and we index those clauses with $[1, q]$ with, for any $j \in [1, q]$:
  - the clause ψ_j being of the form $\alpha_j \vee \beta_j \vee \gamma_j$ with all three literals in $V \cup \overline{V}$
  - and for any given solution ρ of the problem, only one of either $\rho(\alpha_j)$, $\rho(\beta_j)$ or $\rho(\gamma_j)$ is set to $\top$

Then:

- in Sec.9.3.2 we will detail a reduction of a particular instance of 1-in-3-SAT with $p = 4$ and $q = 2$ to the problem of analysing a particular multi-trace against a particular interaction
- in Sec.9.3.3 we generalize the reduction for any 1-in-3-SAT problem with $p \in \mathbb{N}^+$ and $q \in \mathbb{N}^+$

9.3.2 Reduction on a particular instance of the 1-in-3-SAT problem

Let us consider a 1-in-3-SAT problem in the case where $p = 4$ and $q = 2$.

From formula $\phi = \psi_1 \wedge \psi_2$, we define a certain interaction i. Given that $q = 2$, this interaction will be defined up to the signature $\Omega = (L, M)$ with $L = \{l_1, l_2\}$, with one lifeline per clause, and $M = \{m\}$.

This interaction i is then of the form exemplified on Fig.9.6 which is that of a parallelisation of 4 alternatives $alt(i_v, i_{\overline{v}})$ for any of the $p = 4$ variables v_1, v_2, v_3 and v_4 from V. Those 8 sub-interactions i_x for any of the 8 literals x from $V \cup \overline{V}$ are then defined as follows:

- if x occurs in ψ_1 and ψ_2 then $i_x = seq(l_1!m, l_2!m)$
- if x occurs in ψ_1 but not in ψ_2 then $i_x = l_1!m$
- if x occurs in ψ_2 but not in ψ_1 then $i_x = l_2!m$

- if x occurs neither in ψ_1 nor in ψ_2 then $i_x = \varnothing$

Then, the 1-in-3-SAT problem defined by ϕ is equivalent to knowing whether or not the multi-trace $\mu = (l_1!m,\ l_2!m)$ defined over the discrete partition $C = \{\{l_1\}, \{l_2\}\}$ of L belongs to the semantics $\sigma_{|C}(i)$.

Indeed, we can remark the following:

- for each variable v_k the choice of $\rho(v_k) = \top$ or $\rho(v_k) = \bot$ corresponds to the choice of which alternative branch to execute in the sub-interactions $i_k = alt(i_{v_k}, i_{\overline{v_k}})$, with i_{v_k} being chosen if $\rho(v_k) = \top$ and $i_{\overline{v_k}}$ being chosen if $\rho(v_k) = \bot$. The nature of the exclusive alternative constructor alt guarantees that exactly one branch is chosen in any execution of the interaction. The overall structure of interaction i then guarantees that all the 4 choices between the i_{v_k} or $i_{\overline{v_k}}$ are made in any given full execution of the interaction.

- for each clause ψ_j, the component $\mu_{|\{l_j\}} = l_j!m$ of the multi-trace μ is expressed exactly once in any given execution of the interaction i iff exactly one of the sub-interactions i_{α_j}, i_{β_j} or i_{γ_j} is "chosen" during the execution of i. Indeed, if none are chosen then nothing is expressed on lifeline l_j and if more than one are chosen then several instances of $l_j!m$ are expressed on l_j. As a result, the expression of component $\mu_{|\{l_j\}}$ on lifeline l_j is equivalent to the satisfaction of clause ψ_j.

In other words, during the execution of i, given the use of exclusive alternative constructors in $alt(i_v, i_{\overline{v}})$ sub-terms, the choice of either one of the alt branch constitutes an assignment of Boolean variable v. The overall parallel composition then simulates all possible variable assignments (i.e. the search space for ρ).

Then, the satisfaction of ϕ as the conjunction of clauses ψ_1 and ψ_2 in 1-in-3-SAT is equivalent to that of $\mu = (l_1!m,\ l_2!m) \in \sigma_{|C}(i)$. Indeed, the same ρ must be used to solve both ψ_1 and ψ_2 and the same global execution of i must be used to consume both $\mu_{|\{l_1\}}$ and $\mu_{|\{l_2\}}$ exactly.

On a particular formula

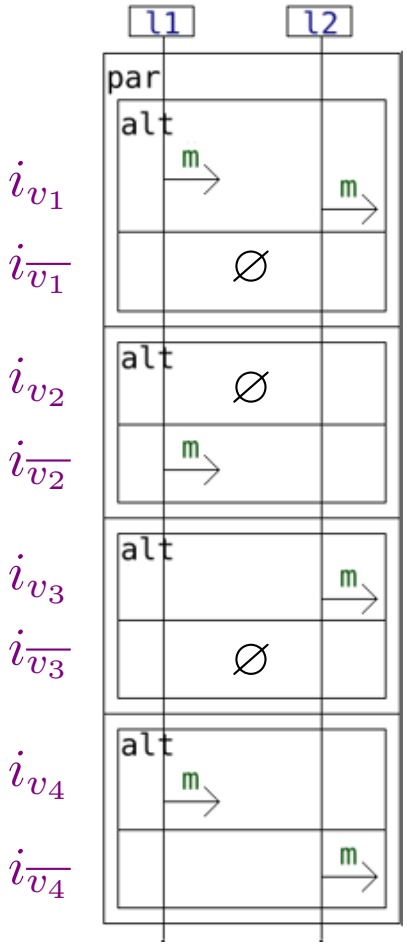
Let us consider the example illustrated on the left of Fig.9.6 which corresponds to an interaction constructed to reflect a specific formula which is:

$$\phi = \psi_1 \wedge \psi_2 = (v_1 \vee \overline{v_2} \vee v_4) \wedge (v_1 \vee v_3 \vee \overline{v_4})$$

In the example, with $\psi_1 = (v_1 \vee \overline{v_2} \vee v_4)$, the fact that $\rho \models \psi_1$ with $\rho : [v_1 \to \bot, v_2 \to \top, v_3 \to \top, v_4 \to \top]$ is equivalent to the fact that $l_1!m$ is expressed exactly once during the execution of i when $i_{\overline{v_1}}$ is chosen over i_{v_1}, i_{v_2} over $i_{\overline{v_2}}$, i_{v_3} over $i_{\overline{v_3}}$, and i_{v_4} over $i_{\overline{v_4}}$.

In the example, $\phi = (v_1 \vee \overline{v_2} \vee v_4) \wedge (v_1 \vee v_3 \vee \overline{v_4})$ is solvable in 1-in-3-SAT by $\rho : [v_1 \to \bot, v_2 \to \top, v_3 \to \top, v_4 \to \top]$. This is equivalent to the fact that $\mu = (l_1!m,\ l_2!m)$ is consumed exactly by the execution of i from Fig.9.6 when $i_{\overline{v_1}}$ is chosen over i_{v_1}, i_{v_2} over $i_{\overline{v_2}}$, i_{v_3} over $i_{\overline{v_3}}$, and i_{v_4} over $i_{\overline{v_4}}$.

For any such 3-CNF formula $\phi = \psi_1 \wedge \psi_2$ defined over $V = \{v_1, v_2, v_3, v_4\}$, the 1-in-3-SAT problem can therefore be reduced to that of the acceptance of $\mu = (l_1!m,\ l_2!m)$ w.r.t. a specific interaction i that can be



Instance of the 1-in-3-SAT problem:
- a set $V = \{v_1, v_2, v_3, v_4\}$ of Boolean variables
- two 3-CNF clauses build on $V \cup \overline{V}$:
  - $C_1 = v_1 \vee \overline{v_2} \vee v_4$
  - and $C_2 = v_1 \vee v_3 \vee \overline{v_4}$
- one formula $\phi = C_1 \wedge C_2$
- finding an interpretation $\rho : V \to \{\top, \bot\}$ such that:
  - $\rho \models \phi$
  - only one literal of C_1 is $\top$
  - only one literal of C_2 is $\top$

Polynomial reduction towards multi-trace analysis:
- signature $\Omega = (L, M)$ with:
  - $L = \{l_1, l_2\}$ one lifeline for each clause
  - $M = \{m\}$
- discrete partition $C = \{\{l_1\}, \{l_2\}\}$ of L
- multi-trace μ s.t. $\mu_{|\{l_1\}} = l_1!m$ and $\mu_{|\{l_2\}} = l_2!m$
- interaction $i = par(i_1, par(i_2, par(i_3, i_4)))$ with:
  - 4 sub-interactions i_k for each variable v_k
  - for any $k \in [1, 4]$, $i_k = alt(i_{v_k}, i_{\overline{v_k}})$
  - for any $x \in V \cup \overline{V}$:
    - $i_x = seq(l_1!m, l_2!m)$ if x appears in both clauses
    - $i_x = l_1!m$ if x only appears in C_1
    - $i_x = l_2!m$ if x only appears in C_2
    - $i_x = \varnothing$ otherwise

Then $\omega_C(i, \mu) = Pass$ iff there is a solution ρ to the 1-in-3-SAT problem.

Figure 9.6: Reduction of a particular instance of 1-in-3-SAT into a multi-trace analysis problem

inferred in polynomial time from the structure of formula ϕ.

As explained earlier, this sketch of proof can be extended to include any numbers p and q of resp. variables and clauses. This will be the object of the following section.

9.3.3 Generalization for any 1-in-3-SAT problem

For practicity, we will use $n-ary$ notations for binary operators $f \in \{strict, seq, par, alt\}$, with $i = f(i_1, \cdots, i_n)$ designating the folding of f s.t. $i = f(i_1, f(\cdots, f(i_{n-1}, i_n)\cdots))$. We now reduce the 1-in-3-SAT Boolean satisfiability problem to multitrace membership so as to prove the NP-hardness of the latter.

Let us consider a 3-CNF formula $\phi = \psi_1 \wedge \cdots \wedge \psi_q$ defined over a set $V = \{v_1, \cdots, v_p\}$ of Boolean variables. ϕ being a 3-CNF formula, for any $j \in [1, q]$, ψ_j is a disjunction of 3 literals $\alpha_j \vee \beta_j \vee \gamma_j$ from $V \cup \overline{V}$.

The 1-in-3 SAT problem consists in finding an interpretation $\rho : V \to \{\top, \bot\}$ s.t. for every clause ψ_j only one of either α_j, β_j or γ_j is set to $\top$.

We will in the following reduce this 1-in-3-SAT problem into a multi-trace analysis problem. This problem will be defined over a signature $\Omega = (L, M)$ with:

- $M = \{m\}$
- $L = \{l_1, \cdots, l_q\}$ with q lifelines, one for each clause
- $C = \{\{l\} \mid l \in L\}$ the discrete partition of L

The multi-trace we will consider is $\mu \in \mathbb{T}_{\Omega|C}$ such that for any $l \in L$, $\mu_{|\{l\}} = l!m$.

For any $k \in [1, p]$ let us define $i_k = alt(i_{v_k}, i_{\overline{v_k}})$ with:

- $i_{v_k} = seq(i^1_{v_k}, \cdots, i^q_{v_k})$
- and $i_{\overline{v_k}} = seq(i^1_{\overline{v_k}}, \cdots, i^q_{\overline{v_k}})$
- such that for any $j \in [1, q]$:
  - if v_k occurs in C_j then $i^j_{v_k} = l_j!m$ and else $i^j_{v_k} = \varnothing$
  - if $\overline{v_k}$ occurs in C_j then $i^j_{\overline{v_k}} = l_j!m$ and else $i^j_{\overline{v_k}} = \varnothing$

Let us then consider $i = par(i_1, \cdots, i_p)$ as illustrated on Fig.9.7. For instance, given $\psi_1 = v_1 \vee v_2 \vee \overline{v_p}$ we have $i^1_{v_1} = l_1!m$, $i^1_{\overline{v_1}} = \varnothing$, $i^1_{v_2} = l_1!m$, $i^1_{\overline{v_2}} = \varnothing$, $i^1_{\overline{v_p}} = l_1!m$ and $i^1_{v_p} = \varnothing$. Likewise, the other emissions of m drawn in Fig.9.7 correspond to $\psi_2 = \overline{v_1} \vee v_2 \vee v_p$ and $\psi_q = v_1 \vee \overline{v_2} \vee \overline{v_p}$.

The 1-in-3-SAT problem defined by ϕ is then equivalent to knowing whether or not $\mu \in \sigma_{|C}(i)$.

Indeed, for any component $\mu_{|\{l_j\}} = l_j!m$ of μ, $\mu_{|\{l_j\}}$ is expressed exactly once iff exactly one of the sub-interactions i_{α_j}, i_{β_j} or i_{γ_j} is chosen during the execution of i (choice w.r.t. their respective parent alt operator). The satisfaction of component $\mu_{|\{l_j\}}$ is therefore equivalent to that of clause ψ_j.

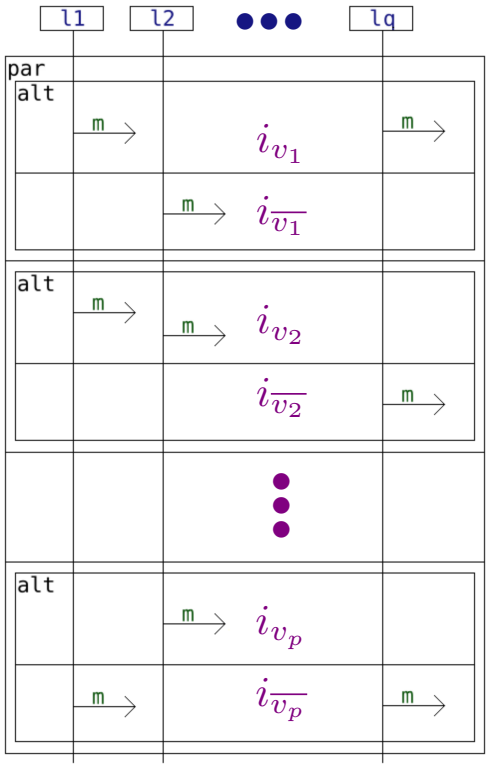


Figure 9.7: Principle of reducing 1-in-3-SAT instances into multi-trace analysis problems

For instance, on the example from Fig.9.7, $\mu_{|\{l_1\}} = l_1!m$ is satisfied iff only one of i_{v_1}, i_{v_2} or $i_{\overline{v_p}}$ is chosen on their respective alternative branches, which exactly corresponds to the satisfaction of $\psi_1 = v_1 \vee v_2 \vee \overline{v_p}$ in 1-in-3-SAT,

In other words, during the execution of i, given the use of exclusive alternative operators in $alt(i_v, i_{\overline{v}})$ sub-terms, the choice of either one of the *alt* branch constitutes an assignment of Boolean variable v. The overall parallel composition then simulates all possible variable assignment (i.e. the search space for ρ).

Then, the satisfaction of ϕ as the conjunction of clauses ψ_j in 1-in-3-SAT is equivalent to that of $\mu \in \sigma_{|C}(i)$, given that the same execution of the model i must satisfy conjointly every component σ_j.

Given that we have identified a case of multi-trace membership equivalent to a NP-complete problem, by reduction, multi-trace membership is NP-hard.

Conclusion

In this chapter, we have defined an algorithm for analysing multi-traces (defined up to any partition of the lifelines into co-localizations) against interaction models. We have proven that this ω_C algorithm solves the membership problem for the semantics of accepted multi-traces $\sigma_{|C}$ (by projection) of interactions that we have defined in Chap.7.

We have presented ω_C through the use of the operational formulation of the semantics with the execution relation $\rightarrow$. However, results from Chap.6 show that we can use the algorithmicized execution semantics for the same results. All definitions related to the execution semantics from Chap.6 are structured by induction on terms and positions, and as such, allow a direct implementation. Similarly, one can implement the ω_C

function by building on-the-fly the sub-graph originating from (i, μ) thanks to queues and usual search heuristics. Moreover, in practice, graph traversals can be interrupted as soon as a Cov verdict is reached. This resulted in the implementation of ω_C (among other features) in the HIBOU tool which we describe in Chap.12.

In this chapter, we have also proven that the problem of analysing multi-traces (for the discrete partition) is NP-Hard. However this level of complexity only concerns some worst case scenarios and is not prohibitive in most cases all the more so that various techniques can be implemented so as to make the process of analysing multi-traces more efficient. Those techniques include (1) the implementation of some heuristics to guide the exploration of the analysis tree, (2) the introduction of stopping criteria on the exploration (once a certain local verdict is reached), (3) the computation of local frontiers to assess whether or not a certain branch of the analysis tree is worth exploring (and if not then it is not explored), etc. Point (3) will be addressed in Chap.10 and implementation details will be described in Chap.12.

In the next chapter we introduce the concept of hiding interaction models and some applications of this hiding operator for the analysis of multi-traces.

Chapter 10

The hiding of interaction terms & applications

Contents

In Chap.9, we have described an algorithm for recognizing (exactly) accepted multi-traces and in Chap.8, we have discussed analyzing multi-traces that were collected from partial observations of executions and might therefore be prefixes of accepted multi-traces. We concluded that simulation steps could be used to replace missing events (i.e. events which were not observed) when analyzing multi-traces obtained from partial observations of executions.

However, in some more extreme cases, large parts of the distributed system may not be observed at all or some sub-systems may remain unobserved for large periods of time resulting in many events being missed. In those situations, it is likely that using simulation steps comes with high overhead costs. It is all the more true when analyzing against interaction models that allow many interleavings or when analyzing incorrect behavior in which case it does not suffice to find one correct path in the graph $\mathbb{G}$ (to ascertain that a behavior is not accepted we have to make sure no path leads to a positive verdict).

A solution to that problem is to restrict the analysis of the multi-trace to a smaller interaction model. Instead of using the whole interaction specification, we use a sub-specification that only concerns the sub-systems that are still observed. This notion of restricting the model to a group of observed sub-systems corresponds to the definition of a "hiding" operator, which is the object of this chapter.

The structure of this chapter is as follows:

- in Sec.10.1 we define the hiding operator and its counterpart in the world of multi-traces.
- in Sec.10.2 we define an algorithm which solves the membership problem for the $\overline{@_{\check{L}}}$ semantics which is the prefix closure of the algebraic multi-trace semantics $@_{\check{L}}$ (which we have defined in Chap.7). Let us also recall that, on the discrete partition of lifelines $\check{L}$, this semantics is equivalent to $\sigma_{|\check{L}}$. As a result, this algorithm equally recognizes $\overline{\sigma_{|\check{L}}}$. The algorithm makes use of hiding steps to erase parts of the interaction model once observation have ceased on those parts. The correctness of this algorithm is formally proven i.e. that it indeed recognizes $\overline{@_{\check{L}}} = \overline{\sigma_{|\check{L}}}$.
- finally, in Sec.10.3, going back to the general case of any partition C, we define the concept of local frontiers and describe how they can be used to improve the previously defined analysis algorithms by allowing the elimination of some branches of the analysis tree.

10.1 Hiding and elimination

10.1.1 Hiding operator on interactions

Hiding an interaction simply consists in willfully ignoring the existence of some lifelines. Let us at first consider the example from Fig.10.1. On the left is described a certain interaction i defined up to the signature $\Omega = (L, M)$ where $L = \{l_1, l_2, l_3\}$ and $M = \{m_1, m_2, m_3, m_4\}$. On that left column we have, at the top the diagram representation of interaction i and on the bottom its corresponding syntax tree. In the middle is represented the hiding of lifeline l_2 from interaction i. On the diagram representation it simply

consists in masking the area of the diagram describing lifeline l_2 and in the syntax, it consists in removing (i.e. replacing by the empty interaction $\varnothing$) all leaf actions occurring on l_2 (as represented by the red crosses). What remains after the application of hiding, is an interaction in which no action occur on the hidden lifeline (here l_2). On the right side of Fig.10.1 is represented the diagram representation of that interaction and a term which is a simplified version (same class of equivalence as per $\approx_E$ from Chap.4) obtained using hiding with simplification steps (by analogy to the results from Chap.6).

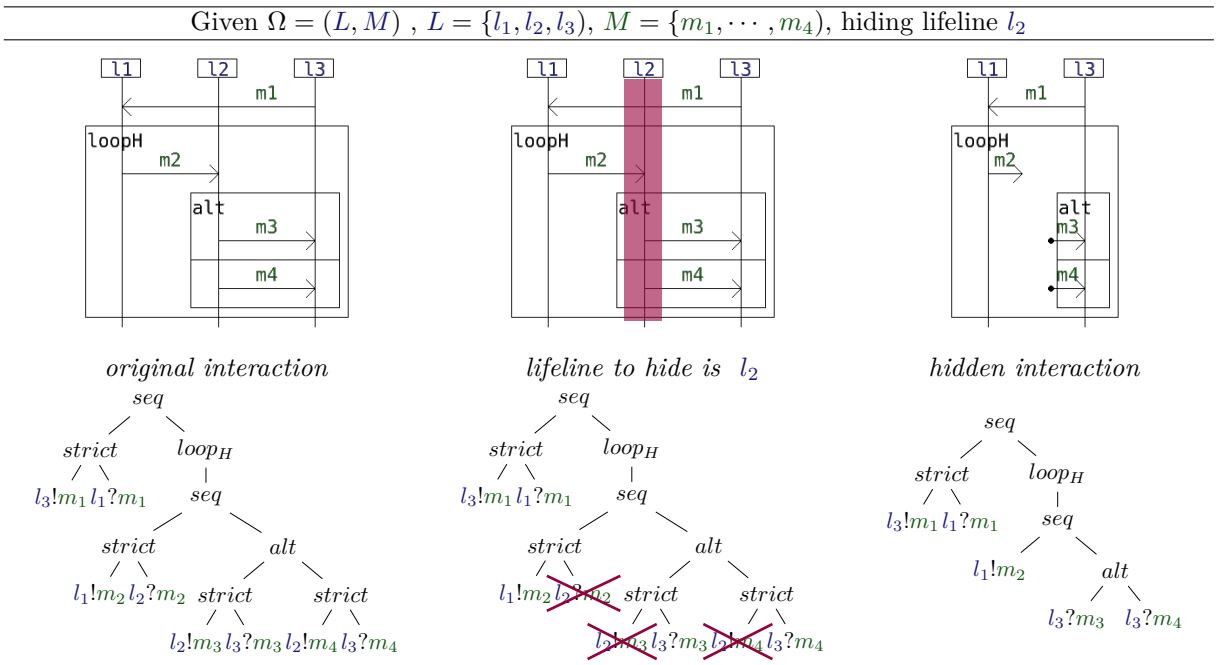


Figure 10.1: Hiding a single lifeline on an example

We can then extend the notion of hiding in the general case of co-localizations. Indeed, for any given interaction i and co-localization c we can hide all lifelines in c from i. This process is described on an example on Fig.10.2. Here, the interaction is defined up to the signature $\Omega = (L, M)$ where $L = \{l_1, l_2, l_3, l_4, l_5\}$ and $M = \{m_1, m_2, m_3, m_4, m_5, m_6\}$. We then consider that the set of lifelines is partitioned into three co-localizations $c_1 = \{l_1, l_2\}$, $c_2 = \{l_3, l_4\}$ and $c_3 = \{l_5\}$ such that $C = \{c_1, c_2, c_3\} \in \text{Part}(L)$. On the left part of Fig.10.2 is given the initial interaction i and the process of hiding the co-localization $c_2 = \{l_3, l_4\}$. On the right part of Fig.10.2 is given the resulting interaction term after hiding and simplification.

The principle of hiding a co-localization is the same as that of hiding a lifeline. Instead of removing actions whenever they occur on a given lifeline, we remove them whenever they occur on a lifeline that is included in a given co-localization.

The process of hiding a lifeline from an interaction enjoys interesting properties. For instance, hiding commutes i.e. it is the same to hide at first a lifeline l_1 and then l_2 than the other way around. We will also see that the semantics of hidden interactions can be characterized. In the following we formalize hiding as a signature projection.

The reader may have noticed that, whenever we hide a lifeline, the resulting interaction does not contain any action occurring on that lifeline. As a result, if $i \in \mathbb{I}_\Omega$, with $\Omega = (L, M)$, then, given $l \in L$, the

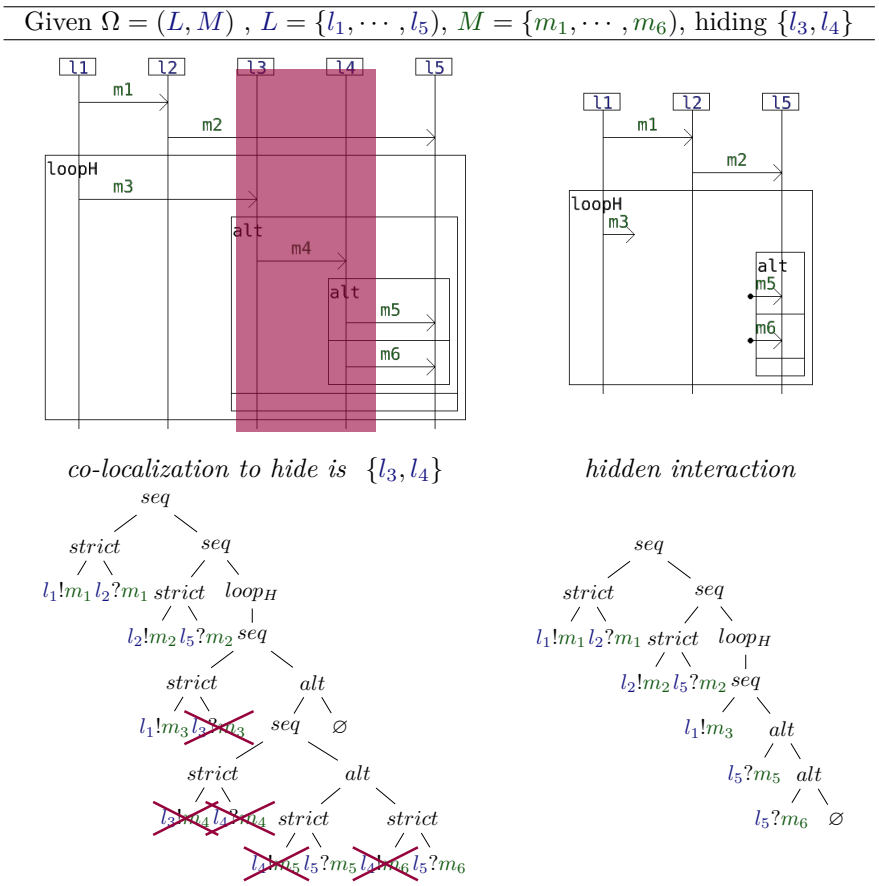


Figure 10.2: Hiding a co-localization on an example

interaction resulting from the hiding of l from i is in $\mathbb{I}_{\Omega'}$ with $\Omega' = (L', M)$ and $L' = L \setminus \{l\}$.

Therefore, hiding can be described as a projection between $\mathbb{I}_\Omega$ and $\mathbb{I}_{\Omega'}$. For the sake of practicity, in the following we will fix the set M of messages and we will simply denote by $\mathbb{I}_L$ interactions defined on (L, M). Likewise, we extend those notations to actions, traces, and multi-traces.

In Def.10.1 we define hiding as a projection operator.

Definition 10.1: Hiding as a projection operator

For any set L and any $h \in \mathcal{P}(L)$ we define: $\mathbf{hide}_h : \mathbb{I}_L \to \mathbb{I}_{L \setminus h}$ the function s.t. for any $i \in \mathbb{I}_L$:

- $\mathbf{hide}_h(i) =$ **match** i **with**

$$
\begin{array}{lll}
\mid \varnothing & \to & \varnothing \\
\mid a \in \mathbb{A}_\Omega & \to & \begin{cases} a & \textbf{if } \theta(a) \notin h \\ \varnothing & \textbf{if } \theta(a) \in h \end{cases} \\
\mid f(i_1, i_2) & \to & f(\mathbf{hide}_h(i_1), \mathbf{hide}_h(i_2)) \textbf{ for } f \in \{strict, seq, alt, par\} \\
\mid loop_k(i_1) & \to & loop_k(\mathbf{hide}_h(i_1)) \textbf{ for } k \in \{S, H, W, P\}
\end{array}
$$

10.1.2 Elimination operator on multi-traces

We can remark that the hiding operator in the algebra of interaction terms is the dual of a certain operator in the algebra of sets of multi-traces. This operator consists of the elimination of a trace component. We define it in Def.10.2.

Definition 10.2: Elimination as a projection operator

For any set L, any partition $C \in \text{Part}(L)$ and any $h \in C$ we define: $\mathbf{elim}_h : \mathbb{T}_{L|C} \to \mathbb{I}_{L\setminus h|C\setminus\{h\}}$ the function s.t. for any $\mu \in \mathbb{I}_{L|C}$:

$$\mathbf{elim}_h(\mu) = (\mu_{|c})_{c\in C\setminus\{h\}}$$

We extend it to sets of multi-traces $\mathbf{elim}_h : \mathcal{P}(\mathbb{T}_{L|C}) \to \mathcal{P}(\mathbb{I}_{L\setminus h|C\setminus\{h\}})$ s.t. for any $T \in \mathcal{P}(\mathbb{I}_{L|C})$:

$$\mathbf{elim}_h(T) = \{\mathbf{elim}_h(\mu) \mid \mu \in T\}$$

The function $\mathbf{elim}_h$ is well defined given that $C \setminus \{h\}$ is indeed a partition of $L \setminus h$.

In fact, the elimination operator $\mathbf{elim}_h$ erases the trace component $\mu_{|h}$ from μ. In order to simplify the notations from that point onwards, for any $h \in C$, we will simply denote by L' the set $L \setminus h$ and by C' the set $C \setminus \{h\}$ so that the signature of $\mathbf{elim}_h$ is $\mathbf{elim}_h : \mathcal{P}(\mathbb{T}_{L|C}) \to \mathcal{P}(\mathbb{T}_{L'|C'})$.

$\mathbf{elim}_h$ preserves the algebraic structures between the $\mathcal{F}$-algebras $\mathcal{A}_C$ and $\mathcal{A}_{C'}$. In the following, in the same manner as what we did in Chap.7 for the projection operator $\mathbf{proj}_C$, we will explain how algebraic structures are preserved.

Lemma 10.1: Elimination preserves sequencing & interleaving

For any multi-traces μ_1 and μ_2 from $\mathbb{T}_{L|C}$:

$$\mathbf{elim}_h(\mu_1 \odot_C \mu_2) = \mathbf{elim}_h(\mu_1) \odot_{C'} \mathbf{elim}_h(\mu_2)$$
$$\mathbf{elim}_h(\mu_1 \otimes_C \mu_2) = \mathbf{elim}_h(\mu_1) \otimes_{C'} \mathbf{elim}_h(\mu_2)$$
$$\mathbf{elim}_h(\mu_1 \parr_C \mu_2) = \mathbf{elim}_h(\mu_1) \parr_{C'} \mathbf{elim}_h(\mu_2)$$

And hence, for any sets of multi-traces T_1 and T_2 from $\mathcal{P}(\mathbb{T}_{L|C})$:

$$\mathbf{elim}_h(T_1 \odot_C T_2) = \mathbf{elim}_h(T_1) \odot_{C'} \mathbf{elim}_h(T_2)$$
$$\mathbf{elim}_h(T_1 \otimes_C T_2) = \mathbf{elim}_h(T_1) \otimes_{C'} \mathbf{elim}_h(T_2)$$
$$\mathbf{elim}_h(T_1 \parr_C T_2) = \mathbf{elim}_h(T_1) \parr_{C'} \mathbf{elim}_h(T_2)$$

Proof. Let us consider multi-traces μ_1 and μ_2. We then have:

$$
\begin{aligned}
\mathbf{elim}_h(\mu_1 \,⦂_C\, \mu_2) &= \mathbf{elim}_h((\mu_{1|c}; \mu_{2|c})_{c\in C}) \\
&= (\mu_{1|c}; \mu_{2|c})_{c\in C'} \\
&= \mathbf{elim}_h(\mu_1) \,⦂_{C'}\, \mathbf{elim}_h(\mu_2) \\
\mathbf{elim}_h(\mu_1 \circledast_C \mu_2) &= \mathbf{elim}_h((\mu_{1|c} ;\!\ast\, \mu_{2|c})_{c\in C}) \\
&= (\mu_{1|c} ;\!\ast\, \mu_{2|c})_{c\in C'} \\
&= \mathbf{elim}_h(\mu_1) \circledast_{C'} \mathbf{elim}_h(\mu_2) \\
\mathbf{elim}_h(\mu_1 \,⫴_C\, \mu_2) &= \mathbf{elim}_h((\mu_{1|c} || \mu_{2|c})_{c\in C}) \\
&= (\mu_{1|c} || \mu_{2|c})_{c\in C'} \\
&= \mathbf{elim}_h(\mu_1) \,⫴_{C'}\, \mathbf{elim}_h(\mu_2)
\end{aligned}
$$

□

Given that the algebraic structures of strict and weak sequencing and interleaving are preserved by the elimination operator, it comes that repetitions of those structures with their Kleene and Head-First closures are also preserved by the elimination operator. We state this in Lem.10.2.

Lemma 10.2: Elimination preserves K-closures and weak HF-closure

For any set of multi-traces T from $\mathcal{P}(\mathbb{T}_{L|C})$:

$$
\begin{aligned}
\mathbf{elim}_h(T^{⦂_C *}) &= \mathbf{elim}_h(T)^{⦂_{C'} *} & \mathbf{elim}_h(T^{\circledast^{\curvearrowleft}_C *}) &= \mathbf{elim}_h(T)^{\circledast^{\curvearrowleft}_{C'} *} \\
\mathbf{elim}_h(T^{\circledast_C *}) &= \mathbf{elim}_h(T)^{\circledast_{C'} *} & \mathbf{elim}_h(T^{⫴_C *}) &= \mathbf{elim}_h(T)^{⫴_{C'} *}
\end{aligned}
$$

Proof. Implied by Lem.10.1 □

As a result, the elimination operator preserves all the algebraic structures that are defined in the $\mathcal{F}$-algebra $\mathcal{A}_C$ of sets of multi-traces up to partition C.

A last interesting property on **elim** is its relationship w.r.t. the multi-trace prefix closure. We state this property in Lem.10.3.

Lemma 10.3: Elimination and prefix closure

For any multi-trace $\mu \in \mathbb{T}_{\Omega|C}$, any set of multi-traces $T \in \mathcal{P}(\mathbb{T}_{\Omega|C})$ and any co-localization $h \in C$:

$$
\left(\begin{array}{c} (\mathbf{elim}_h(\mu) \in \mathbf{elim}_h(T)) \\ \wedge\ (\mu_{|h} = \epsilon) \end{array}\right) \Rightarrow (\mu \in \overline{T})
$$

Proof. If $\mathbf{elim}_h(\mu) \in \mathbf{elim}_h(T)$ this means that $(\mu_c)_{c\in C\setminus\{h\}} \in \mathbf{elim}_h(T)$. Then, there must exist a trace component $t \in \mathbb{T}_{\Omega|h}$ such that a multi-trace $\mu_0 \in T$ verifies $\forall\ c \in C \setminus \{h\}$, $\mu_{0|c} = \mu_{|c}$ and $\mu_{0|h} = t$.
Let us then consider the multi-trace μ_1 such that $\forall\ c \in C \setminus \{h\}$, $\mu_{1|c} = \epsilon$ and $\mu_{1|h} = t$. We then have,

because $\mu_{|h} = \epsilon$, that $\mu_0 = \mu \odot_C \mu_1$ and hence μ is a prefix (in the sense of multi-traces) of $\mu_0 \in T$. Therefore $\mu \in \overline{T}$. □

10.1.3 Duality of hiding and elimination

In Chap.7, we have seen that $@_C$ is a homomorphism between the algebra of interaction terms $(\mathbb{I}_L, \mathcal{F})$ and the algebra of sets of multi-traces $\mathcal{A}_C = (\mathcal{P}(\mathbb{T}_{L|C}), \mathcal{F}^{\mathcal{A}_C})$. This holds for any L and C and therefore it also holds for L' and C' with the notations from this chapter.

In this chapter, we have introduced the $\mathbf{elim}_h$ operator. We have seen that $\mathbf{elim}_h$ preserves the algebraic structures of the algebra of sets of multi-traces $\mathcal{A}_C$ into $\mathcal{A}_{C'}$. Therefore it is a homomorphism between $\mathcal{A}_C$ and $\mathcal{A}_{C'}$.

We have also introduced the $\mathbf{hide}_h$ operator. This operator trivially preserves the algebraic structures of $(\mathbb{I}_L, \mathcal{F})$ into $(\mathbb{I}_{L'}, \mathcal{F})$. It suffices to consider its inductive definition in which the operation symbols from $\mathcal{F}$ of arity $n > 0$ are always left untouched. As a result, it is a homomorphism between $(\mathbb{I}_L, \mathcal{F})$ and $(\mathbb{I}_{L'}, \mathcal{F})$.

Theorem 10.1: Characterizing the multi-trace semantics of hidden interactions

For any interaction $i \in \mathbb{I}_L$ and any $h \in C$:

$$@_{C'}(\mathbf{hide}_h(i)) = \mathbf{elim}_h(@_C(i))$$

Proof. Let us reason by induction on the structure of interaction terms:

- $@_{C'}(\mathbf{hide}_h(\varnothing)) = @_{C'}(\varnothing) = \{\epsilon_{C'}\} = \mathbf{elim}_h(\{\epsilon_C\}) = \mathbf{elim}_h(@_C(\varnothing))$
- for any $a \in \mathbb{A}_L$ we have:
  - if $\theta(a) \in h$:

$$\begin{aligned} @_{C'}(\mathbf{hide}_h(a)) &= @_{C'}(\varnothing) \\ &= \{\epsilon_{C'}\} \\ &= \mathbf{elim}_h(\{\epsilon_C\}) \\ &= \mathbf{elim}_h(\{a \overrightarrow{\odot}_C \epsilon_C\}) \\ &= \mathbf{elim}_h(@_C(a)) \end{aligned}$$

  - if $\theta(a) \notin h$:

$$\begin{aligned} @_{C'}(\mathbf{hide}_h(a)) &= @_{C'}(a) \\ &= \{a \overrightarrow{\odot}_{C'} \epsilon_{C'}\} \\ &= \mathbf{elim}_h(\{a \overrightarrow{\odot}_C \epsilon_C\}) \\ &= \mathbf{elim}_h(@_C(a)) \end{aligned}$$

- for any interactions i_1 and i_2 and any $(f, \circledcirc) \in \{(strict, \dot{;}),\ (seq, \circledast),\ (par, \between),\ (alt, \cup)\}$:

$$\begin{array}{rcll} \circledcirc_{C'}(\mathbf{hide}_h(f(i_1,i_2))) & = & \circledcirc_{C'}(f(\mathbf{hide}_h(i_1),\mathbf{hide}_h(i_2))) & \\ & = & \circledcirc_{C'}(\mathbf{hide}_h(i_1)) \circledcirc_{C'} \circledcirc_{C'}(\mathbf{hide}_h(i_2)) & \\ & = & \mathbf{elim}_h(\circledcirc_C(i_1)) \circledcirc_{C'} \mathbf{elim}_h(\circledcirc_C(i_2)) & \text{per induction hypothesis} \\ & = & \mathbf{elim}_h(\circledcirc_C(i_1) \circledcirc_C \circledcirc_C(i_2)) & \\ & = & \mathbf{elim}_h(\circledcirc_C(f(i_1,i_2))) & \end{array}$$

- for any interaction i and any $(k, \circledcirc) \in \{(S, \dot{;}),\ (H, \circledast^{\curvearrowleft}),\ (W, \circledast),\ (P, \between)\}$:

$$\begin{array}{rcll} \circledcirc_{C'}(\mathbf{hide}_h(loop_k(i))) & = & \circledcirc_{C'}(loop_k(\mathbf{hide}_h(i))) & \\ & = & \circledcirc_{C'}(\mathbf{hide}_h(i))^{\circledcirc_{C'}*} & \\ & = & \mathbf{elim}_h(\circledcirc_C(i))^{\circledcirc_{C'}*} & \text{per induction hypothesis} \\ & = & \mathbf{elim}_h(\circledcirc_C(i)^{\circledcirc_C*}) & \\ & = & \mathbf{elim}_h(\circledcirc_C(loop_k(i))) & \end{array}$$

□

$$\begin{array}{ccc} \left(\mathbb{I}_L, \left\{ \begin{array}{l} \varnothing,\ a \in \mathbb{A}_L, \\ alt,\ strict,\ seq,\ par \\ loop_S,\ loop_H,\ loop_W,\ loop_P \end{array} \right\}\right) & \xrightarrow{\quad \mathbf{hide}_h \quad} & \left(\mathbb{I}_{L'}, \left\{ \begin{array}{l} \varnothing, a \text{ if } \theta(a)\in h \text{ else } \varnothing, \\ alt,\ strict,\ seq,\ par \\ loop_S,\ loop_H,\ loop_W,\ loop_P \end{array} \right\}\right) \\ \Big\downarrow \circledcirc_C & & \Big\downarrow \circledcirc_{C'} \\ \left(\mathcal{P}(\mathbb{T}_{L|C}), \left\{ \begin{array}{l} \{\epsilon_C\},\ \{a \overrightarrow{\odot_C} \epsilon_C\}, \\ \cup,\ \dot{;}_C,\ \circledast_C,\ \between_C, \\ \dot{;}_{C*},\ \circledast_C^{\curvearrowleft *},\ \circledast_{C*},\ \between_{C*} \end{array} \right\}\right) & \xrightarrow{\quad \mathbf{elim}_h \quad} & \left(\mathcal{P}(\mathbb{T}_{L'|C'}), \left\{ \begin{array}{l} \{\epsilon_{C'}\},\ \{a \overrightarrow{\odot_{C'}} \epsilon_{C'} \text{ if } \theta(a)\in h \text{ else } \epsilon_{C'}\}, \\ \cup,\ \dot{;}_{C'},\ \circledast_{C'},\ \between_{C'}, \\ \dot{;}_{C'*},\ \circledast_{C'}^{\curvearrowleft *},\ \circledast_{C'*},\ \between_{C'*} \end{array} \right\}\right) \end{array}$$

All four arrows are homomorphisms (preserve algebraic structures) and the diagram commutes.

Figure 10.3: Duality of hiding & elimination and relation w.r.t. the algebraic multi-trace semantics

We illustrate the relationships between the four homomorphisms $\mathbf{elim}_h$, $\circledcirc_C$, $\mathbf{hide}_h$ and $\circledcirc_{C'}$ on Fig.10.3. The result from Th.10.1 implies that the diagram drawn on Fig.10.3 commutes.

10.2 A multi-trace analysis algorithm using hiding steps

In this section we define a multi-trace analysis algorithm so as to recognize elements of $\sigma_{|\check{L}}(i)$ for any interaction term i. The presentation of this algorithm is similar to the simpler algorithm which we have detailed in Chap.9.

10.2.1 Definition of the algorithm

In the following we will denote by $\mathcal{L}$ the infinite universe of lifelines such that for any set of lifelines L, we have $L \subset \mathcal{L}$. We then denote by $\mathbb{I}_{\mathcal{L}}$ the universe of interaction terms and by $\mathbb{T}_{|\check{\mathcal{L}}} = \bigcup_{L \subset \mathcal{L}} \mathbb{T}_{L|\check{L}}$ the universe of multi-traces defined up to the discrete partition of any set L of lifelines.

The new algorithm also relies on four rules, denoted by R_p (for "pass"), R_h (for "hide"), R_e (for "execute") and R_f (for "fail"). Those rules define a directed graph $\mathbb{G}$ in which vertices are either a tuple $(i, \mu) \in \mathbb{I}_{\mathcal{L}} \times \mathbb{T}_{|\check{\mathcal{L}}}$ or an local verdict $v \in \{Obs, Out\}$. We note $\mathbb{V} = \{Obs, Out\} \cup (\mathbb{I}_{\mathcal{L}} \times \mathbb{T}_{|\check{\mathcal{L}}})$ the set of all such vertices.

The formal definition of the graph $\mathbb{G}$ and how to construct it is given on Def.10.3.

Definition 10.3: Rules of Multi-Prefix Analysis

The analysis relation $\rightsquigarrow \subseteq \mathbb{V} \times \mathbb{V}$ is defined as:

$$(R_p)\ \frac{i \qquad \epsilon_{\check{L}}}{Obs} \qquad\qquad (R_h)\ \frac{i \qquad \mu}{\mathbf{hide}_{\{l\}}(i) \qquad \mathbf{elim}_{\{l\}}(\mu)}\ (\mu_{|\{l\}} = \epsilon) \wedge (|\check{L}| > 1)$$

$$(R_e)\ \frac{i \qquad a \overrightarrow{\odot}_{\check{L}} \mu}{i' \qquad \mu}\ i \xrightarrow{a} i' \qquad\qquad (R_f)\ \frac{i \qquad \mu}{Out}\ \begin{cases} (\nexists\ l \in L \text{ s.t. } \mu_{|\{l\}} = \epsilon) \\ \wedge (\nexists\ i \xrightarrow{a} i' \text{ s.t. } \mu = a \overrightarrow{\odot}_{\check{L}} \mu') \end{cases}$$

We can remark that vertices of the form (i, μ) are not sinks. Indeed:

- if μ is the empty multi-trace $\epsilon_{\check{L}}$ then R_p can apply and there exists an outgoing edge from $(i, \epsilon_{\check{L}})$
- if $\mu \neq \epsilon_{\check{L}}$:
  - if a component $\mu_{|\{l\}}$ of μ is empty then rule R_h can apply and there exists an outgoing edge from (i, μ)
  - if there is a match between a frontier action and the head of a component of the multi-trace then rule R_e can apply and there exists an outgoing edge from (i, μ)
  - if neither condition holds then rule R_f can apply and there exists an outgoing edge from (i, μ)

As a result, local verdicts $\{Obs, Out\}$ are the only two sinks of graph $\mathbb{G}$.

Rules R_o and R_f specify edges from vertices of the form (i, μ) to local verdicts.

The rule R_e behaves identically as the rule $R3$ of the algorithm from Chap.9.

The rule R_h specifies edges $(i, \mu) \rightsquigarrow (i', \mu')$ such that, given the existence of a lifeline $l \in L$ on which μ is empty i.e. such that $\mu_{|\{l\}} = \epsilon$, we have $i' = \mathbf{hide}_{\{l\}}(i)$ and $\mu' = \mathbf{elim}_{\{l\}}(\mu)$.

Definition 10.4: A measure for the vertices of the graph

For any vertex $v \in \mathbb{G}$ we define its measure $|v|$ as a tuple of integers such that:

- if $v \in \{Obs, Out\}$ then $|v| = (-1, -1)$
- if $v = (i, \mu)$ then, given the smallest L such that $i \in \mathbb{I}_L$ and $\mu \in \mathbb{T}_{L|\check{L}}$, we have $|v| = (|\mu|, |L|)$ with $|\mu|$ being the length (in number of actions) of the multi-trace μ (as in Chap.9) and $|L|$ being the cardinal of L

For any transition $(i, \mu) \rightsquigarrow (i', \mu')$ in the graph $\mathbb{G}$, we have that $|(i', \mu')| < |(i, \mu)|$ (in lexicographic order). Indeed, given $|(i, \mu)| = (x, y)$ we have:

- either $|(i', \mu')| = (x - 1, y)$ if R_e applies for the transition $(i, \mu) \rightsquigarrow (i', \mu')$
- or $|(i', \mu')| = (x, y - 1)$ if R_h applies for the transition $(i, \mu) \rightsquigarrow (i', \mu')$

Using those notations, each application of R_e decreases x by 1. If x reaches 0, there are no more actions in the multi-trace and hence $\mu = \epsilon_{\check{L}}$ and rule R_p applies. Likewise, each application of R_h decreases y by 1. y cannot reach 0 because once it reaches 1 rule R_h cannot be applied anymore (by definition).

As a result, any outgoing path from a node (i, μ) is finite because there can only be a finite number of applications of R_e and R_h before ultimately reaching either of Obs or Out, both of which are sinks.

Moreover, for any vertex (i, μ) there exists a finite number of outgoing transitions that is bound by $|\texttt{frt}(i)| + |L|$ given that:

- there cannot be more than $|\texttt{frt}(i)|$ different applications of R_e because there cannot be more matches than the cardinal of the frontier of execution $\texttt{frt}(i)$
- there cannot be more than $|L|$ different applications of R_h because there cannot be more than $|L|$ empty trace components on μ

From the two last points, we can conclude that, from any given vertex (i, μ) there is only a finite subgraph of $\mathbb{G}$ that is reachable. Also, as in Chap.9, $\mathbb{G}$ is an acyclic graph. Then, similarly as in Chap.9 we define the verdict of multi-trace analysis in Def.10.5.

Definition 10.5: Multi-Prefix Analysis

For any $L \subset \mathcal{L}$, we define $\overline{\textcircled{\omega}_L} : \mathbb{I}_L \times \mathbb{T}_{L|\check{L}} \to \{Pass, Fail\}$ such that for any $i \in \mathbb{I}_L$ and $\mu \in \mathbb{T}_{L|\check{L}}$ we have:

- $\overline{\textcircled{\omega}_L}(i, \mu) = Pass$ iff there exists a path $(i, \mu) \overset{*}{\rightsquigarrow} Obs$
- $\overline{\textcircled{\omega}_L}(i, \mu) = Fail$ otherwise;
  i.e. for all path $(i, \mu) \overset{*}{\rightsquigarrow} v$ with $v \in \{Obs, Out\}$, then $v = Out$

In the following section, we prove the correctness of this algorithm, i.e. that it exactly recognize all the elements of $\sigma_{\check{L}}(i)$. In other words that we have, for any $i \in \mathbb{I}_L$ and $\mu \in \mathbb{T}_{L|\check{L}}$:

$$(\overline{\textcircled{\omega}_L}(i,\mu) = Pass) \Leftrightarrow (\mu \in \overline{\sigma_{|\check{L}}}(i))$$

10.2.2 Proof of correctness

At first, we need to prove a property of confluence of the graph $\mathbb{G}$ that is explored during the analysis. This property, given in Lem.10.4 states that if, from a given node (i,μ), we can reach Obs by any given means, then, if we can also apply rule R_h so that $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ for any lifeline l, then we can also reach Obs from $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$. This is indeed a property of confluence given that it states that we may take another path, in which we might as well hide lifeline l, so as to reach Obs.

Lemma 10.4: A confluence property for the analysis graph

For any interaction $i \in \mathbb{I}_L$, any action $a \in \mathbb{A}_L$, any multi-trace $\mu \in \mathbb{T}_{L|\check{L}}$ and any lifeline l we have that:

$$\frac{(i,\mu) \overset{*}{\rightsquigarrow} Obs \qquad (i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))}{(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \overset{*}{\rightsquigarrow} Obs}$$

Proof. Let us reason by induction on the measure $|(i,\mu)| = (x,y)$:

- If $x = 0$ then $\mu = \epsilon_{\check{L}}$ and:
  - if $y = 1$ the premise do not hold because we cannot apply R_h
  - if $y > 1$ we have that $(i, \epsilon_{\check{L}}) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \epsilon_{\check{L}'})$ and we can immediately apply rule R_p so that the conclusion holds
- If $y = 1$ the premise do not hold
- If $y = 2$, let us note $L = \{l, l'\}$. Then, if $(i,\mu) \overset{*}{\rightsquigarrow} Obs$, we may have as a first transition in the path:
  - either an application of R_e and in that case there exists a, i' and μ' s.t. $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'$ and $i \xrightarrow{a} i'$ and we have $(i,\mu) \rightsquigarrow (i',\mu') \overset{*}{\rightsquigarrow} Obs$. Then:
    - on the one hand we can apply the induction hypothesis on (i',μ') because we have that $(i',\mu') \rightsquigarrow (\mathbf{hide}_{\{l\}}(i'), \mathbf{elim}_{\{l\}}(\mu'))$ trivially still holds. Then we can conclude that we have $(\mathbf{hide}_{\{l\}}(i'), \mathbf{elim}_{\{l\}}(\mu')) \overset{*}{\rightsquigarrow} Obs$
    - on the other hand, given $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'$, we must have $l \neq \theta(a)$ for the hypothesis $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ to hold. Therefore if a is executable in i then it is also executable in $\mathbf{hide}_{\{l\}}(i)$ and we have $\mathbf{hide}_{\{l\}}(i) \xrightarrow{a} \mathbf{hide}_{\{l\}}(i')$ because $\mathbf{hide}_{\{l\}}$ is a homomorphism and hence preserves the algebraic structures of the Interaction Language. Also, we have that $\mathbf{elim}_{\{l\}}(\mu) = \mathbf{elim}_{\{l\}}(a \overset{\rightarrow}{\odot}_{\check{L}} \mu') = a \overset{\rightarrow}{\odot}_{\check{L}'} \mathbf{elim}_{\{l\}}(\mu')$. This then implies that we can apply R_e from $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ so that $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i'), \mathbf{elim}_{\{l\}}(\mu'))$

The two points above allow to conclude that $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \overset{*}{\rightsquigarrow} Obs$

– or an application of R_h. Given that there are only two lifelines l and l' we have either that:

* it is l that is hidden and hence we have $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \overset{*}{\rightsquigarrow} Obs$ and we can immediately conclude

* or it is l' that is hidden. Then we can remark that the fact that we have both $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l'\}}(i), \mathbf{elim}_{\{l'\}}(\mu))$ and $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ imply that $\mu = \epsilon_{\check{L}}$ and hence we can apply R_p from $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ so that the conclusion holds

- If $x > 1$ and $y > 2$ and we have $(i,\mu) \overset{*}{\rightsquigarrow} Obs$ then we may have as a first transition in the path:

  – either an application of R_e and in that case we can reason in the same way as previously

  – or an application of R_h and in that case there exists a lifeline $l' \in L$ such that we have $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l'\}}(i), \mathbf{elim}_{\{l'\}}(\mu)) \overset{*}{\rightsquigarrow} Obs$ and then:

  * if $l' = l$ we can immediately conclude

  * if $l' \neq l$ then we can remark that:

    · firstly $(\mathbf{hide}_{\{l'\}}(i), \mathbf{elim}_{\{l'\}}(\mu)) \rightsquigarrow (\mathbf{hide}_{\{l\}}(\mathbf{hide}_{\{l'\}}(i)), \mathbf{elim}_{\{l\}}(\mathbf{elim}_{\{l'\}}(\mu)))$ and, given that we have decremented the measure by applying a first time R_h, we can apply the induction hypothesis so that $(\mathbf{hide}_{\{l\}}(\mathbf{hide}_{\{l'\}}(i)), \mathbf{elim}_{\{l\}}(\mathbf{elim}_{\{l'\}}(\mu))) \overset{*}{\rightsquigarrow} Obs$

    · secondly we can remark that
    $\mathbf{hide}_{\{l\}}(\mathbf{hide}_{\{l'\}}(i)) = \mathbf{hide}_{\{l'\}}(\mathbf{hide}_{\{l\}}(i))$
    and $\mathbf{elim}_{\{l\}}(\mathbf{elim}_{\{l'\}}(\mu)) = \mathbf{elim}_{\{l'\}}(\mathbf{elim}_{\{l\}}(\mu))$

    · finally we have:

$$\begin{aligned}(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) &\rightsquigarrow (\mathbf{hide}_{\{l'\}}(\mathbf{hide}_{\{l\}}(i)), \mathbf{elim}_{\{l'\}}(\mathbf{elim}_{\{l\}}(\mu))) \\ &= (\mathbf{hide}_{\{l\}}(\mathbf{hide}_{\{l'\}}(i)), \mathbf{elim}_{\{l\}}(\mathbf{elim}_{\{l'\}}(\mu))) \\ &\overset{*}{\rightsquigarrow} Obs\end{aligned}$$

    and hence the property holds

□

Theorem 10.2: Correctness of multi-prefix analysis via hiding

For any interaction $i \in \mathbb{I}_L$ and any multi-trace $\mu \in \mathbb{T}_{L|\check{L}}$:

$$\left(\mu \in \overline{\sigma_{|\check{L}}}(i) \right) \Leftrightarrow \left(\overline{\textcircled{\omega}_L}(i,\mu) = Pass \right)$$

Proof. Let us reason by induction on the measure $|(i,\mu)| = (x,y)$.

- If $x = 0$ then $\mu = \epsilon_{\check{L}}$ and hence we have both $\overline{@\omega_L}(i,\mu) = Pass$ because rule R_p immediately applies and $\mu \in \overline{\sigma_{|\check{L}}}(i)$ because the empty multi-trace $\epsilon_{\check{L}}$ is in the prefix closure of any non-empty set of multi-traces.

- If $y = 1$ then there is a single co-localization i.e. a single lifeline that is still observed. Then:

  - either $\mu = \epsilon_{\check{L}}$ and we are in the same case as the one above
  - or $\mu \neq \epsilon_{\check{L}}$. Let us then use notations from Chap.5 and use t instead of μ. $t \neq \epsilon$ implies that there exists an action a s.t. $t = a.t'$. Then:

    $\Rightarrow$ if $a.t' \in \overline{\sigma}(i)$ it is a prefix of an accepted trace. Let us then consider t_+ such that $a.t'.t_+ \in \sigma(i)$. This implies that there exists i' s.t. $i \xrightarrow{a} i'$ and $t'.t_+ \in \sigma(i')$ and hence $t' \in \overline{\sigma}(i')$ by prefix-closure. Given that $|(i',t')| < |(i,t)|$ we can apply the induction hypothesis. We then obtain that $(i',t') \overset{*}{\rightsquigarrow} Obs$ which implies $(i,t) \rightsquigarrow (i',t') \overset{*}{\rightsquigarrow} Obs$ and therefore $\overline{@\omega_L}(i,\mu) = Pass$

    $\Leftarrow$ if $\overline{@\omega_L}(i,\mu) = Pass$ then there exists a path $(i, a.t') \overset{*}{\rightsquigarrow} Obs$. Given that we cannot apply R_h because there is only a single co-localization, we must be able to apply R_e. Hence there exists i' s.t. $i \xrightarrow{a} i'$ and $(i,a.t') \rightsquigarrow (i',t') \overset{*}{\rightsquigarrow} Obs$. Then, given that $|(i',t')| < |(i,t)|$ we can apply the induction hypothesis. We then obtain that $t' \in \overline{\sigma}(i')$. This implies the existence of t_+ s.t. $t'.t_+ \in \sigma(i')$, then, given that $i \xrightarrow{a} i'$ this implies that $a.t'.t_+ \in \sigma(i)$ and hence, by prefix-closure that $a.t' \in \overline{\sigma}(i)$

- Let us then consider $x > 0$ and $y > 1$. Then:

  - If there exists a lifeline l s.t. $\mu_{|\{l\}} = \epsilon$ then we can apply rule R_h and we have $(i,\mu) \rightsquigarrow (\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu))$ and then:

    $\Rightarrow$ if $\mu \in \overline{\sigma_{|L}}(i) = \overline{@\sigma_L}(i)$ then, as per Fig.10.3 we have $\mathbf{elim}_{\{l\}}(\mu) \in \overline{\sigma_{|L'}}(\mathbf{hide}_{\{l\}}(i))$ (with $L' = L \setminus \{l\}$). Given that we have decremented the measure, we can apply the induction hypothesis which implies that $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \overset{*}{\rightsquigarrow} Obs$. Then, by transitivity $(i,\mu) \overset{*}{\rightsquigarrow} Obs$ and hence $\overline{@\omega_L}(i,\mu) = Pass$

    $\Leftarrow$ if $\overline{@\omega_C}(i,\mu) = Pass$ we have a path $(i,\mu) \overset{*}{\rightsquigarrow} Obs$ then we can apply Lem.10.4 to obtain that $(\mathbf{hide}_{\{l\}}(i), \mathbf{elim}_{\{l\}}(\mu)) \overset{*}{\rightsquigarrow} Obs$ and we can then apply the induction hypothesis so that $\mathbf{elim}_{\{l\}}(\mu) \in \overline{\sigma_{|L'}}(\mathbf{hide}_{\{l\}}(i)) = \overline{@\sigma_{L'}}(\mathbf{hide}_{\{l\}}(i)) = \mathbf{elim}_{\{l\}}(\overline{@\sigma_L}(i))$. Then, given that $\mu_{|\{l\}} = \epsilon$, we can apply Lem.10.3 to conclude that $\mu \in \overline{\overline{@\sigma_L}(i)} = \overline{@\sigma_L}(i) = \overline{\sigma_{|L}}(i)$

  - If there are no lifeline l s.t. $\mu_{|\{l\}} = \epsilon$ then:

    $\Rightarrow$ if $\mu \in \overline{\sigma_{|L}}(i)$, then there exists μ_+ s.t. $\mu \odot_{\check{L}} \mu_+ \in \sigma_{|\check{L}}(i)$. Then, because $\mu \odot_{\check{L}} \mu_+ \neq \epsilon_{\check{L}}$, as per Lem.7.8 there exists a, i' and μ' s.t. $\mu \odot_{\check{L}} \mu_+ = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'_*$ and $i \xrightarrow{a} i'$ and $\mu'_* \in \sigma_{|\check{L}}(i')$. Then, because, there is no empty trace component on μ action a must be taken from μ and not from μ_+. Therefore there exists μ' and μ'_+ such that $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'$ and $\mu \odot_{\check{L}} \mu_+ = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'_* = (a \overset{\rightarrow}{\odot}_{\check{L}} \mu') \odot_{\check{L}} \mu'_+$ and therefore $\mu' \odot_{\check{L}} \mu'_+ = \mu'_* \in \sigma_{|\check{L}}(i')$. Hence $\mu' \in \overline{\sigma_{|\check{L}}(i')} = \overline{\sigma_{|L}}(i')$. Then:

- on the one hand we can apply the induction hypothesis on i' and μ' so that we have $(i', \mu') \overset{*}{\rightsquigarrow} Obs$
- on the other hand, the fact that $i \xrightarrow{a} i'$ and $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'$ allows us to apply rule R_e so that we have $(i, \mu) \rightsquigarrow (i', \mu')$

From the two last points we conclude by transitivity that $(i, \mu) \overset{*}{\rightsquigarrow} Obs$ and hence the property holds.

$\Leftarrow$ if $\overline{\textcircled{\omega}_L}(i, \mu) = Pass$ we have a path $(i, \mu) \overset{*}{\rightsquigarrow} Obs$ given that we cannot apply rule R_h, the only possible first transition in this path is an application of rule R_e i.e. there must exists a, i' and μ' s.t. $i \xrightarrow{a} i'$ and $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu'$ and $(i, \mu) \rightsquigarrow (i', \mu') \overset{*}{\rightsquigarrow} Obs$. Then:

- on the one hand we can apply the induction hypothesis on i' and μ' so that we have $\mu' \in \overline{\sigma_{|\check{L}}}(i')$ which implies the existence of μ'_+ such that $\mu' \odot_{\check{L}} \mu'_+ \in \sigma_{|\check{L}}(i')$
- on the other hand the fact that $i \xrightarrow{a} i'$ and $\mu' \odot_{\check{L}} \mu'_+ \in \sigma_{|\check{L}}(i')$, as per Lem.7.8 this implies that $a \overset{\rightarrow}{\odot}_{\check{L}} (\mu' \odot_{\check{L}} \mu'_+) \in \sigma_{|C}(i)$. In particular, this implies that $\mu = a \overset{\rightarrow}{\odot}_{\check{L}} \mu' \in \overline{\sigma_{|\check{L}}}(i)$

□

We have therefore defined an algorithm which solves the membership problem for $\overline{\sigma_{|\check{L}}}$.

10.2.3 Illustrative example

Let us consider the example given on Fig.10.4. In this example, we consider $\Omega = (L, M)$ with $L = \{l_1, l_2, l_3\}$ and $M = \{m_1, m_2, m_3\}$ and we analyze a multi-trace μ such that $\mu_{|\{l_1\}} = l_1!m_1.l_1?m_3$, $\mu_{|\{l_2\}} = \epsilon$ and $\mu_{|\{l_3\}} = l_3?m_2$ against an interaction $i = seq(loop_H(\cdots), \cdots)$.

In fact, this multi-trace μ is such that $\mu \odot_{\check{L}} \mu_+ \in \textcircled{\sigma}_{\check{L}}(i)$ with μ_+ such that $\mu_{+|\{l_1\}} = \epsilon$, $\mu_{+|\{l_2\}} = l_2?m_1.l_2!m_2.l_2!m_3$ and $\mu_{+|\{l_3\}} = \epsilon$. And, as a result, $\mu \in \overline{\sigma_{|\check{L}}}(i)$.

In other words, the execution which observation corresponds to μ corresponds to a behavior that is indeed specified by the interaction, with one instanciation of the $loop_H$. However, the observation of that execution has only been partial, given that nothing was observed on lifeline l_2. As a result, the collected multi-trace is a prefix (in the sense of multi-traces) of an accepted multi-trace. It is not however the projection of a prefix of an accepted trace given that the events that are missing do not occur at the end of the globally reordered behavior.

In Fig.10.4 we represent the use of two algorithms to recognize that $\mu \in \overline{\sigma_{|\check{L}}}(i)$. On the left, an algorithm with simulation steps, which we have informally described in Chap.8. And, on the right, the $\overline{\textcircled{\omega}_L}$ algorithm with hiding steps which we have formally defined.

The algorithm based on simulation steps, which application is represented on the left of Fig.10.4, reconstitutes the entire global trace which projects onto $\mu \odot_{\check{L}} \mu_+$. When events of this trace are in μ, this is done via steps in which the action is consumed from μ. And, when this is not the case, simulation steps fill-in the gaps.

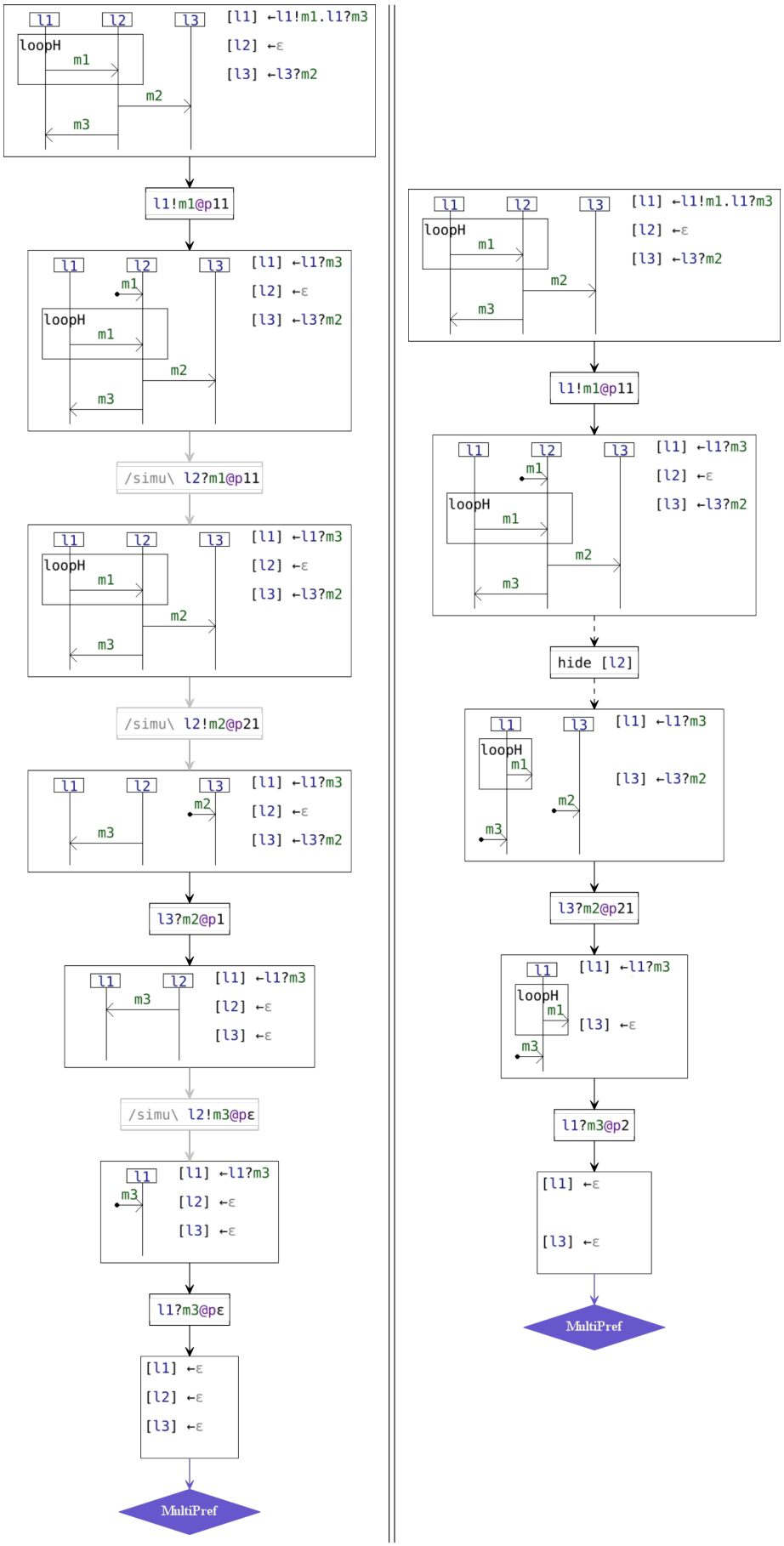


Figure 10.4: Illustrating the use of the multi-prefix analysis algorithm with hiding

Let us then consider the algorithm based on hiding steps, which application is represented on the right of Fig.10.4. We can see that similarly to the first, we can, as a first step, consume action $l_1!m_1$ with a normal execution step. From that second vertex we have no match between head actions of the multi-trace and frontier actions and hence, no action can be consumed. However, we have that the component trace on lifeline l_2 is empty. As a result, we can hide this lifeline on the model. This allows the further consumption of the multi-trace until the same verdict (as the one from the algorithm with simulation steps) is reached.

By contrast to the algorithm with simulation, the algorithm with hiding takes less steps. In this particular example it took only 4 steps instead of 6. However, in more complex cases, we argue that the algorithm with hiding is increasingly more efficient that the one with simulation.

10.3 Local frontiers

Let us now consider the example from Fig.10.5. In this example, the behavior that is analyzed corresponds to a multi-trace μ that is in the semantics $\sigma^{\dagger}_{|\check{L}}(i)$ of the specifying interaction i. Also, we use an algorithm which makes use of hiding steps. From the starting vertex at the top of Fig.10.5 we can see that we have two matches between head actions and frontier actions. Indeed, in the interaction term we have two occurrences of action $l_1!m_1$. As a result, both branches are explored. However, the branch on the left yields Out which means that it cannot consume the multi-trace. The branch on the right however yields $TooShort$ which signifies (as we have seen on Chap.8) that the algorithm recognized that $\mu \in \sigma^{\dagger}_{|\check{L}}(i)$.

In the example analysis represented on Fig.10.5, when there are several matches, the algorithm explores the paths opened-up by matching frontier actions in the lexicographic order of their respective positions in i. Also, we use a Depth-First-Search heuristic to explore the graph $\mathbb{G}$. As a result, the branch on the left of Fig.10.5 is entirely explored before the right branch is. Once we have found $TooShort$ we then decide to stop the analysis even though there may remain some other branches to explore (in this precise example we can identify 5 distinct paths to local verdicts).

More generally, depending on heuristics that might be used to explore the graph $\mathbb{G}$ and depending on whether or not we stop the analysis when reaching a certain local verdict, the number of vertices that are explored during a given analysis may widely vary. Reducing the span of the sub-graph of $\mathbb{G}$ that is explored before concluding the analysis is important because it mechanically reduces the time and memory complexity of the analysis[1].

In the following we propose one such mechanism. It relies on the definition of "local frontiers" which, for any given co-localization $c \in C$, gathers the actions which may be executed on c the next time an action is executed on that co-localization c. In other words, by contrast to the frontier of execution which gathers actions which are immediately executable globally, a local frontier gathers actions which may be executed next locally.

[1] of course mechanisms put in place to reduce this span may also incur some costs w.r.t. time and memory

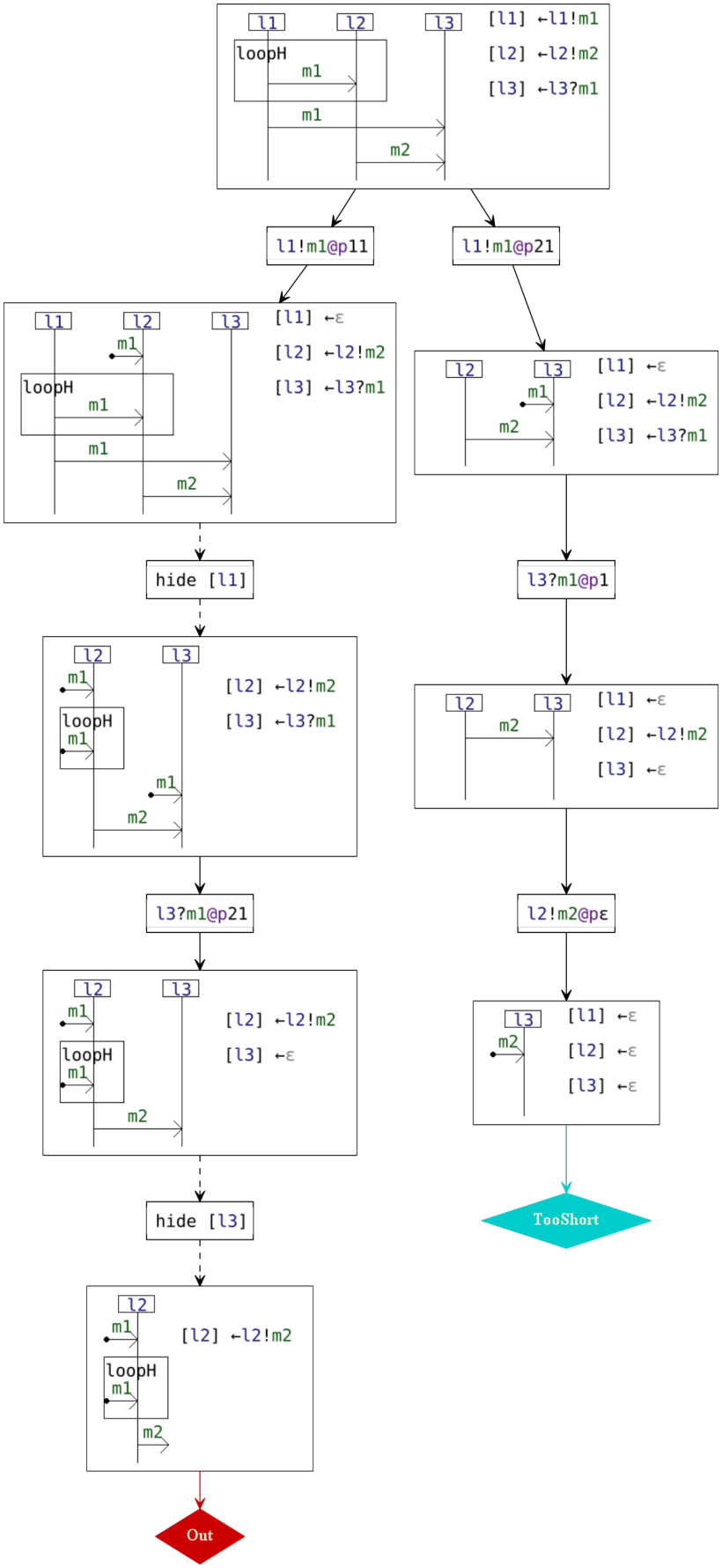


Figure 10.5: Analysis without checking local frontiers

10.3.1 Definition

For any co-localization c, the local frontier of an interaction i up to c is the frontier of $\mathbf{hide}_{L\setminus c}(i)$ where we have hidden all lifelines not in c. We thus define the "locfrt" function in Def.10.6.

Definition 10.6: Local frontiers

$\texttt{locfrt} : \mathbb{I}_\Omega \times \mathcal{P}(L) \to \mathcal{P}(\{1,2\}^*)$ is the function s.t. for any $i \in \mathbb{I}_\Omega$ and $c \in \mathcal{P}(L)$:

- $\texttt{locfrt}(i,c) = \texttt{frt}(\mathbf{hide}_{L\setminus c}(i))$

Let us then consider the example on Fig.10.6. On the left of Fig.10.6 is represented an interaction i defined over 4 lifelines. Let us suppose that $c = \{l_2, l_3\}$ constitutes a co-localization. On the right of Fig.10.6 is then represented $\mathbf{hide}_{L\setminus c}(i)$ i.e. the interaction i from which we have hidden l_1 and l_4 which are not in the co-localization c. We can then compute the frontier of $\mathbf{hide}_{L\setminus c}(i)$ in which there are two actions $l_3?m_1$ and $l_3?m_2$. The local frontier of i w.r.t. co-localization c is therefore composed of those two actions, which we have also illustrated on the left of Fig.10.6.

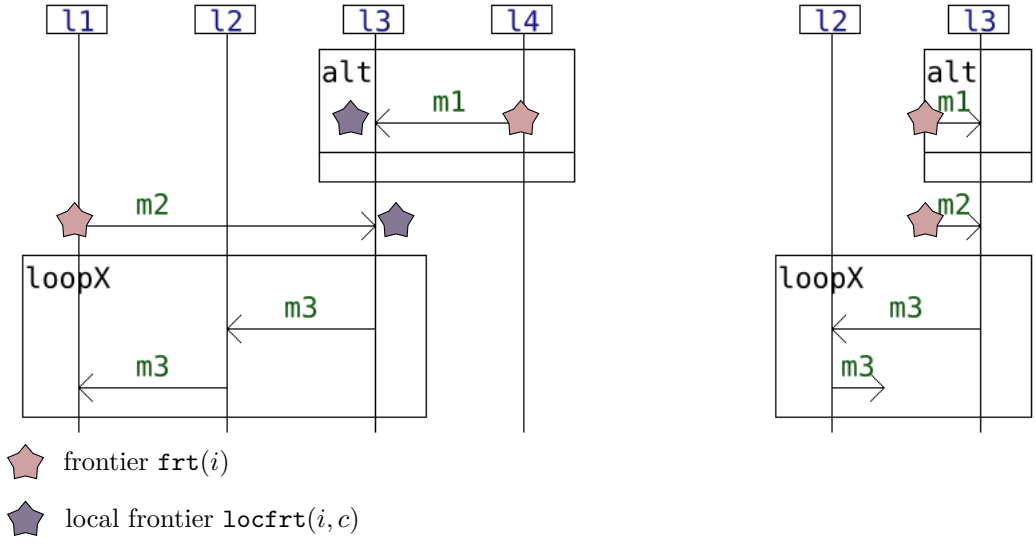


Figure 10.6: Example showcasing the local frontier on $c = \{l_2, l_3\}$

We can therefore see that local-frontier and frontier are two distinct notions. In the example from Fig.10.6 we can indeed see that $\texttt{frt}(i)$ and $\texttt{locfrt}(i,c)$ do not even intersect. Indeed, aside from its definition, `locfrt` is not necessarily related to `frt`. A characterization which holds however, is that given in Lem.10.5.

Lemma 10.5: A characterization of local frontiers

For any $i \in \mathbb{I}_\Omega$ and any $c \in \mathcal{P}(L)$:

$$\{p \in \texttt{frt}(i) \mid \theta(i_{|p}) \in c\} \subset \texttt{locfrt}(i,c)$$

Proof. It suffices to reason by induction on the structure of interaction terms and use the facts that $\texttt{locfrt}(i,c) = \texttt{frt}(\mathbf{hide}_{L\setminus c}(i))$ and that $\mathbf{hide}_{L\setminus c}$ preserves the structure of terms. □

10.3.2 Characterization w.r.t. the multi-trace semantics

As mentioned earlier, and as illustrated in Fig.10.6, a local frontier gathers actions which can be executed at the soonest on a certain co-localization. In the example from Fig.10.6 it is indeed not possible to have any other action than either $l_3?m_1$ or $l_3?m_2$ being the first action to be executed on $c = \{l_2, l_3\}$.

We can formalize this property of local frontiers in terms of accepted multi-traces. Whenever a multi-trace μ is accepted by an interaction i, then all its head actions i.e. all the actions a such that $\mu = a \overrightarrow{\odot}_C \mu'$, must be the first action which is executed on the co-localization $\theta_C(a)$ on which it occurs. As a result, we must have the existence of $p \in \texttt{locfrt}(i, \theta_C(a))$ such that $a = i_{|p}$. In Lem.10.6, we state this property for multi-traces that belong to $@_C(i)$, but, because $\sigma_{|C}(i) \subset @_C(i)$ as per Lem.7.9, this also holds true for $\mu \in \sigma_{|C}(i)$.

Lemma 10.6: Local frontiers and multi-trace semantics

For any $C \in \text{Part}(L)$, any $i \in \mathbb{I}_\Omega$, any $a \in \mathbb{A}_\Omega$ and any $\mu' \in \mathbb{T}_{\Omega|C}$:

$$(a \overrightarrow{\odot}_C \mu' \in @_C(i)) \Rightarrow (\exists p \in \texttt{locfrt}(i, \theta_C(a)),\ \text{s.t. } i_{|p} = a)$$

Proof. Let us then reason by induction on $|C|$:

- If $|C| = 1$ then we are on the trivial partition. Let us then use notations and results pertaining to the global trace semantics σ. We have $a.t' \in \sigma(i)$, which implies that there must exist i' s.t. $i \xrightarrow{a} i'$ and hence as per the equivalence of the execution and operational semantics, there must exist $p \in \texttt{frt}(i)$ such that $i_{|p} = a$ (and $\texttt{exe}(i, p) = i'$). Then, we can conclude using Lem.10.5.
- If $|C| > 1$, then we have at least two co-localizations. Let us then consider $c = \theta_C(a)$ the co-localization on which a occurrs and $h \in C \setminus \{c\}$. The fact that $a \overrightarrow{\odot}_C \mu' \in @_C(i)$ and $\theta(a) \notin h$ then implies (as per Fig.10.3) that $a \overrightarrow{\odot}'_C \mathbf{elim}_h(\mu') \in @_{C'}(\mathbf{hide}_h(i))$ with $C' = C \setminus \{h\}$. Given $|C'| < |C|$ we can apply the induction hypothesis so that we obtain $p \in \texttt{locfrt}(\mathbf{hide}_h(i), \theta_C(a))$ such that $(\mathbf{hide}_h(i))_{|p} = a$. This then implies that $p \in \texttt{locfrt}(i, \theta_C(a))$ with $i_{|p} = a$ because $h \neq \theta_C(a)$ and hence we can conclude

□

As a direct consequence of Lem.10.6, whenever a multi-trace starts with an action a, if this action is not in the local frontier of i w.r.t. co-localization $\theta_C(a)$, then it cannot be an accepted multi-trace. This theorem Th.10.3 motivates the checking of local frontiers whenever we analyze multi-traces.

Theorem 10.3: Checking `locfrt` as a sufficient condition to stop analyses

For any $C \in \text{Part}(L)$, any $i \in \mathbb{I}_\Omega$, any $a \in \mathbb{A}_\Omega$ and any $\mu' \in \mathbb{T}_{\Omega|C}$:

$$(\nexists p \in \texttt{locfrt}(i, \theta_C(a)),\ \text{s.t. } i_{|p} = a) \Rightarrow (a \overrightarrow{\odot}_C \mu' \notin \sigma_{|C}(i))$$

Proof. This in fact the contraposition of Lem.10.6 when applied to a multi-trace from $\sigma_{|C}(i)$.
It suffices to remark, as per Lem.7.9 that $\sigma_{|C}(i) \subset @_C(i)$. □

We then define a predicate for "checking local frontiers" in Def.10.7. This predicate can then be used in the formulation of multi-trace analysis algorithms.

Definition 10.7: Checking of local frontiers

For any $C \in \text{Part}(L)$, $\texttt{checkloc}_C : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{\top, \bot\}$ is s.t. for any $i \in \mathbb{I}_\Omega$ and $\mu \in \mathbb{T}_{\Omega|C}$:

$$\texttt{checkloc}_C(i,\mu) \Leftrightarrow (\forall\ c \in C, \mu_{|c} = a.t \Rightarrow \exists\ p \in \texttt{locfrt}(i,c),\ i_{|p} = a)$$

Checking the local frontiers is not however a replacement for checking the frontier. Indeed, we must still have a rule in the algorithm for indicating failure to consume the multi-trace even in the case where $\texttt{checkloc}_C(i,\mu) = \top$. This can be illustrated by the example from Fig.10.7 in which we have $\texttt{checkloc}_C(i,\mu) = \top$ because we indeed have that $l_1?m_2$ is in the local frontier w.r.t. l_1 and $l_2?m_1$ is in the local frontier w.r.t. l_2. However, from that vertex (i,μ), it is not possible to consume any action. And the multi-trace is indeed not accepted and not even a prefix of an accepted multi-trace.

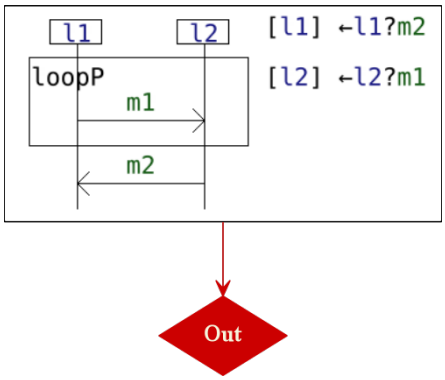


Figure 10.7: Example where checking the local frontiers does not replace checking the frontier

10.3.3 Application to improve multi-trace analysis

Following on the last example analysis from Fig.10.5, we introduce in Fig.10.8 the same analysis but with an algorithm which includes the checking of local frontiers. We can see that, by contrast to the algorithm from Fig.10.5 which explored the branch on the left until the *Out* verdict, the new algorithm, which checks local frontier, stops as soon as the first vertex of the left branch.

Indeed, in that vertex we have that $\mu_{|\{l_2\}} = l_2!m_2$ which would imply, if the multi-trace was acceptable, that $l_2!m_2$ would be in the local frontier w.r.t. $\{l_2\}$. However, we only have $l_2?m_1$ in that local frontier. Therefore the vertex does not satisfies the condition related to Th.10.3. Consequently, we can stop the analysis of that branch at this point because we know that is cannot yield to a positive result. Hence we decorate this path with a *Dead* local verdict on Fig.10.8.

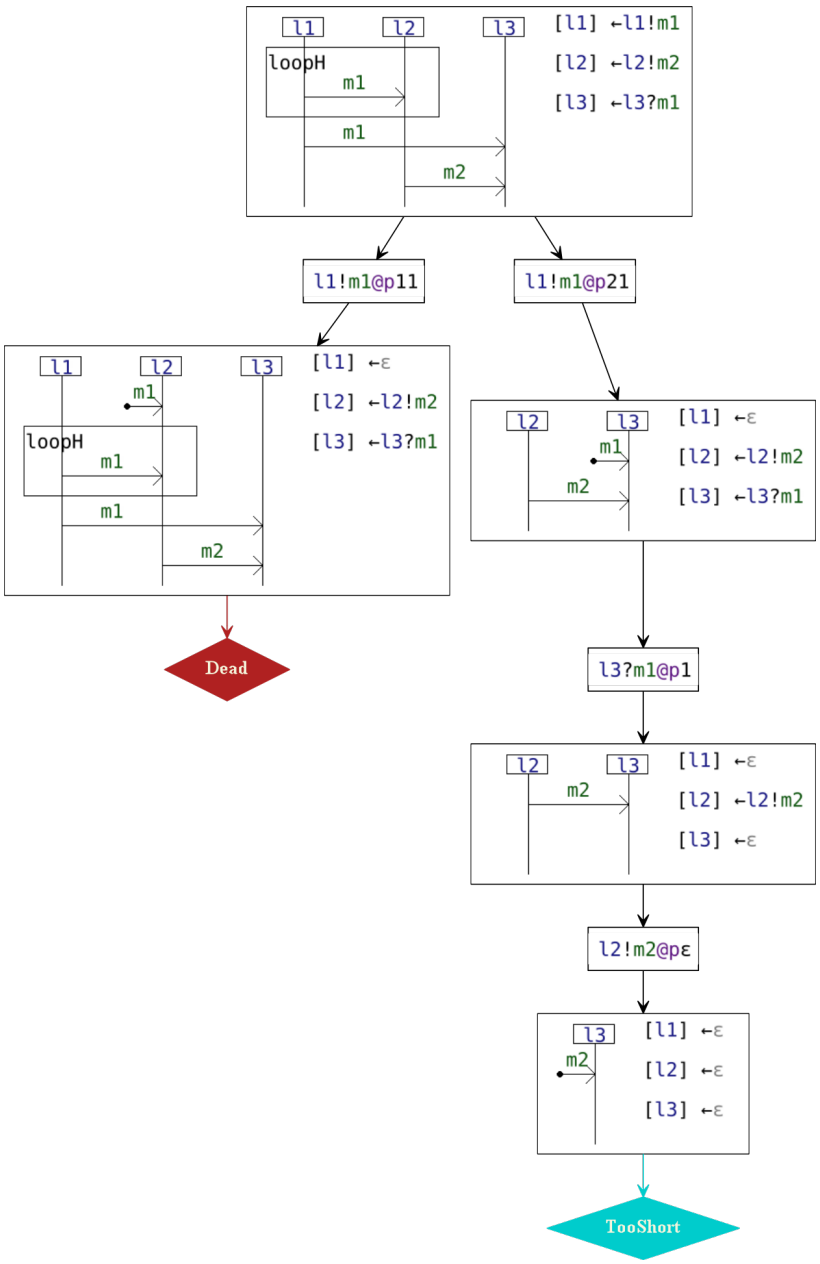


Figure 10.8: Analysis with the checking of local frontiers

The checking of local frontiers can be used to improve the efficiency of any multi-trace analysis algorithm of the type which we have described. We can for instance use it to improve ω_C from Chap.9 or $\overline{\omega}_L$ from Sec.10.2. In Chap.8 we have discussed algorithms based on simulation steps. In particular, one can use simulation steps as a preamble to the consumption of actions from a certain co-localization. The checking of local frontiers can be adapted to be compatible with those algorithms as well, as we will see in the next chapter.

Conclusion

In this chapter we have presented the notions of hiding interactions and eliminating co-localizations from multi-traces. We have shown the duality of those two notions with regards to the algebraic multi-trace

semantics. We then defined two practical uses of those operators. The first one is an algorithm which makes use of hiding steps to recognize prefixes (in the senses of multi-traces) of accepted multi-traces defined up to the discrete partition. The second one is a technique to improve the efficiency of any analysis algorithm. This technique allows one to avoid exploring some paths of the analysis graph that cannot yield a positive result.

Part III

Extensions & tools

Chapter 11

Immediate extensions

Contents

In this third and final part of the thesis, we discuss extending and implementing the proven theoretical results presented in the first two parts. We notably introduce two Formal Verification tools that were implemented as part of this thesis.

In this chapter we present some additional unproven (by contrast to all that was defined beforehand) theoretical results and discuss the ease of implementation of tools derived from the results of this thesis.

The plan of this chapter is as follows:

- in Sec.11.1 we present multi-trace analysis algorithms based on the use of simulation steps,
- in Sec.11.2 we present an additional scheduling operator, which, although it can be considered to be some form of syntactic sugar, is often found in informal Interaction Languages and proves quite useful in practice,
- finally, in Sec.11.3, we discuss the implementation of theoretical results.

11.1 Algorithms with simulation steps

In Chap.9 we defined an algorithm for recognizing accepted multi-traces and in Chap.10 another for identifying prefixes (in the sense of multi-traces) of accepted multi-traces defined up to the discrete partition. However, in Chap.7, we have defined a wide variety of different semantics and we have not yet presented algorithms to recognize all of those. In Chap.8 we have informally discussed the use of simulation steps as a useful mechanism to define such algorithms. In the beginning of Chap.10, we have dismissed those algorithms based on simulation steps due to their potential high overhead costs. However, with techniques such as the checking of local frontiers (introduced at the end of Chap.10) and some heuristics, simulating unobserved behavior seems once again practicable.

In this section we expand upon the results from the previous chapters by adapting the multi-trace analysis algorithm from Chap.9 for the analysis of multi-traces that are collected under contexts of degraded observability. In particular we define algorithms which use simulation steps so as to recognize prefixes and slices of accepted multi-traces defined up to any partition of lifelines.

The plan of this section is as follows:

- in Sec.11.1.1 we discuss the issue of the termination of analysis algorithms that rely on simulation steps,
- in Sec.11.1.2 we define an analysis algorithm which may use simulation steps for identifying prefixes of accepted multi-traces i.e. elements of $\overline{\sigma_{|C}}$ defined up to any partition C,
- in Sec.11.1.3 we present a generalized algorithm that allows one to identify and discriminate between exactly accepted multi-traces and the three different kinds of partially observed multi-traces which we have described in Chap.7.

11.1.1 The issue of termination & loops

In Chap.8, we explained that simulation steps could be used to fill-in gaps in a partially observed multi-traces in which events that have not been observed are missing. However, let us then consider this very trivial example: given $L = \{l\}$ and $M = \{m_1, m_2\}$, let us suppose that we have $i = loop_S(l!m_1)$ and we want to analyze $\mu = (l!m_2)$. Then, a very naive algorithm with simulation steps might hope that by simulating the execution of $l!m_1$ so that $i \xrightarrow{l!m_1} i$ it might somehow unlock the further consumption of $l!m_2$. Moreover, if there are no mechanisms to limit simulation steps, this algorithm might even never terminate.

This issue of termination ultimately relates to the presence of loops in the interaction term. Indeed, if there are no loops, then there can only be a finite number of simulation steps, until all the actions that are present in the initial term have been executed either via the usual execution and consumption of the action from the multi-trace or via some simulation step.

Hence, so as to ensure termination of the process, one must fix a limit to the number of times loops may be instantiated. However, this limit might not be global i.e. the same during the whole of the analysis. Indeed, a global behavior to which a multi-trace projects might be interspersed with missing events (i.e. gaps in observation). As a result, we might need to alternate between steps of consumption and steps of simulation. We argue for resetting this limit each time the normal consumption of the multi-trace is resumed.

Consequently, the limit on the instanciation of loops corresponds to the number of times loops may be instantiated during any given sub-path of $\mathbb{G}$ consisting of consecutive applications of simulation steps. We can fix the limit as an initial value to a certain index. Then, each time a loop is instantiated in a simulation step, this index is decremented of a certain value. Once the simulation has unlocked the further consumption of the multi-trace, and once an event has effectively been consumed from the multi-trace, this index can be reset. Details on this will be presented in Sec.11.1.2 and Sec.11.1.3 in which we concretely define algorithms which make use of simulation steps. In any case, this method allows to ensure the termination of the process. Indeed there can only be a finite number of "reset" and hence only a finite number of instanciations of loops and hence only a finite number of actions that can be executed either via consumption or simulation.

The limit that we fix i.e. the initial value of the index is set at the maximum loop depth of the current interaction term. We formalize this notion of "loop depth" $\beta_{\circlearrowleft}$ and "maximum loop depth" $\widehat{\beta}_{\circlearrowleft}$ in Def.11.1. This limit $\widehat{\beta}_{\circlearrowleft}(i)$ corresponds to the maximum depth of nested loops that can be found in the interaction term i.

Definition 11.1: Loop depth and maximum loop depth

We define $\beta_{\circlearrowleft} : \mathbb{I}_\Omega \times \{1,2\}^* \to \mathbb{N}$ such that for any interaction $i \in \mathbb{I}_\Omega$ and any position $p \in pos(i)$:

- if $p = \epsilon$ then $\beta_{\circlearrowleft}(i, \epsilon) = 0$
- if $p = 1.p_1$ then:
  - if $i = loop_f(i_1)$ then $\beta_{\circlearrowleft}(i, p) = 1 + \beta_{\circlearrowleft}(i_1, p_1)$
  - else then $i = f(i_1, i_2)$ and $\beta_{\circlearrowleft}(i, p) = \beta_{\circlearrowleft}(i_1, p_1)$
- if $p = 2.p_2$ then $i = f(i_1, i_2)$ and $\beta_{\circlearrowleft}(i, p) = \beta_{\circlearrowleft}(i_2, p_2)$

We can then define $\widehat{\beta}_{\circlearrowleft} : \mathbb{I}_\Omega \to \mathbb{N}$ such that for any interaction i:

$$\widehat{\beta}_{\circlearrowleft}(i) = max_{p \in pos(i)}\ \beta_{\circlearrowleft}(i, p)$$

This notion of maximum loop depth can be explained on some examples. For instance:

$$\begin{aligned}
\widehat{\beta}_{\circlearrowleft}(l_1!m_1) &= 0 \\
\widehat{\beta}_{\circlearrowleft}(alt(l_1!m_1, l_1!m_2)) &= 0 \\
\widehat{\beta}_{\circlearrowleft}(loop_S(l_1!m_1)) &= 1 \\
\widehat{\beta}_{\circlearrowleft}(loop_S(seq(l_1!m_1, loop_H(l_1!m_2)))) &= 2
\end{aligned}$$

11.1.2 Identifying prefixes of accepted multi-traces with simulation

In Chap.9, we have presented an analysis algorithm which allows for the identification of accepted multi-traces. This algorithm can, for any given interaction $i \in \mathbb{I}_\Omega$, and any multi-trace $\mu \in \mathbb{T}_{\Omega|C}$ determine whether or not $\mu \in \sigma_{|C}(i)$.

In this section, we present a modified version of that previous algorithm, that instead allows one to determine whether or not $\mu \in \overline{\sigma_{|C}}(i)$. This means that this algorithm allows one to identify prefixes (in the sens of multi-traces) of accepted multi-traces. At the same time we will introduce the use of the checking of local frontier from Chap.10.

By analogy with the results from Chap.9, we introduce the analysis algorithm via the definition of a directed graph $\mathbb{G}$ in which vertices are either a tuple $(i, \mu, j) \in \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \times \mathbb{N}$ or a local verdict $v \in \{Obs, Out\}$. We then note $\mathbb{V} = \{Obs, Out\} \cup (\mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \times \mathbb{N})$ the set of all vertices.

The algorithm then relies on the exploration of some paths within $\mathbb{G}$ that start from a specific node and in which edges are characterized by the relation $\rightsquigarrow$ defined in Def.11.2 via the formulation of four rules.

For analysing a multi-trace μ_0 against an interaction i_0, the analysis starts from the node $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0))$. Here $\widehat{\beta}_{\circlearrowleft}(i_0)$ acts as the maximum number of loops that can be instantiated in a consecutive succession of simulation steps.

From any node of the form (i, μ, j), we can apply one of the five rules (and as a result, (i, μ, j) nodes are not sinks):

- if $\mu = \epsilon_C$ i.e. if it is empty, then we can apply R_p and reach the *Obs* local verdict

- if μ is not empty then:

  - if the checking of local frontiers fails then we can apply R_d and reach the *Out* local verdict

  - if not (i.e. if $\texttt{checkloc}_C(i, \mu) = \top$) then:

    * if there is a match between an immediately executable action from the interaction i and the head of a component of μ then we can apply R_e. With this rule, we simultaneously execute the matching action in the interaction term and consume it in the multi-trace. Also, as explained before, we reset the value of the index j to the maximum loop depth of the new follow-up interaction.

    * if there exists an empty trace component, let us then consider $c \in C$ such that $\mu_{|c} = \epsilon$. Then we may allow the simulation of the behavior of lifelines $l \in c$. This means that we may allow the execution of frontier actions $i_{|p}$ with $p \in \texttt{frt}(i)$ if $\theta_C(i_{|p}) = c$. However, as explained before, we restrict which actions can be executed depending on whether or not their execution involves the instantiation of loops. This constraint corresponds to the $j \geq \beta_{\circlearrowleft}(i, p)$ condition, and allows the process to terminate. Those simulation steps therefore correspond to the application of rule R_s. Given the execution of a frontier action $i \xrightarrow{a@p} i'$, if the previous constraints are satisfied then R_s yields the node $(i', \mu, (j - \beta_{\circlearrowleft}(i, p)))$, with the index j being decremented by the loop-depth of the executed action.

    * if no action can be executed either via consumption or simulation then we can apply rule R_f and reach the *Out* local verdict.

As in case of the previous algorithms, the only two sinks of the graph $\mathbb{G}$ are the local verdicts *Obs* and *Out*. We define the analysis relation, which defines edges of graph $\mathbb{G}$ in Def.11.2.

Definition 11.2: Rules for Multi-Prefix Analysis with simulation

The analysis relation $\rightsquigarrow \subseteq \mathbb{V} \times \mathbb{V}$ is defined as:

$$(R_p)\ \frac{i \quad \epsilon_C \quad j}{Obs} \qquad (R_d)\ \frac{i \quad \mu \quad j}{Out}\ \texttt{checkloc}_C(i,\mu) = \bot$$

$$(R_e)\ \frac{i \quad a \overrightarrow{\odot} \mu \quad j}{i' \quad \mu \quad \widehat{\beta}_{\circlearrowleft}(i')} \begin{cases} (\texttt{checkloc}_C(i, a \overrightarrow{\odot} \mu) = \top) \\ \wedge\ (i \xrightarrow{a} i') \end{cases}$$

$$(R_s)\ \frac{i \quad \mu \quad j}{i' \quad \mu \quad (j - \beta_{\circlearrowleft}(i,p))} \begin{cases} (\texttt{checkloc}_C(i,\mu) = \top) \\ \wedge\ (i \xrightarrow{a@p} i') \wedge\ (j \geq \beta_{\circlearrowleft}(i,p)) \\ \wedge\ (\mu \neq \epsilon_C)\ \wedge\ (\mu_{|\theta_C(a)} = \epsilon) \end{cases}$$

$$(R_f)\ \frac{i \quad \mu \quad j}{Out} \begin{cases} (\texttt{checkloc}_C(i,\mu) = \top) \wedge (\mu \neq \epsilon_C) \\ \wedge \left(\nexists i \xrightarrow{a@p} i' \text{ s.t.} \left(\begin{array}{l} (\mu = a \overrightarrow{\odot}_C \mu') \\ \vee\ \left((\mu_{\theta_C(a)} = \epsilon) \wedge (j \geq \beta_{\circlearrowleft}(i,p)) \right) \end{array} \right) \right) \end{cases}$$

Similarly as what we did in Chap.9, we can then remark that the sub-graph of $\mathbb{G}$ (the analysis graph) that is thus defined (starting from $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0))$) is finite and:

- there exists at least one path from $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0))$ to either Obs or Out
- either Obs or Out are reachable from any node of the form (i, μ, j)

As a consequence, we can define a function $\overline{\omega_C} : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{Pass, Fail\}$ which returns the global verdict resulting from the analysis of a multi-trace μ_0 against an interaction i_0. The definition of this $\overline{\omega_C}$ is reminiscent of that of ω_C from Chap.9 and is given in Def.11.3.

Definition 11.3: Multi-Trace Prefix Analysis with simulation

$\overline{\omega_C} : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{Pass, Fail\}$ is s.t. for any $i \in \mathbb{I}_\Omega$ and $\mu \in \mathbb{T}_{\Omega|C}$ we have:

- $\overline{\omega_C}(i,\mu) = Pass$ iff there exists a path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i)) \overset{*}{\rightsquigarrow} Obs$
- $\overline{\omega_C}(i,\mu) = Fail$ otherwise;
  i.e. for all path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i)) \overset{*}{\rightsquigarrow} v$ with $v \in \{Obs, Out\}$, then $v = Out$

The correctness of algorithm ω_C from Chap.9 has been proven in Th.9.1 and Th.9.2 and equates having, for any interaction i and any multi-trace μ:

$$\left(\ \omega_C(i,\mu) = Pass \ \right) \Leftrightarrow \left(\ \mu \in \sigma_{|C}(i) \ \right)$$

By analogy the correctness of this new algorithm $\overline{\omega_C}$ equates to having, for any interaction i and any

multi-trace μ:

$$\left(\ \overline{\omega_C}(i,\mu) = Pass \ \right) \Leftrightarrow \left(\ \mu \in \overline{\sigma_{|C}}(i) \ \right)$$

Writing a formal proof of this predicate would require the formulation of a property stating that fixing the maximum number of loop instantiation at $\widehat{\beta}_{\circlearrowleft}(i)$ is sufficient to explore all paths that would allow the further consumption of μ. We have not written such a proof. However, extensive use of this algorithm via the HIBOU tool implementation (that we will present in Chap.12) seems to confirm that this predicate holds. Moreover, we have defined in Chap.10 the $\overline{\circledcirc_L}$ algorithm which is proven correct. Given that both $\overline{\circledcirc_L}$ and $\overline{\omega_L}$ are implemented in HIBOU, we can compare their outputs.

11.1.3 Algorithm to discriminate elements of different semantics

In this chapter we present an algorithm that allows to discriminate between multi-traces belonging to any or none of the four semantics $\sigma_{|C}$, $\sigma^{\dagger}_{|C}$, $\overline{\sigma_{|C}}$ and $\underline{\overline{\sigma_{|C}}}$.

This algorithm is defined in the same manner as the previous. It relies on defining and exploring a directed graph $\mathbb{G}$ in which vertices are:

- either a tuple (i, μ, j, s, O) where:
  - $i \in \mathbb{I}_{\Omega}$ is the specifying interaction
  - $\mu \in \mathbb{T}_{\Omega|C}$ is the multi-trace to analyse
  - j is an index to count the number of loops that one is still allowed to instantiate in successive simulation steps (as in the previous algorithm)
  - $s \in \mathcal{P}(\{s_{pre}, s_{post}\})$ is a set that keeps track of whether or not we have already done some simulation steps, either as a preambule or as a postambule to the consumption of any given trace component
  - $O \in \mathcal{P}(C)$ is a set that keeps track of the co-localization of which the corresponding trace component we have already started to consume during the analysis
- or a local verdict $v \in \{Cov, Short, MultiPref, Slice, Out\}$

As was previously done, we can denote by $\mathbb{V}$ the set of such vertices and we then define the algorithm via a relation $\rightsquigarrow$ on $\mathbb{V}$.

The analysis of the multi-trace μ_0 against the interaction i_0 then corresponds to the exploration of a sub-graph of $\mathbb{G}$ that starts from the node $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0), \emptyset, \emptyset)$.

We adapt the checking of local frontiers to the fact that we may allow simulation steps to be taken as a preamble to the consumption of actions. It is the set $O \in \mathcal{P}(C)$ which keeps track of which trace components have been consumed. As a result, we may check the local frontiers of the components that are in O but not of the components that are in $C \setminus O$.

Definition 11.4: Adapted checking of local frontiers

$\texttt{checkloc}_C : \mathbb{I}_\Omega \times \mathcal{P}(C) \times \mathbb{T}_{\Omega|C} \to \{\top, \bot\}$ is s.t. for any $i \in \mathbb{I}_\Omega$, any $O \in \mathcal{P}(C)$ and any $\mu \in \mathbb{T}_{\Omega|C}$:

$$\texttt{checkloc}_C(i, O, \mu) \Leftrightarrow (\forall\ c \in O, \mu_{|c} = a.t \Rightarrow \exists\ p \in \texttt{locfrt}(i, c),\ i_{|p} = a)$$

From any node of the form (i, μ, j, s, O) we can apply one of nine rules:

- if $\mu = \epsilon_C$ then:
  - if we have not made any simulation steps (i.e. if $s = \emptyset$) then:
    - if i can express the empty trace i.e. if $i \downarrow$ then we can apply R_{p1} and reach the Cov local verdict
    - if i cannot express the empty trace i.e. if $i \not\downarrow$ then we apply R_{p2} and reach the $Short$ local verdict
  - if we have made some simulation steps only as postambles to the consumption of some trace components i.e. if $s = \{s_{post}\}$ then we apply R_{p3} and reach the $MultiPref$ local verdict
  - if we have made some simulation steps including some as preambles to the consumption of some trace components i.e. if $s_{pre} \in s$ then we apply R_{p4} and reach the $Slice$ local verdict
- if μ is not empty then:
  - if $\texttt{checkloc}_C(i, O, \mu) = \bot$ then we can apply R_d and reach the Out local verdict
  - if $\texttt{checkloc}_C(i, O, \mu) = \top$ then:
    - if there is a match between a frontier action and the head of a trace component then we can apply the R_e rule. This rule, as in the previous algorithm, consists in removing the head action from the multi-trace, executing its match in the interaction, and resetting the counter j. However, we additionally keep track of the co-localization on which we have consumed the action by adding it to the set O. Adding $\theta_C(a)$ into O will then prevent any further simulation steps as preambule to the analysis of the behavior of co-localization $\theta_C(a)$.
    - if there is no match then we may proceed to two kinds of simulation steps to try to unlock the further consumption of the multi-trace. For any frontier action p:
      - if the trace component $\mu_{|\theta_C(i_{|p})} = \epsilon$ i.e. if there is nothing left to observe on the co-localization on which $i_{|p}$ occurs, then, as in the previous algorithm, we may simulate the behavior of that co-localization as a postamble. As previously, we need to respect the constraint $j \geq \beta_{\circlearrowleft}(i, p)$ to do so. Additionally, when applying R_s^+, we update s as $s \cup \{s_{post}\}$ to keep track of the application of the simulation step.
      - if $\mu_{|\theta_C(i_{|p})} \neq \epsilon$ and if the consumption of events has not yet started on the co-localization on which $i_{|p}$ occurs i.e. if $\theta_C(i_{|p} \notin O$ then we may execute $i_{|p}$ as a preamble simulation step

if $j \geq \beta_{\circlearrowleft}(i,p)$. Doing so may serve as replacing an event which has not been observed due to a late start of the observation on $\theta_C(i_{|p})$. In the new node, following the application of rule R_s^-, we then update s with $s \cup \{s_{pre}\}$ given that we have made a preamble simulation step.

- if no action can be executed either via consumption or simulation then we can apply rule R_f and reach the Out local verdict

Definition 11.5: Rules for discriminating multi-traces from the various semantics

The analysis relation $\rightsquigarrow \subseteq \mathbb{V} \times \mathbb{V}$ is defined as:

$$(R_{p1})\ \frac{i \quad \epsilon_C \quad j \quad s \quad O}{Cov}\ (i\downarrow) \wedge (s = \emptyset)$$

$$(R_{p2})\ \frac{i \quad \epsilon_C \quad j \quad s \quad O}{Short}\ (i\not\downarrow) \wedge (s = \emptyset)$$

$$(R_{p3})\ \frac{i \quad \epsilon_C \quad j \quad s \quad O}{MultiPref}\ (s_{post} \in s) \wedge (s_{pre} \notin s)$$

$$(R_{p4})\ \frac{i \quad \epsilon_C \quad j \quad s \quad O}{Slice}\ (s_{pre} \in s)$$

$$(R_d)\ \frac{i \quad \mu \quad j \quad s \quad O}{Out}\ \texttt{checkloc}_C(i, O, \mu) = \bot$$

$$(R_e)\ \frac{i \quad a \vec{\odot} \mu \quad j \quad s \quad O}{i' \quad \mu \quad \widehat{\beta}_{\circlearrowleft}(i') \quad s \quad (O \cup \{\theta_C(a)\})}\ \begin{cases} (\texttt{checkloc}_C(i, O, \mu) = \top) \\ \wedge\ (i \xrightarrow{a} i') \end{cases}$$

$$(R_s^-)\ \frac{i \quad \mu \quad j \quad s \quad O}{i' \quad \mu \quad (j - \beta_{\circlearrowleft}(i,p)) \quad (s \cup \{s_{pre}\}) \quad O}\ \begin{cases} (\texttt{checkloc}_C(i, O, \mu) = \top) \\ (i \xrightarrow{a@p} i') \wedge\ (j \geq \beta_{\circlearrowleft}(i,p)) \\ \wedge\ (\mu \neq \epsilon_C)\ \wedge\ (\mu_{|\theta_C(a)} \neq \epsilon) \\ \wedge\ (\theta_C(a) \notin O) \end{cases}$$

$$(R_s^+)\ \frac{i \quad \mu \quad j \quad s \quad O}{i' \quad \mu \quad (j - \beta_{\circlearrowleft}(i,p)) \quad (s \cup \{s_{post}\}) \quad c}\ \begin{cases} (\texttt{checkloc}_C(i, O, \mu) = \top) \\ (i \xrightarrow{a@p} i') \wedge\ (j \geq \beta_{\circlearrowleft}(i,p)) \\ \wedge\ (\mu \neq \epsilon_C)\ \wedge\ (\mu_{|\theta_C(a)} = \epsilon) \end{cases}$$

$$(R_f)\ \frac{i \quad \mu \quad j \quad s \quad O}{Out}\ \begin{cases} (\texttt{checkloc}_C(i, \mu) = \top) \wedge (\mu \neq \epsilon_C) \\ \wedge \left(\nexists i \xrightarrow{a@p} i', \left(\begin{array}{l} (\mu = a \vec{\odot}_C \mu') \\ \vee \left((j \geq \beta_{\circlearrowleft}(i,p)) \wedge \left(\begin{array}{l} (\mu_{\theta_C(a)} = \epsilon) \\ \vee\ (\theta_C(a) \notin O) \end{array} \right) \right) \end{array} \right) \right) \end{cases}$$

Similarly to the previous algorithms, the analysis explores a sub-graph of $\mathbb{G}$ that starts from a node $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0), \emptyset, \emptyset)$. This sub-graph is finite and:

- there exists at least one path from $(i_0, \mu_0, \widehat{\beta}_{\circlearrowleft}(i_0), \emptyset, \emptyset)$ to one of either local verdicts
- a local verdict is reachable from any node of the form (i, μ, j, s, O)

As a consequence, we can define a function $\omega^*_C : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{Pass, WP_{short}, WP_{muprf}, WP_{slice}, Fail\}$ which returns the global verdict resulting from the analysis of a multi-trace μ_0 against an interaction i_0.

Definition 11.6: Multi-trace analysis with discrimination

For any signature $\Omega = (L, M)$ and any partition $C \in \text{Part}(L)$ of lifelines, we define $\omega^*_C : \mathbb{I}_\Omega \times \mathbb{T}_{\Omega|C} \to \{Pass, WP_{short}, WP_{muprf}, WP_{slice}, Fail\}$ such that for any $i \in \mathbb{I}_\Omega$ and $\mu \in \mathbb{T}_{\Omega|C}$ we have:

- $\omega^*_C(i, \mu) = Pass$ iff there exists a path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} Cov$
- $\omega^*_C(i, \mu) = WP_{short}$ iff there exists a path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} Short$
  and there are no paths $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} Cov$
- $\omega^*_C(i, \mu) = WP_{muprf}$ iff there exists a path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} MultiPref$
  and there are no paths $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} v$ for any $v \in \{Cov, Short\}$
- $\omega^*_C(i, \mu) = WP_{slice}$ iff there exists a path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} Slice$
  and there are no paths $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} v$ for any $v \in \{Cov, Short, MultiPref\}$
- $\omega^*_C(i, \mu) = Fail$ otherwise;
  i.e. for all path $(i, \mu, \widehat{\beta}_{\circlearrowleft}(i), \emptyset, \emptyset) \overset{*}{\rightsquigarrow} v$,
  if $v \in \{Cov, Short, MultiPref, Slice, Out\}$, then $v = Out$

In intention, this new algorithm characterizes the semantics of interactions and the various notions of prefixes as follows:

$$\left(\omega^*_C(i, \mu) = Pass \right) \Leftrightarrow \left(\mu \in \sigma_{|C}(i) \right)$$
$$\left(\omega^*_C(i, \mu) = WP_{short} \right) \Leftrightarrow \left(\mu \in \sigma^\dagger_{|C}(i) \setminus \sigma_{|C}(i) \right)$$
$$\left(\omega^*_C(i, \mu) = WP_{muprf} \right) \Leftrightarrow \left(\mu \in \overline{\sigma_{|C}}(i) \setminus \sigma^\dagger_{|C}(i) \right)$$
$$\left(\omega^*_C(i, \mu) = WP_{slice} \right) \Leftrightarrow \left(\mu \in \underline{\sigma_{|C}}(i) \setminus \overline{\sigma_{|C}}(i) \right)$$

This algorithm is implemented in the HIBOU tool, which we describe in Chap.12.

11.2 Co-regions

In addition to the three scheduling constructors *strict*, *seq* and *par*, we can consider a family of scheduling constructors called co-regions.Those constructors behave like *par* when considering some lifelines, and like *seq* when considering all others. As a result, only minor adaptations are needed so as to use the various results presented in this thesis on those constructors. A co-region must be configured by a subset of lifelines and there exists as many co-region operators as there exists subsets of the set of lifelines (i.e. $\mathcal{P}(L)$). We

use co-regions to schedule behaviors that must occur sequentially on some sub-systems but that can occur concurrently on some other sub-systems.

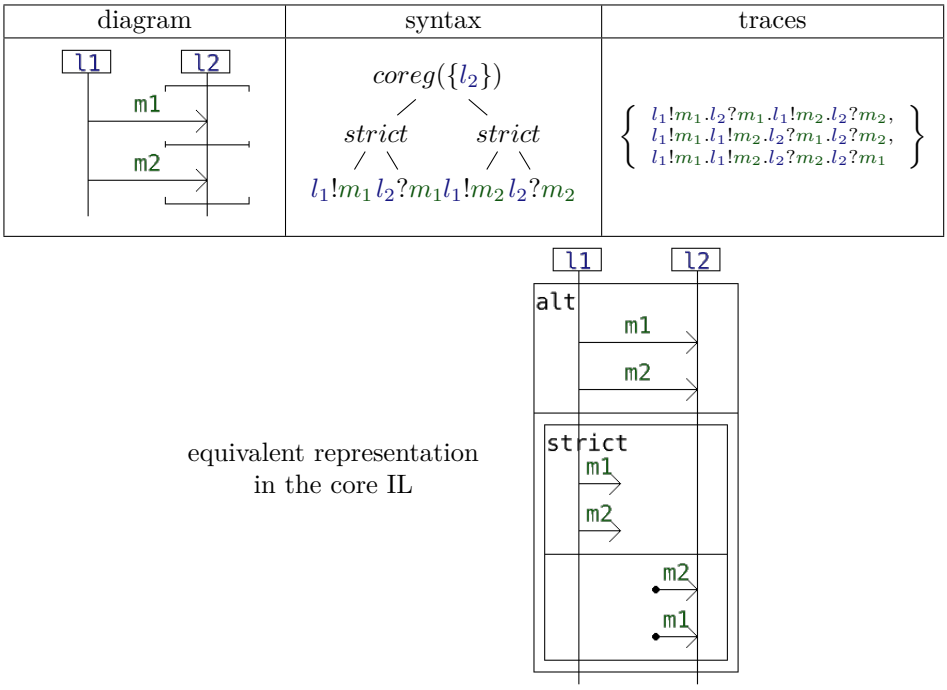


Figure 11.1: Example using a co-region constructor

The use of co-regions is illustrated on Fig.11.1. Here, the two sub-interactions $strict(l_1!m_1, l_2?m_1)$ and $strict(l_1!m_2, l_2?m_2)$ are scheduled using $coreg(\{l_2\})$, meaning that they are scheduled sequentially w.r.t. lifeline l_1 and concurrently w.r.t. lifeline l_2. As a result, the semantics of this interaction contains 3 accepted traces. Either the first message passing is completed before the second starts (first trace), or the message passings are interleaved (second and third traces). Given that the behaviors are scheduled sequentially w.r.t. lifeline l_1, this interleaving cannot concern the $l_1!m_1$ and $l_1!m_2$ actions, which keep the same relative ordering (i.e. $l_1!m_1$ before $l_1!m_2$). However, we can switch the order of $l_2?m_1$ and $l_2?m_2$.

The use of co-regions constitute a syntactic sugar. Indeed, as exemplified on Fig.11.1 it is always possible to express, using the core IL, an interaction that is semantically equivalent to an interaction that uses co-regions. However doing so is often impractical and inelegant.

Moreover, for modelling real-life DSs, the co-region constructs are particularly useful. Indeed, the non-determinism of the support networks through which messages transit makes so that messages send in a certain order (that is enforced on the emitter sub-system / lifeline) may not be received in the same order (order not enforced on the receiver sub-systems / lifelines).

Let us remark that for a co-region $coreg(x)$, with $x \in \mathcal{P}(L)$, if $x = \emptyset$ then $coreg(x)$ behaves for all intent and purposes, exactly like the *seq* operator; and if $x = L$ then $coreg(x)$ behaves for all intent and purposes, exactly like the *par* operator.

11.3 Towards an implementation

The manner in which we have defined all the theoretical tools from Chap.6, Chap.9, Chap.10 and Sec.11.1 related to the analysis of multi-traces allows for an immediate transcription into code.

We have indeed defined in Chap.6 the simplifying execution semantics $\sigma_{\underset{e}{\approx}}$ which relies on (1) the frontier function $\texttt{frt}$ to compute, for an input interaction i, all the positions of actions in i that are immediately executable, and (2) an execution function $\texttt{exe}_{\approx}$ which, for any interaction i and frontier position $p \in \texttt{frt}(i)$ computes a follow-up interaction which exactly specifies what remains to be executed after the execution of the action $a = i_{|p}$ at position p. This simplifying execution function offers the additional guarantee that the follow-up term is simplified so that we keep intermediate terms compact and simple.

We have also mentioned in Chap.6 that all those functions and all the predicates and functions on which they depend i.e. the termination "$\downarrow$" predicate, the evasion "$\downarrow^{\divideontimes}$" predicate, and the pruning function with simplifications $\texttt{prn}_{\approx}$ can be express in the style of functional programming languages.

Additionally, the types on which our theory is based i.e. that of interactions, positions and multi-traces can all be encoded using classic data types of functional programming languages. With for instance, actions being tuples of integers (with a 1-to-1 mapping to a set of lifeline names, message names, etc.) interactions being inductive types, positions being lists of integers and multi-trace lists of actions.

We can therefore easily implement the basis for animating i.e. executing and displaying interaction models. From this basis we can then implement features to (1) explore the semantics of interactions i.e. perform traversals of their execution trees and (2) analyze multi-traces against interactions i.e. perform traversals of a corresponding analysis graph (which nature depends on the kind of algorithm we choose to use among those defined in Chap.9, Chap.10 and Sec.11.1).

As is the case whenever traversals of trees or graphs are involved, we can use a queue and heuristics to guide the process of exploration. On Fig.11.2 we illustrate the use of such a configurable process for performing the analysis of a multi-trace μ against an interaction i using an algorithm similar to that of Chap.9. This tooled and configurable algorithm returns $Pass$ whenever $\mu \in \sigma_{|C}(i)$, WP (for "WeakPass") whenever $\mu \in \sigma^{\dagger}_{|C}(i)$ (without guaranteeing that $\mu \notin \sigma_{|C}(i)$) and $Fail$ otherwise. The two $\bullet$ correspond to the initialization, with the queue q being initialized as an empty queue, and the information of the initial vertex (i, μ) being provided as entry data.

Then, starting from (i, μ):

- we check if μ is the empty multi-trace (as indicated by $(\mu = \epsilon_C)$? on Fig.11.2) and:
  - if it is the case then we check if i terminates (as indicated by $(i \downarrow)$? on Fig.11.2) and:
    - if it is the case then the algorithm stops and returns the global verdict $Pass$. We can see that, by contrast to the more theoretical algorithms presented before, this consists in defining a stopping criterion on the exploration of the analysis graph. Indeed, we stop exploring the graph as soon as we find a path in which μ can be entirely consumed. In this case, the fact

that we have uncovered a path in the analysis graph in which we have consumed μ entirely and in which there remain i s.t. $i\downarrow$ constitutes a formal proof that $\mu \in \sigma_{|C}(i)$ and we do not need to explore the graph any further.

* in the other case, WP is returned, indicating that we have proven (by discovering a certain path in the analysis graph) that $\mu \in \sigma^{\dagger}_{|C}(i)$

– if it is not the case, then we try to simultaneously consume an action at the head of multi-trace and execute a matching frontier action of the interaction model, as in rule *R3* of the algorithm from Chap.9. In this "tooled" analysis algorithm, we compute a set of matches of the form (i, p, μ') such that p is a frontier action and $\mu = i_{|p} \overrightarrow{\odot}_C \mu'$. We then insert all those matches in our queue q, with heuristics being able to modulate how those matches are inserted. Given that initially, the queue was empty, in the first pass of the algorithm, this inserts into the queue all of the matches found from the initial vertex (i, μ) of the analysis graph.

As a result, this first pass of the algorithm either yields a global verdict or results in the insertion of some matches in the queue. Let us then describe what happens in the latter case.

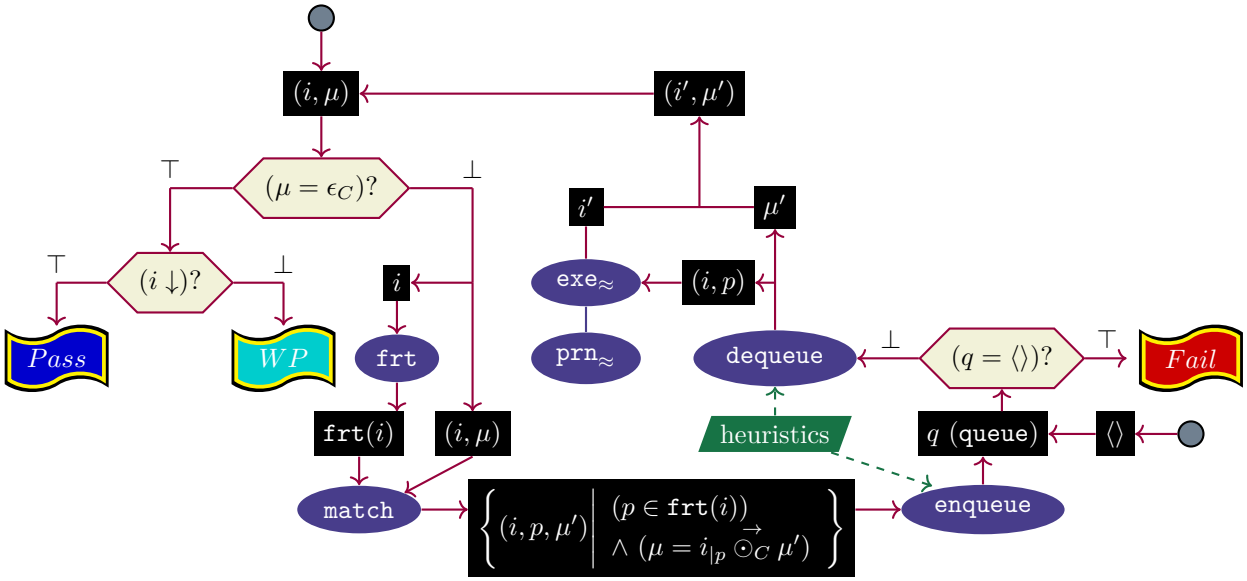


Figure 11.2: Principle of a configurable implementation of an analysis algorithms like that of Chap.9

We start by checking if the queue indeed has some elements in it because it could be so that no matches were found during the first pass. On Fig.11.2, this is indicated by $(q = \langle\rangle)?$. Then:

- if the queue is empty then the algorithm immediately stops and returns a global verdict $Fail$. In the case of the first pass of the algorithm, this is explained by the fact that we have found no matches allowing transitioning out of the initial vertex (i, μ). In the case of later passes of the algorithm, this means that no further exploitation of matches found globally can be performed. We have explored every reachable vertex of the form (i, μ) without being able to consume μ and hence the global verdict $Fail$ is justified.

- if the queue is not empty, then this means that there are some possible applications of rule $R3$ that would allow to reach some further vertices of the analysis graph that have not yet been visited. The decision of which element of the queue i.e. which match should be exploited first can be taken according to some heuristics. In any case, an element (i, p, μ') of the queue is dequeued. Then, we execute action $i_{|p}$ at position p in i i.e. we compute $i' = \texttt{exe}_{\approx}(i, p)$. At this point we have therefore visited a new vertex (i', μ') of the analysis graph and the process can be repeated.

With the example presented in this section, we have illustrated the ease of implementation permitted by the formulation of the theoretical results of the thesis. The definition of this type of process that is configurable via the use of heuristics can be applied to all previously defined multi-trace analysis algorithms.

Conclusion

In this chapter we have presented immediate extensions to the results of Part.I and Part.II. We have notably defined algorithms that make use of simulation steps so as to solve the membership problem for various multi-trace semantics. With those additional algorithms, we can now identify and discriminate between elements of all the projected multi-trace semantics which we have defined in Chap.7. The termination of those algorithms is guaranteed by a criterion on the number of loop instantiations during consecutive successions of simulation steps. We argue that a formal proof for the correctness of those algorithms is possible. It would rely on demonstrating that instantiating once each loop is sufficient so as to reveal all potential manners to further consume the multi-trace.

In this chapter we have also presented a small but practical extension to the language in the form of the *coreg* constructor which mixes weak sequencing and interleaving.

Finally, we have demonstrated, on an example analysis algorithm, the ease with which one can implement the various algorithms which we have previously defined. Such implementations can additionally be made configurable through the use of search heuristics for the traversal of analysis graphs. By setting a stopping criterion, the whole of the graph may not need to be visited.

In the next chapter we present the actual tool implementation HIBOU which implements almost everything that was presented in the previous chapters and including this one.

Chapter 12

The HIBOU tool

Contents

In this chapter we present the HIBOU tool which implements most of the theoretical results that were previously presented.

The plan of this chapter is the following:

- in Sec.12.1 we quickly present the software and relevant parts of its user interface,
- in Sec.12.2 we present the entry language for encoding interaction terms and multi-traces,
- in Sec.12.3 we present the feature for exploring the semantics (i.e. execution trees) of interactions and introduce possible configurations of heuristics for the traversal of execution trees and analysis graphs,
- in Sec.12.4 we present the feature for analysing multi-traces, which allows the use of any of the previously defined analysis algorithms.

12.1 Overview

HIBOU (for Holistic Interaction Behavioral Oracle Utility) provides utilities for drawing and manipulating interaction models, computing their normal forms, exploring their semantics and analysing executions of DSs (collected and transposed in the form of traces or multi-traces) with regards to formal specifications written as interaction models. This enables, among other things, the incremental design of precise and complex behavioral models of DSs.

We have implemented HIBOU using the Rust programming language [7]. The use of this language which is reliable, fast and memory efficient and in which it is difficult to make runtime errors allowed us to concentrate on the faithful implementation of the theoretical results presented in the first two parts of the thesis. The full code of HIBOU is hosted on github and available in [91]. It contains (for version 0.5.6) 11526 lines in 96 files among which 8373 lines of code. The version of HIBOU which we describe in this chapter is version 0.5.6.

HIBOU takes the form of an executable which offers a user Command Line Interface (CLI). Precompiled binaries are available for Windows and Linux in [91].

The use of HIBOU is associated to two kinds of input files:

- "hibou specification files", in which interaction models are encoded and which correspond to the ".hsf" filename extension
- "hibou trace file", in which traces or multi-traces are encoded and which correspond to the ".htf" filename extension

Then, the `hibou` executable provides a CLI which includes (among others) the following commands:

- "`hibou draw <.hsf file>`" draws an interaction in the manner which we have seen throughout the thesis

- "`hibou canonize [-s] <.hsf file>`" computes and displays the computation of the unique normal form (as per the results from Chap.4) of an interaction. By default, only one sequence of transformations is computed to reach the normal form. However, with the "`-s`" optional flag we can choose to compute and display all possible sequences of transformations (for example so as to make sure of the confluence of the implemented rewrite system).
- "`hibou explore` <.hsf file>" explores the semantics of an interaction i.e. it computes and may display parts of the execution tree of that interaction.
- "`hibou analyze` <.hsf file> <.htf file>" analyzes a multi-trace against an interaction i.e. it computes and may display parts of an analysis graph related to that analysis and returns a global verdict.

12.2 Entry language for interaction terms & multi-traces

In order to encode interaction terms and multi-traces (as well as other things such as the configuration of analysis algorithms, etc.), we have used a Parsing Expression Grammar [57]. Instead of detailing the inner workings of the parser, let us simply introduce the input language via some examples. As mentioned previously, HIBOU accepts two kinds of input files: ".hsf" and ".htf" which we quickly cover respectively in Sec.12.2.1 and Sec.12.2.2.

12.2.1 Encoding interaction terms

In the example on Fig.12.1, we have on the left the encoding of an interaction term in a ".hsf" file. On the right we have a drawing of that interaction made by HIBOU.

We can see that we have some headers at the top of the ".hsf" file. In those headers, we declare the names of the lifelines and messages that will be used:

- in `@lifeline` we declare lifelines separated by ;
- in `@message` we declare messages separated by ;

Then, the interaction term is written in a similar fashion as it would be done using the previous mathematical notations. We have the *strict*, *seq*, *par*, *alt*, $loop_S$, $loop_H$, $loop_W$, $loop_P$ and *coreg* constructors and we use parenthesis to enclose sub-interactions. Notable differences with mathematical notations are that:

- for any associative operator, we allow n-ary notations. For instance, $seq(i_1, i_2, i_3)$ will then be interpreted as $seq(i_1, seq(i_2, i_3))$
- for ease of use, we use a dedicated notation for the empty interaction, with "`o`" (lowercase letter o) instead of $\varnothing$
- and we also use notations inspired by WebSequenceDiagrams [21] / PlantUML [5] for communication actions:

```
@message{
    m1;m2;m3;m4;m5;m6;m7;m8;m9
}
@lifeline{
    l1;l2;l3;l4
}
seq(
    alt(
        seq(
            alt(
                m1 -> l4,
                o
            ),
            l4 -- m1 -> l1,
            coreg(l2)(
                l1 -- m2 -> (l2,l3),
                loopX(
                    l3 -- m3 -> l2
                )
            )
        ),
        seq(
            l1 -- m4 -> l2,
            l2 -- m4 -> l3
        )
    ),
    par(
        l2 -- m5 -> l3,
        loopP(
            seq(
                l2 -- m6 -> l3,
                l3 -- m7 -> l4,
                l4 -- m8 ->|,
                l4 -- m9 -> l2
            )
        )
    )
)
```

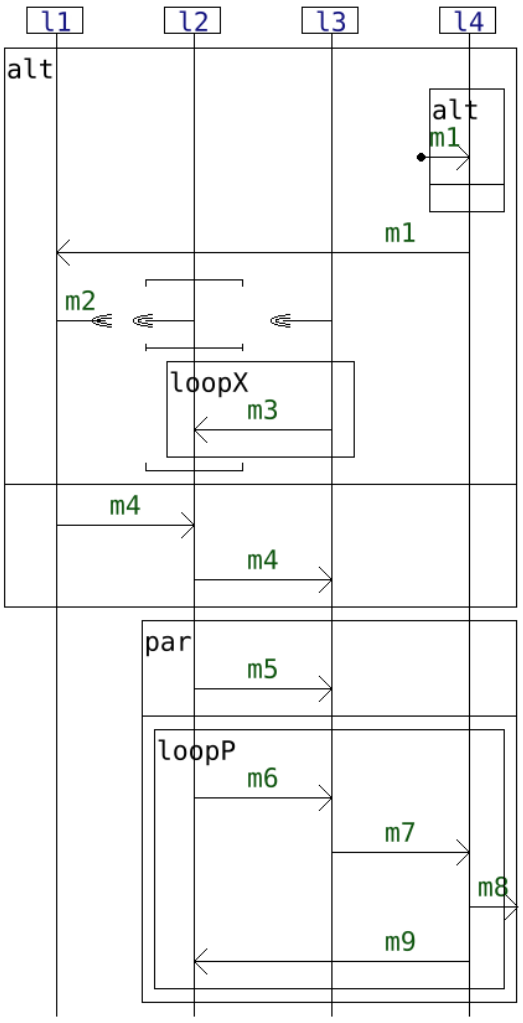


Figure 12.1: Example of an interaction encoded in HIBOU

– instead of $\varnothing$ we may write "`o`" (lowercase letter o)

– instead of $l_1!m$ we may write "`l1 -- m ->|`"

– instead of $l_2?m$ we may write "`m -> l2`"

– instead of $strict(l_1!m, l_2?m)$ we may write "`l1 -- m -> l2`"

– instead of $strict(l_1!m, seq(l_2?m, l_3?m))$ we may write "`l1 -- m -> (l2,l3)`"

Let us remark that we can also configure explorations and multi-trace analysis in the header of ".hsf" files. However, this will be covered in Sec.12.3 and Sec.12.4.

12.2.2 Encoding multi-traces

In the example on Fig.12.2, we have on the left the encoding of a multi-trace term in a ".htf" file. On the right we have a drawing of that multi-trace made by HIBOU (as part of an analysis). For the analysis of a multi-trace, the declaration of the sets M of messages and L of lifelines is made in the ".hsf" file specifying the interaction model. As a result, when parsing a ".htf" we presuppose the existence of L and M.

The encoding of multi-traces is quite straightforward. Each line correspond to a trace component and lines are separated by ";". On each line, we have on the left the definition of the co-localization corresponding to the trace component. This takes the form of a lists of lifelines from L separated by commas and in between square brackets. The same lifeline cannot be in several distinct co-localizations. Then we have the trace component as a sequence of atomic actions separated by ".". Each action is denoted with the notations $l!m$ or $l?m$ with l being a member of the co-localization and m in M.

```
{
    [l1,l2] l1!m1.l1!m3.l2?m1.l2!m2;
    [l3,l4] l3?m3.l3!m4.l4?m4;
    [l5]    l5?m2
}
```

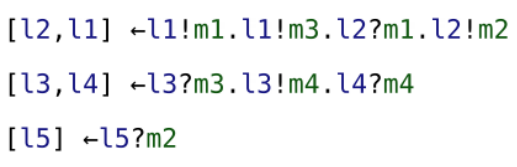


Figure 12.2: Example of a multi-trace encoded in HIBOU

We may use some additional keywords for ease of use:

- `[#all] l1!m1.l3!m4` for instance signifies that all lifelines are in this co-localization and therefore we have a global trace. For instance if $L = \{l_1, l_2, l_3, l_4\}$ then all of them are considered.
- by contrast, `[#any] l1!m1.l3!m4` signifies that the lifelines appearing on the definition of the trace component are taken into account to define the co-localization. For instance here, if $L = \{l_1, l_2, l_3, l_4\}$, then the co-localization is $\{l_1, l_3\}$ because only those two appear.
- if a lifeline from L is left without any co-localization that contains it, then an empty trace component is created for it

12.3 Semantic exploration and heuristics

We have mentioned in Chap.6 that the semantics that is implemented in HIBOU is the simplifying execution semantics $\sigma_{\underset{e}{\approx}}$. This enables us to compute and display the execution tree of any interaction term. We provide this option as a feature of HIBOU with the "`hibou explore`" command.

On Fig.12.4 is represented an exploration of the execution tree of the interaction specified on Fig.12.3. The graphical representation associated to this exploration and that is presented on Fig.12.4 is generated by HIBOU.

On the top of Fig.12.3, we can see that the exploration that is performed is configured in a certain manner which corresponds to that specified in the ".hsf" file from Fig.12.3 via the `@explore_option` section.

Notably, the generation of the image from Fig.12.4 is enabled by the `loggers = [graphic[svg,vertical]]` line. This line signifies that a graphical logger is created and listens to what happens during the exploration. This logger being a graphical logger, it draws the exploration. We can configure its output format (svg or png) and we can decide whether the drawing is drawn from top to bottom (vertical) or left to right (horizontal).

```
@explore_option{
    loggers = [graphic[svg,vertical]];
    strategy = DepthFS;
    filters = [ max_depth = 2,
                max_loop_depth = 1,
                max_node_number = 7 ]
}
@message{
    m1;m2;m3;m4
}
@lifeline{
    l1;l2;l3
}
seq(
    loopS(
        alt(
            l1 -- m1 -> l2,
            l2 -- m2 -> l3
        )
    ),
    par(
        l1 -- m3 -> l2,
        l3 -- m4 -> l2
    )
)
```

Figure 12.3: ".hsf" of the interaction which semantics is explored on Fig.12.4

12.3.1 Search strategies and filters

Continuing on our example, given that the interaction contains a loop, an exploration without constraints would not terminate. With HIBOU, via the `@explore_option` section, we can set some limits on the exploration. We can see here, with the "`filters`" that we have set the maximum depth of the explored tree at 2 (depth 0 being that of the root), that we only authorize loops to be instantiated once in each path, and that we authorize the final tree to contain a maximum of 7 nodes.

Moreover, we specified that the strategy for the exploration of the tree corresponds to a Depth First Search. From any interaction i, its frontier $\texttt{frt}(i)$ contains a set of position which we may order lexicographically. Then, the sub-trees starting from the execution of any such frontier action can be explored in that order. With a Depth First Search, given $\texttt{frt}(i) = [p_1, p_2]$, the sub-tree starting from i'_1 s.t. $i \xrightarrow{a@p_1} i'_1$ is entirely explored before starting the exploration of the other one. As a result, we can see on Fig.12.4, that some frontier actions of the initial interaction (at the top) have not been executed at all. Indeed, the nodes in brown on Fig.12.4 correspond to frontier actions that have not been exploited due to constraints on the

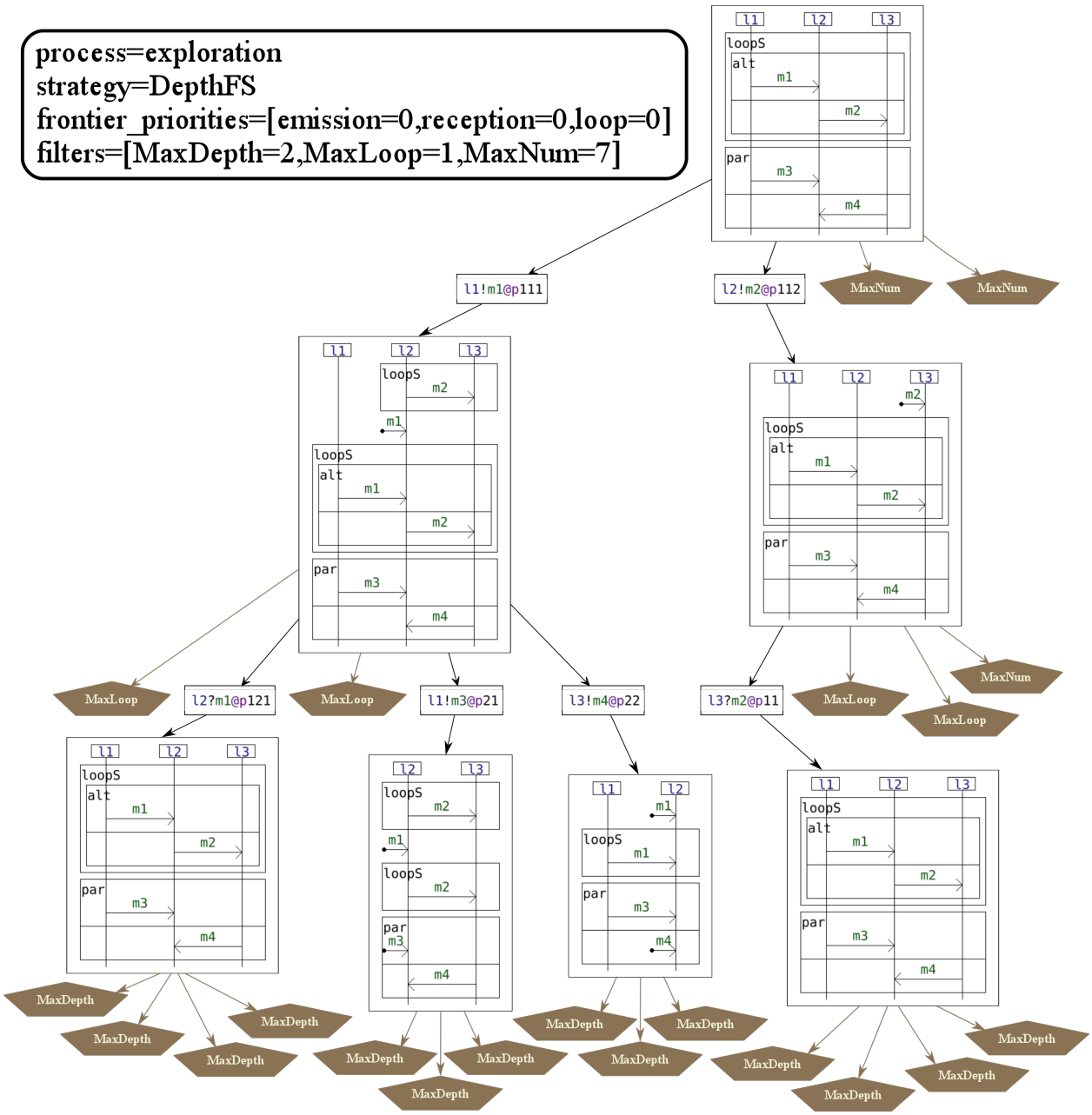


Figure 12.4: An exploration of the execution tree of the interaction specified in Fig.12.3 by HIBOU

size of the explored tree. In that example, we can see brown nodes caused by the three distinct constraints:

- "MaxLoop" correspond to frontier actions which were not executed due to having reached (in the path leading to that node) the limit concerning the number of loops that may be instantiated. On the example from Fig.12.4 we can see that the first execution of $l_1!m_1$ (first transition on the left of the topmost node) created a new instance of a loop. Given that our limit was fixed at 1, withtin that sub-tree, no loops can be instantiated anymore. Hence, on the left node in the second row, we cannot execute $l_2!m_2$ which is the first frontier position in lexicographical order. As a result, we have a brown "MaxLoop" node on the leftmost of the second row.

- "MaxDepth" correspond to frontier actions which cannot be executed due to having reached the limit concerning the maximum depth of the explored part of the execution tree. We can see here that all

frontier actions belonging to interactions from the third row cannot be executed and yield a brown "MaxDepth" node.

- "MaxNum" correspond to cases in which we have reached the maximum number of nodes in the overall tree. On Fig.12.4 we can indeed see that we have exactly 7 nodes containing interactions. This limit is reached when the node on the bottom right is created (as expected due to having a Depth First Search strategy). Then, if we look at the node on the right of the second row, we can see that we have 4 frontier actions, which, by lexicographic order of their respective positions are:
  - $l_3?m_2$ which is executed and yield the node underneath
  - $l_1!m_1$ which is not executed due to it being underneath a loop and the exploration having reached the maximum loop depth
  - $l_2!m_2$ which is not executed for the same reasons
  - and $l_1!m_3$ which is not executed because at this point in the exploration, we have already created 7 nodes and no new nodes are allowed to be created

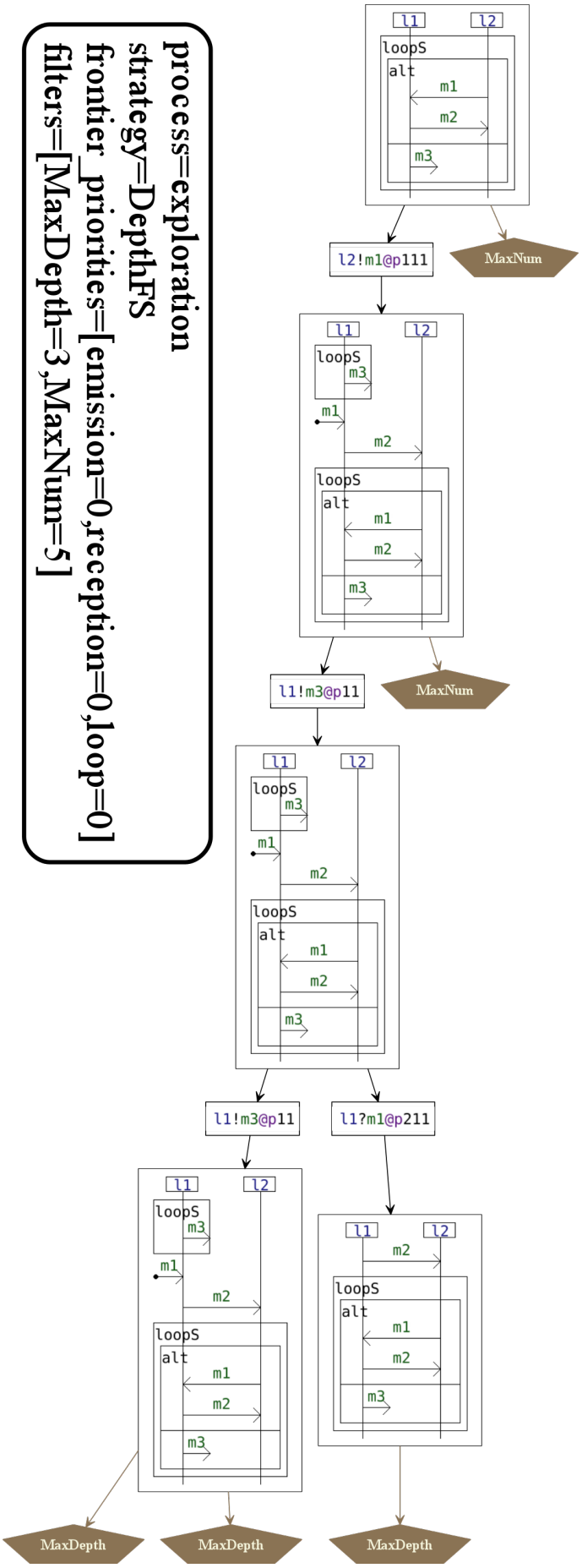
In addition to the Depth First Search strategy, which we may select using "`strategy = DepthFS;`", HIBOU also implements Breadth First Search which can be selected with `strategy = BreadthFS;`".

12.3.2 Frontier priorities

We have seen that, with HIBOU, the order in which frontier actions are evaluated to compute child nodes is, by default, the lexicographic order of their respective positions. Hence, actions "at the top" of the sequence diagram representation will be evaluated before those "at the bottom". We have seen that this is the case for "exploration", which computes parts of execution trees but this also holds for "analysis" which computes parts of analysis graphs. This can be problematic in certain cases. In particular, let us consider the example from Fig.12.5. We can see that in this exploration, we have set a maximum depth of 3 and a maximum number of nodes at 5. We then use a Depth First Search strategy to explore the execution tree of the interaction. At the first transition, we execute action $l_2!m_1$ underneath the $loop_W$. Given that in the follow-up interaction, we have the $loop_W(l_3!m_3)$ that was put in front, due to the pruning of the initial loop, it is then action $l_1!m_3$ that is at the first position in the lexicographic order of positions. Then, given that we have $loop_W(l_3!m_3) \xrightarrow{l_3!m_3} loop_W(l_3!m_3)$ we can indefinitely execute $l_3!m_3$ which may not necessarily be an interesting exploration.

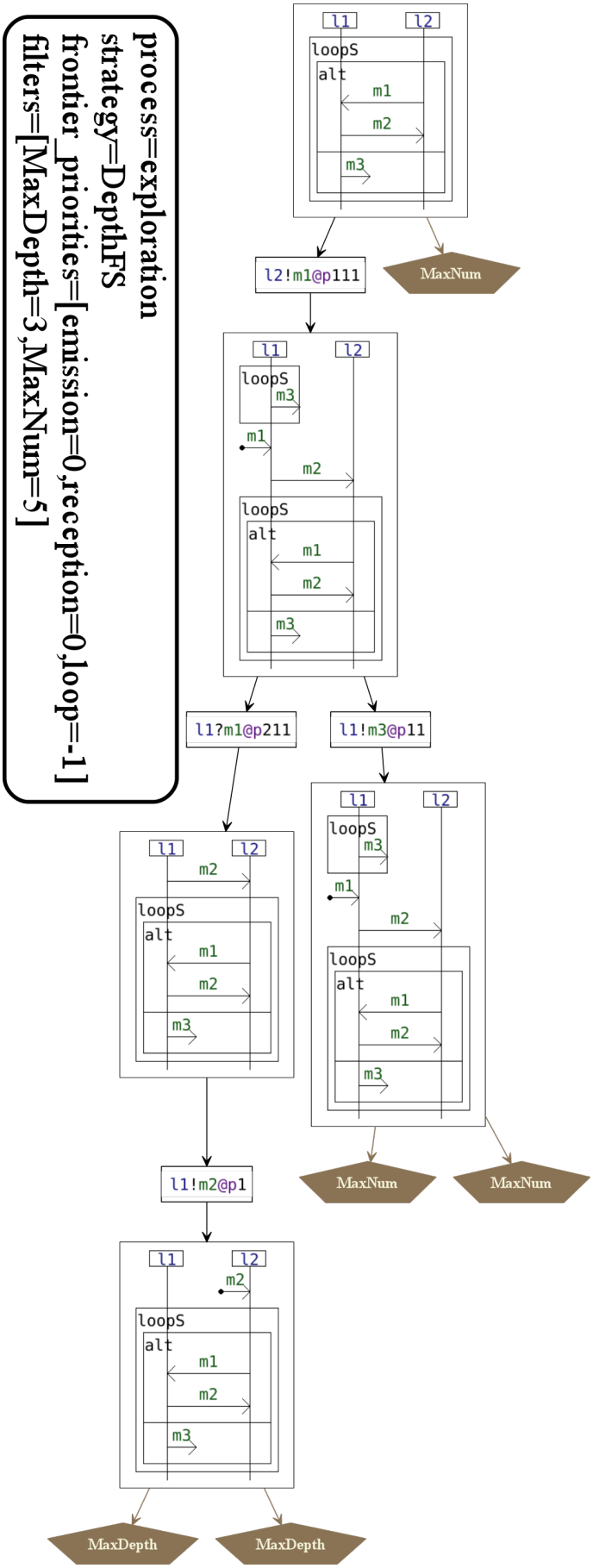
The concept of frontier priorities is to change the order in which frontier actions are evaluated. In HIBOU, we can set integer priority levels that may change the order of actions depending on:

- whether they are emissions of receptions
- whether or not they are underneath loops and at which depth




```
@explore_option{
  loggers=[graphic[svg]];
  strategy=DepthFS;
  filters=[max_depth=3,
           max_node_number=5]
}
@message{
    m1;m2;m3
}
@lifeline{
    l1;l2
}
loopW(
    alt(
        seq(
            l2 -- m1 -> l1,
            l1 -- m2 -> l2
        ),
        l1 -- m3 ->|
    )
)
```

Figure 12.5: Example exploration without setting frontier priorities (default)




```
@explore_option{
  loggers=[graphic[svg]];
  strategy=DepthFS;
  filters=[max_depth=3,
           max_node_number=5];
  frontier_priorities=[loop=-1]
}
@message{
    m1;m2;m3
}
@lifeline{
    l1;l2
}
loopW(
    alt(
        seq(
            l2 -- m1 -> l1,
            l1 -- m2 -> l2
        ),
        l1 -- m3 ->|
    )
)
```

Figure 12.6: Example exploration with setting a -1 priority to frontier actions underneath loops

On Fig.12.6 we showcase the use of frontier priorities by performing the same exploration as in Fig.12.5 (i.e. same search strategy and same filters) but with having set a priority of -1 to actions underneath loops. We can then observe that the action that is at the first position in the second node from the top is now $l_1?m_1$ instead of $l_1!m_3$ (which was the case in Fig.12.5). This is explained by the fact that $l_1!m_3$ is underneath a loop while $l_1?m_1$ is not. In more details, $l_1?m_1$ is at loop level 0 while $l_1?m_1$ is at loop level 1. Then their relative priorities are respectively $0 * (-1) = 0$ and $1 * (-1) = -1$ and hence $l_1?m_1$ is of higher priority for evaluation. We can then see that we manage to execute almost entirely the sub-behavior that was instantiated (not entirely due to the depth limit).

With this example, we have showcased the use of frontier priorities in the case of exploration i.e. the computation of parts of some execution trees. However, frontier priorities may be more particularly useful in context of multi-trace analysis.

12.4 Multi-trace analysis

In Chap.9, Chap.10 and Chap.11 we have defined various multi-trace analysis algorithms. All of those algorithms are implemented in a configurable manner in HIBOU.

We have seen in Sec.12.3 that the exploration of an execution tree could be configured via the declaration of options within a `@explore_option` section of the input ".hsf" file. Similarly, the analysis of a multi-trace against an interaction can be configured via a `@analyze_option` section. Those options includes those already available to `@explore_option` i.e.:

- we can use for instance `loggers = [graphic[png,horizontal]];` to create a logger that will draw the process from left to right in a png file. This can notably be helpful for debugging e.g. understanding why a certain multi-trace is non-conform. However generating a "svg" image is most often a better choice. Also it is not recommended to use a graphical logger if the time required for the analysis is a limiting factor given that it reduces the overall performance of HIBOU.

- we can set a search strategy with for instance `strategy = DepthFS;`. Using a Depth First Search is most often recommended for multi-trace analysis given that it favors going deeper in some paths in which many actions from the multi-trace can be consumed instead of exploring bits of many different paths.

- we can impose limits on the exploration of the analysis graph, with for instance `filters = [max_depth = 3];` However imposing such limitations in the case of multi-trace analysis is not recommended given that it removes the guarantees on the correctness of the returned verdict given by the formal proof. Indeed imposing such limits artificially restricts which parts of the analysis graphs can be reached and hence some local verdicts may be missed.

- we can set frontier priorities for the evaluation of frontier actions, with for instance `frontier_priorities = [reception=1];`

In addition, some other options are available for the `@analyze_option` section. Those are specifically related to the analysis process and include:

- the "`use_locfront`" option which specifies whether or not the checking of local frontiers (as defined in Chap.10) should be used with:
  - "`use_locfront = false`" specifying that local frontiers should not be checked
  - "`use_locfront = true`" specifying that local frontiers should be checked, and in case of failure, a local verdict $Dead$ is reached rather than pursuing the analysis
- the "`goal`" option which specifies a criterion for stopping the analysis before the whole of the reachable parts of the analysis graph is explored:
  - with "`goal = None`" the analysis explore the whole of the graph
  - with "`goal = Pass`" the analysis stops and returns $Pass$ once a Cov local verdict is reached
  - with "`goal = WeakPass`" the analysis stops and returns $Pass$ once a Cov local verdict is reached and also stops and returns $WeakPass$ once (depending on which algorithm is selected using "`analysis_kind`") either of a $TooShort$, $MultiPref$ or $Slice$ local verdict is reached
- the "`analysis_kind`" option which can be set to specify which analysis algorithm should be used:
  - "`analysis_kind = accept`" specifies that the algorithm from Chap.9, for identifying exactly accepted multi-traces should be used. This algorithm returns $Pass$ if $\mu \in \sigma_{|C}(i)$ and $Fail$ otherwise
  - "`analysis_kind = prefix`" specifies that a variant of that algorithm, which may discriminate between $\sigma_{|C}$ and $\sigma^{\dagger}_{|C}$ should be used. This algorithm returns $Pass$ if $\mu \in \sigma_{|C}(i)$, $WeakPass$ if $\mu \in \sigma^{\dagger}_{|C}(i)$ and either $Inconc$ or $Fail$ otherwise. The use of $Inconc$ here is related to the uncertainty of whether or not a multi-trace μ is in $\overline{\sigma_{|C}}(i)$. This uncertainty cannot be solved without using either hiding or simulation.
  - "`analysis_kind = hide`" specifies that a generalized version of the algorithm with hiding steps from Chap.10 should be used. This algorithm behaves exactly the same as the one from Chap.10 whenever the multi-trace is defined up to the discrete partition. However, if the partition is not discrete, then a local verdict $Inconc$ might be produced rather than a $MultiPref$.
  - "`analysis_kind = simulate[multi-prefix]`" corresponds to the algorithm with simulation steps only as a postamble to the consumption of trace components (as in Sec.11.1.2)
  - "`analysis_kind = simulate[multi-slice]`" corresponds to the algorithm with simulation steps both as a preamble and as a postamble to the consumption of trace components (as in Sec.11.1.3)

On Fig.12.7 we illustrate the analysis of a multi-trace defined on a partition $C = \{\{l_1, l_2\}, \{l_3\}\}$ of lifelines. This multi-trace is a slice of a multi-trace that is accepted by the interaction $i = coreg(\{l_2\})(seq(\cdots), loop_W(\cdots))$. Indeed, the actions that are missing i.e. have not been observed are:

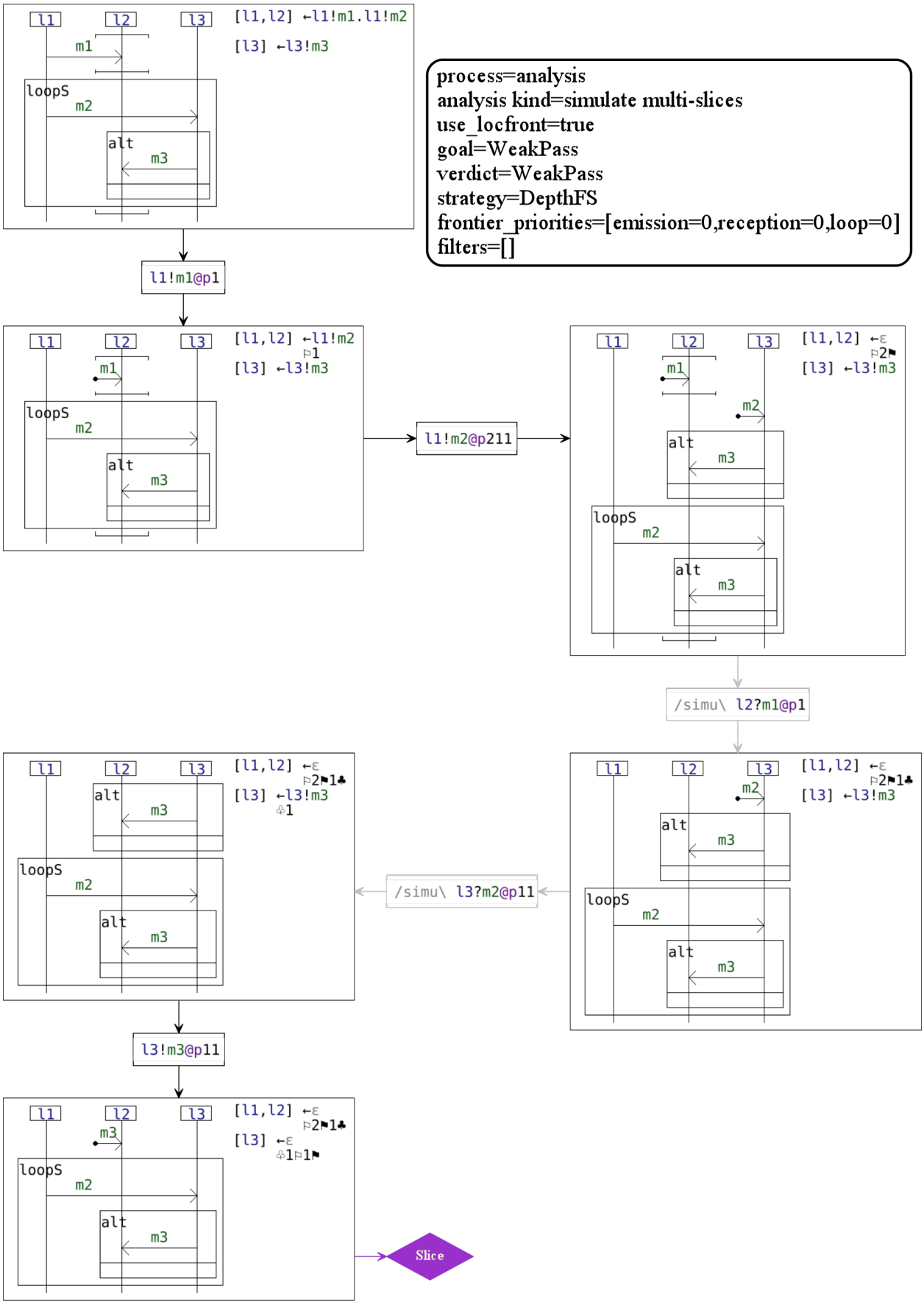


Figure 12.7: Showcasing the algorithm with simulation steps as preamble and postamble

- occurences of $l_2?m_1$ and $l_2?m_3$ (in any order) at the end of component $\{l_1, l_2\}$
- an occurence of $l_3?m_2$ at the beginning of component $\{l_3\}$

The analysis that is illustrated on Fig.12.7 make use of the algorithm from Sec.11.1.3, with allows simulation steps both as a preamble and as a postamble to the consumption of actions. We can see that with a Depth First Search, the local verdict "Slice" is found quickly. And, given that we have set the goal to "WeakPass", the algorithm immediately returns.

Conclusion

In this chapter we quickly presented the HIBOU tool which implements most results from the first two parts of the thesis. We have seen that this tool allows for the configurable analysis of multi-traces against interaction models, which was one of our initial objectives.

Chapter 13

Extension to data

Contents

In this chapter we discuss extending our Interaction Language to include interactions enriched with data, and extending multi-trace analysis to include multi-traces enriched with concrete message parameters.

The plan of this chapter is as follows:

- in Sec.13.1 we discuss a specific limitation caused by abstracting information that is exchanged within a DS with simple labelled messages. We then explain that taking into account the passing of concrete data (value passing) can be a solution to this limitation.
- in Sec.13.2 we discuss the manner in which data can be introduced into interaction models.
- in Sec.13.3 we discuss the formalisation of interactions enriched with data.
- and, in Sec.13.4 we provide some more details on the use of symbolic execution to explore the semantics of symbolic interactions and analyse multi-traces enriched with data.

13.1 An issue with abstracting exchanged information as messages

A working hypothesis that comes with using our IL and the framework for multi-trace analysis provided by HIBOU is that the information that transit at the interfaces of the DS's sub-systems can be abstracted as simple labelled messages $m \in M$ from a set of messages M. Moreover, given that interaction terms are finite trees, within any $i \in \mathbb{I}_\Omega$, there can be only a finite number of distinct messages that appear. As a result, within the context of the analysis of a multi-trace against an interaction, we might as well consider M to be a finite set.

Abstracting away typed data with an arbitrarily wide space of possible values into a finite set of labels obviously comes with its own set of limitations. In this section we present one such limitation (among many) and explain how value passing, and by extension, the consideration of interaction models enriched with data, can help mitigate this limitation.

13.1.1 A limitation on examples with $loop_P$

Let us consider the example from Fig.13.1. In this example, we have a simple $loop_P$ which specifies the parallel repetition of instances of the passing of message m_1 from lifeline l_1 to lifeline l_2.

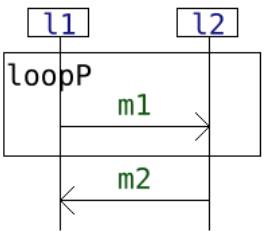


Figure 13.1: Example interaction with a $loop_P$

Let us then suppose that several instances of message m_1 are emitted consecutively. Then, there are no

means to distinguish between the receptions events $l_2?m_1$ that might correspond to the various instances of the emission events $l_1!m_1$.

On Fig.13.2 we illustrate the analysis of the global trace $t = l_1!m_1.l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1.l_2?m_1.l_2!m_1$ against the interaction from Fig.13.1.

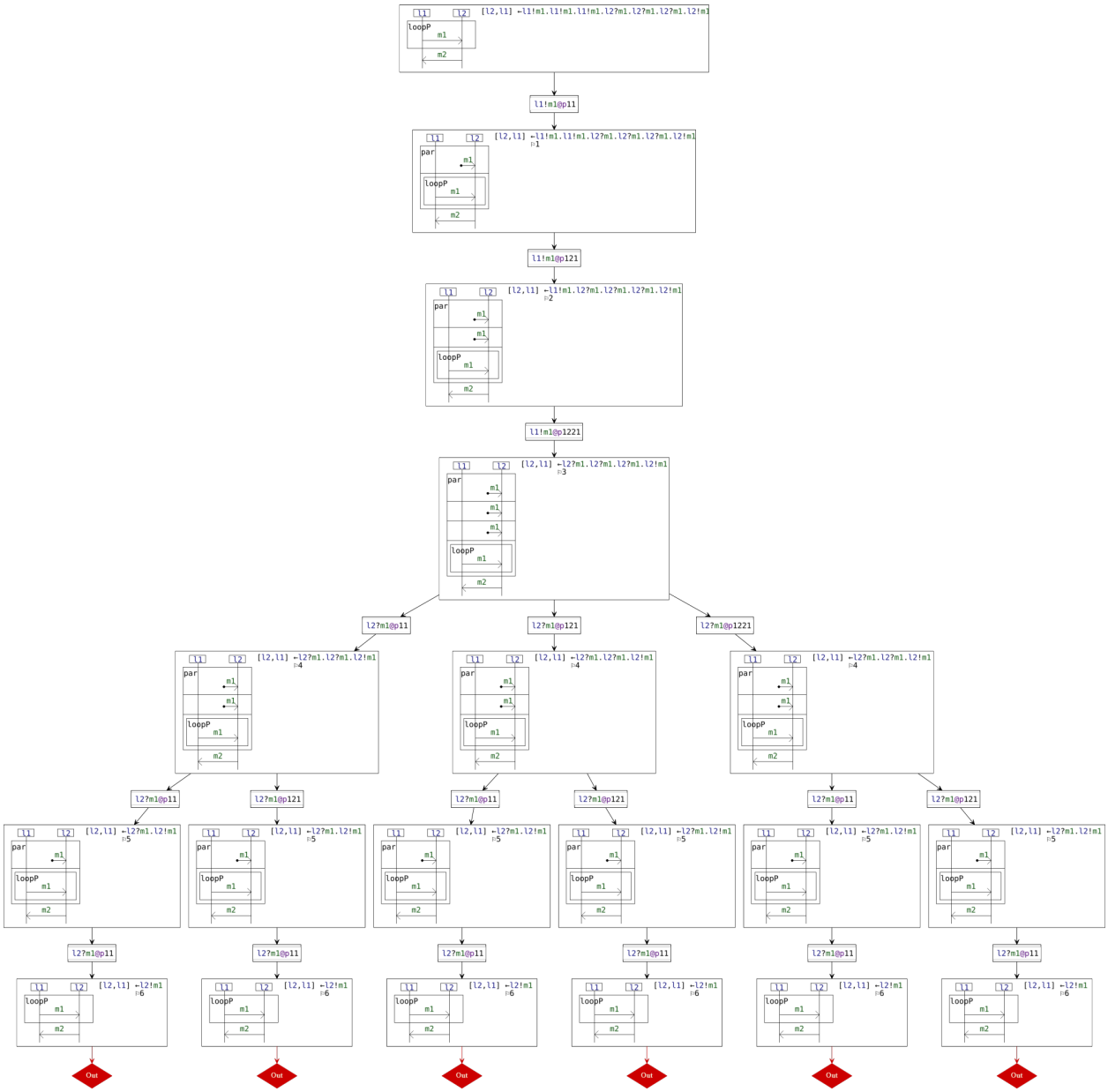


Figure 13.2: Example analysis against the example from Fig.13.1

We can see that this trace is not an accepted trace (nor a prefix of one). However, as illustrated on Fig.13.2, proving that this is the case is relatively costly given that we need to explore all paths in the analysis graph. Here, even an extended version of the checking of local frontiers (which would project the remaining trace components on the discrete partition) would not help at all given that all that remains to analyze occurs on the same lifeline which is l_2. The reason why there are so many paths that open-up in this analysis is the use of the $loop_P$ constructor and the inability to distinguish between the 3 instances of the passing of m_1. Indeed, starting from the vertex in the middle of Fig.13.2 (before the split), the three

remaining instances of $l_2?m_1$ on the multi-trace could each be interpreted as either of the three instances of $l_2?m_1$ that are scheduled with *par* within the remaining interaction term.

We have already evoked this problem in our publication [93] where we used a partial high-level modelisation of the MQTT telecommunication protocol (version 3.1.1) [23] as a use-case for global trace analysis. The diagram representation of the corresponding interaction model is given on the left of Fig.13.3. This model states that a communication session between a client and a broker starts with a sequential connection phase and ends with a disconnection phase. In between, at any time, any number of instances of one of the 5 proposed subinteractions can be run concurrently. The use of $loop_P$ reflects that sessions can be opened and run concurrently at any time.

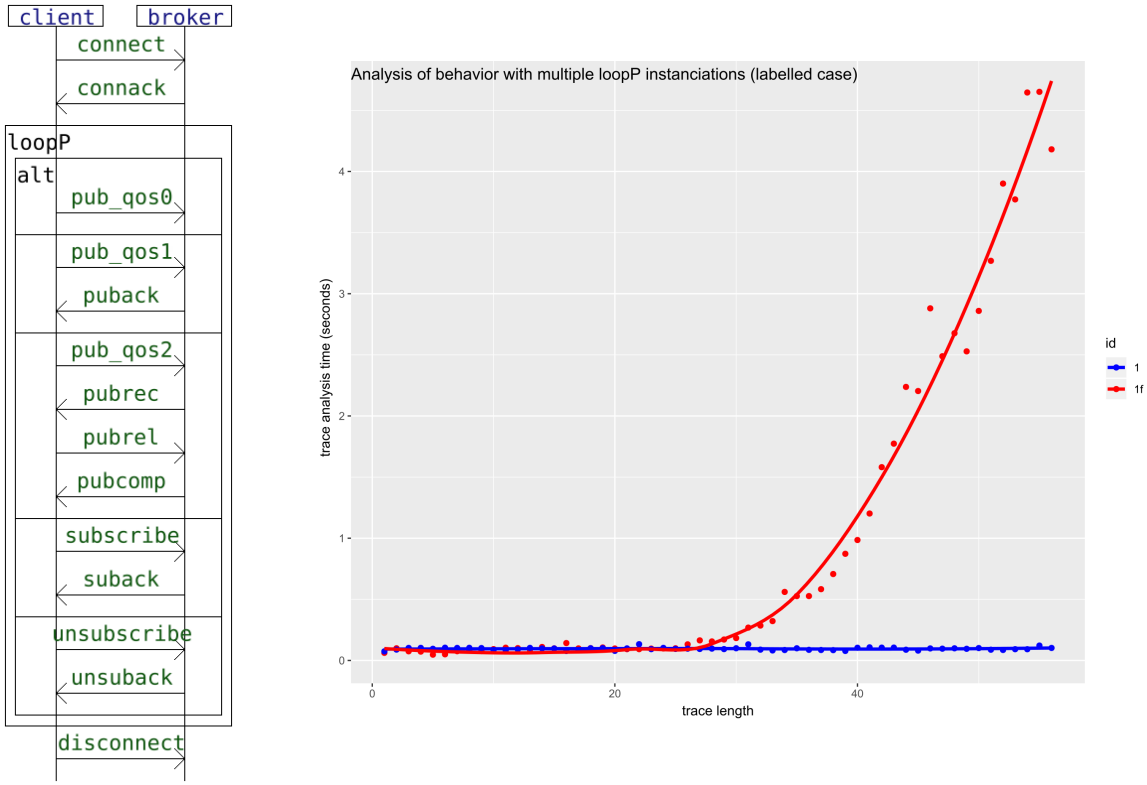


Figure 13.3: Abstract model of the MQTT protocol (v.3.1.1) and analysis time on particular edge cases

In [93], we plotted the required time for the analysis of traces generated by a multithreaded Python script which implemented this communication scheme. On the right of Fig.13.3 we present an updated and simplified plot of similar data, obtained using a newer version of HIBOU. On this diagram we can see two curves:

- The curve in blue (below) represent the time required for analyzing an accepted trace and all its prefixes. The curve is indeed obtained by linear regression on the data points (blue dots), each representing the time required for analyzing a prefix of that trace, with the right-most blue dot being the time required to analyze the trace in its entirety. For collecting this data, we have used an analysis with a Depth First Search strategy and we stopped whenever we obtained a Cov or $TooShort$ local verdict (i.e. $Pass$ or $WeakPass$ global verdict). In this data (in blue) we cannot see any increase of the time that is

required for the analysis w.r.t. the size of the trace because the time taken by the tool for parsing the input files is more significant at this scale.

- Each data point in red (that make up the curve in red (above)) corresponds to the time required for analyzing a mutant of a corresponding blue data point. This mutant is obtained by adding an unexpected action at the end which guarantees that the trace is not an accepted trace nor a prefix of one. As a result, each red data point corresponds to the analysis of a trace which returns *Fail* and hence requires the exploration of the whole analysis graph (instead of stopping at the first *Cov* or *Short*) as in Fig.13.2. In order to obtain this curve we have chosen a particular "worst-case" scenario, with many overlapping instances of the passing of the `pub_qos0` message. We can see, as could be imagined given the example from Fig.13.2, that we have an exponential increase in the time required to terminate the analysis.

13.1.2 Mitigating the problem with value passing

The problem which we have described in the previous section is specifically related to the use of $loop_P$ and to the inability to distinguish between several instances of communications which are abstracted by the same message "label". This limitation of "labelled interactions" i.e. interactions in which exchanged information is abstracted as messages by labels can be mitigated by conserving more information when abstracting observed communication.

In the particular case of communication protocols, such as the MQTT protocol presented in the example from Fig.13.3, individual sessions of communication (for instance the publication of a message, a subscription, etc.) are uniquely identified by an integer message identifier. By incorporating the information of those message identifiers into the abstracted trace and interaction model we can therefore mitigate the aforementioned effect.

Let us indeed consider the example analysis from Fig.13.5, which reproduces the analysis from Fig.13.2 but with the added information of unique message identifiers for instances of the message m_1. We can see on the vertex before the split that we have in the remaining interaction term three sub-interactions with a reception of m_1 that are scheduled with *par*, as in the previous case. However, the three messages expected to be received can be distinguished from one another. In the top instance, $m1(12)$ with the unique identifier 12 is expected while in the second instance it is $m1(55)$ and in the third $m1(76)$. Then, when analyzing the three reception events that remain to be analyzed, those can be uniquely matched to specific instances of the $loop_P$. On Fig.13.5 the cases in which the concrete data in the message parameters do not match are indicated by the "UNSAT" vertices.

As a result of having taken into account additional information for distinguishing instances of messages, we can see that the resulting analysis graph is significantly reduced in size (from 19 vertices to 7 vertices). This reduction in the size of the explored analysis graph is exponential with the size of the trace to analyze, provided that the message identifiers are indeed unique (or at least that the use of identical identifiers is

sparse enough so as not to incur ambiguities). This then allows us to avoid the worst case depicted by the red curve on Fig.13.3.

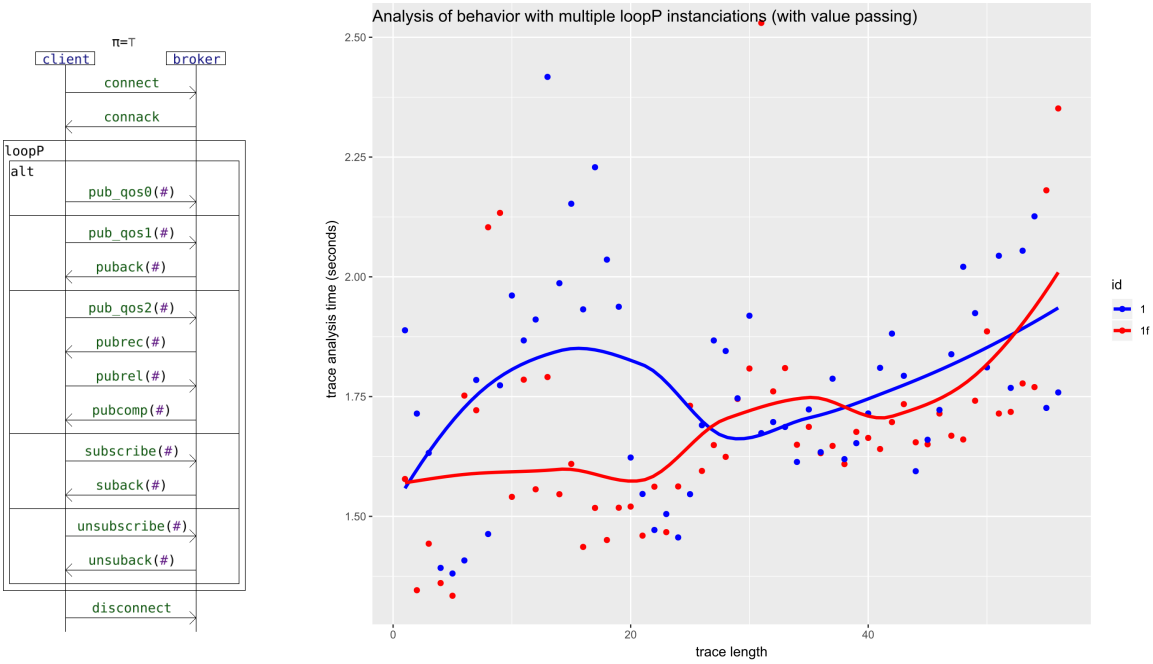


Figure 13.4: Model of MQTT with value passing for message identifiers and analysis time on edge cases

In Fig.13.4 we have indeed applied the same process to compare the analysis time of traces enriched with data against the MQTT model enriched with value passing for taking into account the unique message identifiers. As in the previous case, we have used a multithreaded Python script to generate the input trace and we have plotted two curves: one (in blue) describing the analysis time of the of the accepted trace and its prefixes and the other (in red) describing the analysis time of corresponding mutants (defined by adding an unexpected event at the end). we can observe that, by contrast to Fig.13.3, there is no exponential explosion of the red curve. We have therefore successfully annulled the issue caused by the use of $loop_P$.

Let us remark that the code for reproducing the results of this small experiment with a MQTT model is available in [92]. With this experiment we have seen that taking into account value passing can improve the quality of the analysis of multi-traces. However, we can go one step further by enriching the IL with variables, guards and message parameters, which is the object of the following section.

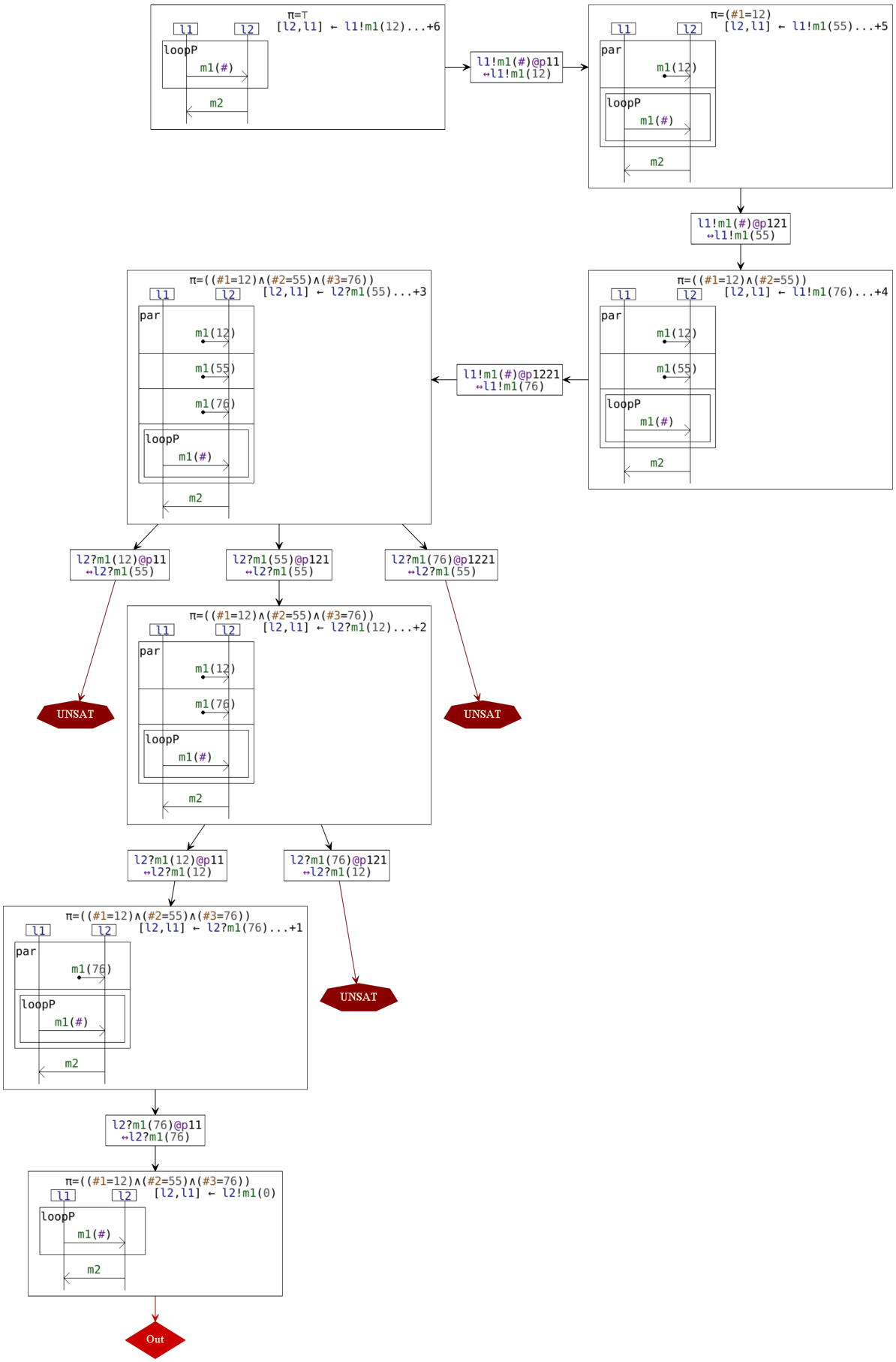


Figure 13.5: Analysis with added information of unique message identifiers

13.2 Introducing data

The Interaction Language which we have defined in Chap.4 can be canonically extended to include data. Those interaction models enriched with data may involve (informally):

- "variables" that can be owned by specific lifelines and which values may vary during the execution of the interaction. Variables are strongly typed and local to specific lifelines.
- "message parameters" i.e. the messages which are exchanged between lifelines are not simple labels anymore but instead they can carry some values which can be computed on the fly (at the moment of the emission) and transmitted between lifelines
- "assignments" which are basic operations that can accompany the observation of an event within the interaction model (emission or reception) and modify the value of a given "variable". An instance of this variable can be already owned by the lifeline on which the assignment occurs, and in this case, its value is simply modified. If, on the other hand, an instance of the variable do not exist on the lifeline, then it is created when evaluating the assignment. The value that is assigned to a variable by an assignment is computed from a term that can be constructed from raw values, variables, message parameters and typed operators.
- "guards", which are Boolean conditions on the expression of a certain events (emission or reception). Those Boolean expressions constrain the variables (their current values) of the lifeline on which the event occurs so that the event can only occur if those constraints are satisfiable.

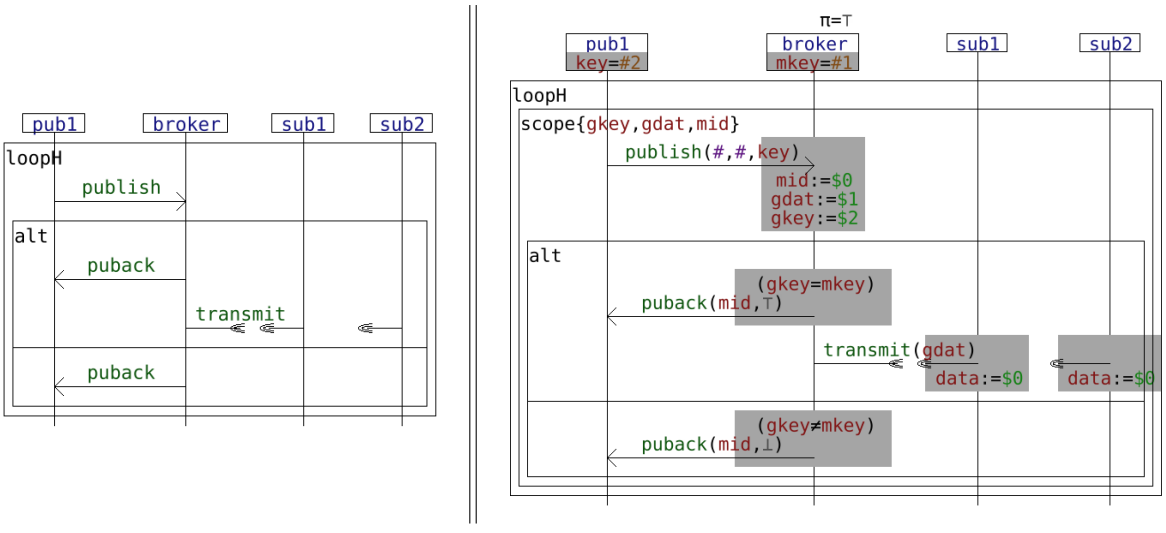


Figure 13.6: Refinement of a labelled interaction into a symbolic interaction with data

On Fig.13.6, we illustrate how one can use those new concepts so as to refine a "labelled" interaction model (on the left, without data), into a "symbolic" interaction model (on the right, with data). We can see that the overall structure of the interaction remains the same. However, parameters are attached to messages and statements accompany emission and reception events. Those blocks of statements are executed either

before the execution of the communication action (if they are drawn above) or after it (if they are drawn below). Moreover, the interaction comes with the knowledge of which variables exist and which are their values (for instance the `key` variable on lifeline `pub1` has the value #2 which is here an unknown value represented by an instantiated symbol) as well as constraints on those values (here there are no constraints as indicated by $\pi = \top$ at the top). The reader should understand the use of the # sign as a call to the creation of a new symbol (of symbolic execution) i.e. a new unknown of the problem while the signs #1, #2 and any #j (with j an integer value) represent already created symbols.

We can then use symbolic execution to animate those enriched interaction models. Instead of assigning concrete values to variables, we assign terms build over data which may be abstracted as symbols. Those symbols correspond to input data that may enter the system through any means (user input, random values, etc.). The use of symbols in this manner reflects what is classically done in the symbolic execution of programs [73, 33].

Using symbolic execution instead of concrete execution allows us to synthesize in a compact execution tree (or analysis graph) explorations that would otherwise be far larger. Indeed, a constraint on a symbol (for instance $\#1 > 0$) might synthesize an arbitrarily large span of data (for instance positive 64 bits integers).

13.3 A discussion on formalizing interactions with data

In this section we discuss the formalisation of interactions with data. The contents of this sections do not constitute a formalisation of interaction with data and of their semantics. We simply mean to discuss the nature of those objects and to help the reader have a better understanding of their semantics.

As we have seen in the example from Fig.13.6, sequences of assignments and guards can form a preamble or a postamble to the execution of specific communication actions. We can associate those sequences to simple programs. Then, each small-step of the execution of an interaction with data may be understood as the simultaneous execution of the interaction model and of those programs that share a global state.

13.3.1 Data signature and syntax of programs

Let us consider a data signature $\Sigma = (S, Op)$ where S is a set of type names and Op is a set of typed operations i.e. for any $g \in Op$, there exists an arity $n \geq 0$ and types $s_1, \cdots, s_{n+1}$ from S such that the operation g is of profile $s_1 \times \cdots \times s_n \to s_{n+1}$. Let us also remark that the definition of concrete values for each type is also included in this definition, given that each concrete value (for instance integer 1) may correspond to an operation of arity 0 (constant) of the corresponding type.

In the continuation of our concern for the distributed nature of behaviors, we also consider a set L of lifelines for defining programs. Let us then also consider a set V of variables which is indexed by L i.e. such that we have $V = \bigcup_{l \in L} V_l$. Then:

- the set of typed terms $T_\Sigma(V_l)$ build over V_l w.r.t. Σ can be defined as is classically done (as presented in Chap.4 but respecting the types signature of operators).

- and the set of sentences $S_\Sigma(V_l)$ can be defined as the set of all equations on terms of $T_\Sigma(V_l)$ using the rules of equational logic (as presented in Chap.4 but respecting types and with the rule for the negation)

We can then define a program (P) as a (possibly empty) list of statements. Such a statement can either be an assignment (A), the creation of a new symbol (F for "fresh"), or a guard (G). Assignments $l : v := t$ specify that the term $t \in T_\Sigma(V_l)$ must be assigned onto variable $v \in V_l$. In the context of an action occurring on a lifeline l, we denote by $v := \#$ the creation of a new symbol and its assignment to variable $v \in V_l$. Guards $l : s$ specify that the Boolean condition expressed by $s \in S_\Sigma(V_l)$ must be satisfied. The program language can therefore be defined as:

$$\begin{aligned} P &::= (A \mid F \mid G)^* \\ A &::= l : v := t \\ F &::= l : v := \# \\ G &::= l : s \end{aligned}$$

Now that we have defined the syntax of the language which represents what can be executed as a preamble or postamble to communication actions, let us discuss the associated operational semantics.

13.3.2 Symbolic state & execution of programs

An operational semantics associated to this program language can be defined as a transition relation between "states" which represent the state of the system.

By system we understand the set of lifelines from L, each of which may own variables which in turn are associated to values, etc. In addition, given that we have discussed using symbolic execution instead of concrete execution, those values are not concrete values but terms build over symbols. Let us therefore define a universe $\mathcal{X}$ of symbols.

Then, this notion of state can be described by a "symbolic state" which is a triplet (X, η, π) where:

- $X \subset \mathcal{X}$ keeps track of the symbols which have already been introduced as input data during the execution
- $\eta : V \to T_\Sigma(X)$ associates to each variable its current value as a term build over the already introduced symbols from X. η may be called the "interpretation".
- $\pi \in S_\Sigma(X)$ is an equation on the symbols from X which keeps track of the constraints on (the values of) those symbols that may be introduced during the execution. π may be called the "path condition".

A symbolic state (X, η, π) is a representation of the current state of the system, which contains all the information that we require. We can then define the semantics of the program language by relating the execution of individual statements to a transition between two symbolic states.

Such a transition relation between symbolic states is defined on Fig.13.7. We indeed have that:

$$\frac{}{(X,\eta,\pi) \xrightarrow{l:v:=t} (X,\eta[v \leftarrow \eta(t)],\pi)}\ \text{assignment}$$

$$\frac{}{(X,\eta,\pi) \xrightarrow{l:v:=\#} (X \cup \{x\},\eta[v \leftarrow x],\pi)}\ \text{symbol creation } (x \in \mathcal{X} \setminus X)$$

$$\frac{}{(X,\eta,\pi) \xrightarrow{l:s} (X,\eta,\pi \wedge \eta(s))}\ \text{guard}$$

$$\frac{}{(X,\eta,\pi) \xrightarrow{\langle\rangle} (X,\eta,\pi)}\ \text{empty program}$$

$$\frac{(X,\eta,\pi) \xrightarrow{s} (X',\eta',\pi') \qquad (X',\eta',\pi') \xrightarrow{P} (X'',\eta'',\pi'')}{(X,\eta,\pi) \xrightarrow{s;P} (X'',\eta'',\pi'')}\ \text{non-empty program}$$

Figure 13.7: Symbolic execution of the program language

- whenever an assignment $l : v := t$ is carried-out then we update the value of variable v in the interpretation η with the current interpretation of the term t which is assigned to it. In other words, we change the interpretation η into $\eta[v \leftarrow \eta(t)]$
- whenever a symbol creation $l : v := \#$ is carried-out, we must create a new symbol, which is not in X. This is always possible given that $\mathcal{X}$ is infinite and X is finite. Hence, in the next symbolic state, we have $(X \cup \{x\}, \eta[v \leftarrow x], \pi)$ because we keep track of x being introduced and of η being updated with the new value assigned to v.
- whenever a guard $l : s$ is evaluated, the constraint specified by s must be satisfied. As a result, in the next step, we update the path condition π as $\pi \wedge \eta(s)$ to keep track of the newly introduced constraints.
- the empty program i.e. the empty list of statements can be executed and does not change the symbolic state
- we can then define the semantics of the program language as a transition relation such that for any program P we have a certain $(X,\eta,\pi) \xrightarrow{P} (X',\eta',\pi')$. This is described on Fig.13.7.

13.3.3 Informal illustrative examples

Let us consider the three examples illustrated on Fig.13.8. In the following we describe informally those three examples which serves the two purposes of (1) helping the understanding of assignment, symbol creation and guard and (2) introducing the concept of merging interaction models and programs of the program language.

Example on the left In the example displayed on the left of Fig.13.8, we have a single lifeline l, which, initially (at the top) has a variable v of value $\#2$ which is a symbol. Hence, the state of the system is a certain (X_0, η_0, π_0) in which $\#2 \in X_0$ and $\eta_0(v) = \#2$.

Then, we consider the successive executions of two transformations. The first one consists in the assignment $l : v := (1 + v)$. Given that $\eta_0(v) = \#2$, evaluating this assignment leads to (X_1, η_1, π_1) such that

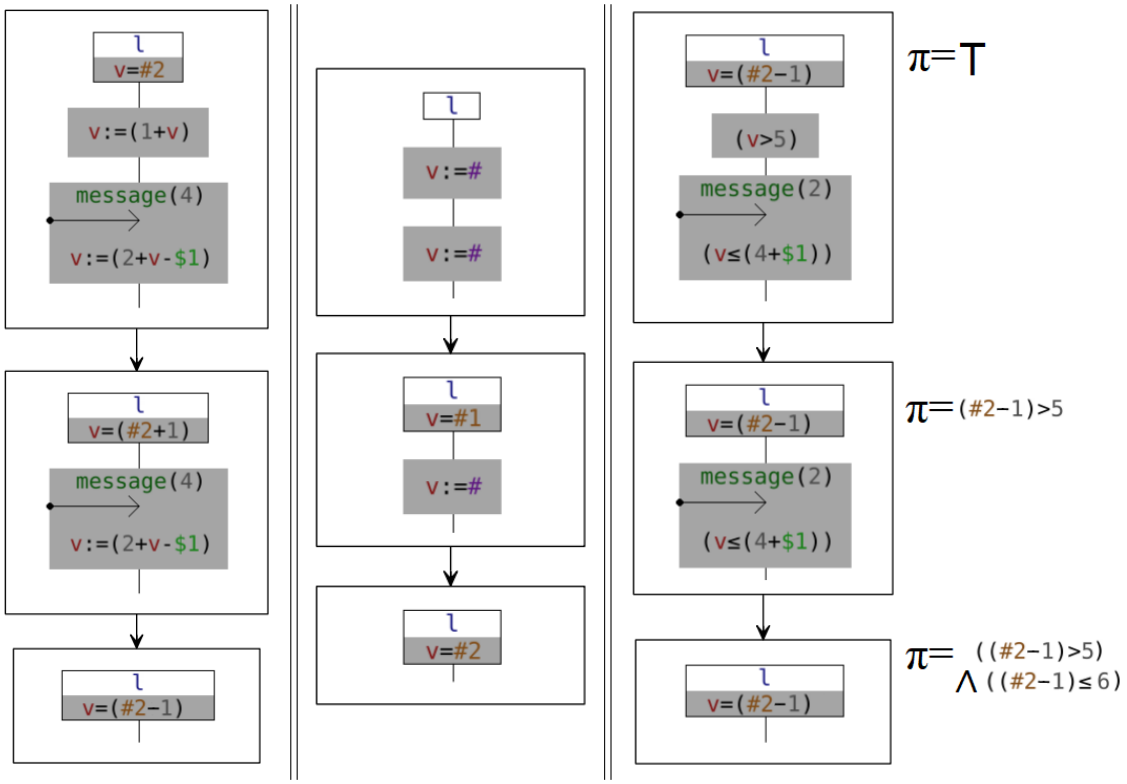


Figure 13.8: Illustrating the notions of assignment, symbol creation and guard

$X_1 = X_0$, $\eta_1 = \eta_0[v \leftarrow \eta_0(1+v)]$ and $\pi_1 = \pi_0$. Given that $\eta_0(1+v) = 1 + \eta_0(v) = 1 + \#2$ the evaluation of that assignment is correctly illustrated on Fig.13.8.

The second step consists in the reception, on lifeline l, of a message which carries a concrete integer value of "4". This reception is immediately followed by the evaluation of an assignment in which the term $(2 + v - \$1)$ is assigned to v. Here, the symbol $\$1$ refers to the first argument of the message, which, in this context, is the concrete value "4". As a result, the evaluation of this transition yields to a symbolic state (X_2, η_2, π_2) in which $\eta_2 = \eta_1[v \leftarrow \eta_1(2 + v - \$1)]$ and we have $\eta_1(2 + v - \$1) = 2 + (1 + \#2) - 4 = \#2 - 1$ and hence the example is correctly illustrated on Fig.13.8.

Example on the middle In the second example, we illustrate the use of symbol creation. Let us suppose we start from the symbolic state $(X_0 = \emptyset, \eta_0 = [], \pi_0 = \top)$. We then consider the successive execution of two symbol creations. The first one corresponds to creating a first symbol, which we denote $\#1$ and assigning it to v. As a result, we have:

$$(\emptyset, [], \top) \xrightarrow{l:v:=\#} (\{\#1\}, [v \leftarrow \#1], \top)$$

Likewise, in the second transition, we create a new symbol $\#2$ distinct from the first and assign it to v so that:

$$(\{\#1\}, [v \leftarrow \#1], \top) \xrightarrow{l:v:=\#} (\{\#1, \#2\}, [v \leftarrow \#2], \top)$$

Example on the right The third example illustrates the use of guards. Let us suppose we start from a symbolic state (X_0, η_0, π_0) such that $\#2 \in X_0$ and $\eta_0(v) = \#2 - 1$. Then, the first transition displayed on

Fig.13.8 consists in the evaluation of a guard $(v > 5)$. Hence, we have:

$$(X_0, \eta_0, \pi_0) \xrightarrow{l:(v>5)} (X_0, \eta_0, \pi_0 \wedge (\#2 > 4))$$

With the second transition, we receive a message carrying the concrete integer value "2" and immediately after we use this value to formulate the guard $(v \leq 4 + \$1)$. Hence we have:

$$(X_0, \eta_0, \pi_0 \wedge (\#2 > 4)) \xrightarrow{l:(v\leq 4+2)} (X_0, \eta_0, \pi_0 \wedge (\#2 > 4) \wedge (\#2 \leq 5))$$

13.3.4 Symbolic actions and symbolic interactions

We can then define symbolic actions as triplets (P_-, a, P_+) such that P_- and P_+ are programs of the simple program language and a is an action (enriched with message parameters). For instance, such a symbolic action is illustrated on Fig.13.9, where we have P_- being the empty program, $a = l?message(4)$ and P_+ containing the three statements that are represented.

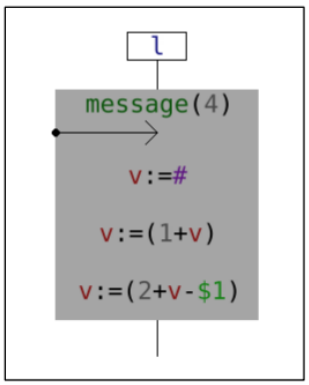


Figure 13.9: A symbolic action

Then, we can define symbolic interactions in the same manner that we defined "labelled" interactions in Chap.4 i.e. by induction on those base symbolic actions and using some constructors. In addition to the previously described constructors, we may also add a `scope` constructor for defining unique instances of variables.

The semantics of those objects can be characterized by an execution relation (similarly as labelled interaction) which can be roughly understood as follows:

$$\frac{\texttt{lab}(i) \xrightarrow{a@p} \texttt{lab}(i') \qquad (X, \eta, \pi) \xrightarrow{P_-;P_+} (X', \eta', \pi')}{(i, X, \eta, \pi) \xrightarrow{P_- a P_+ @p} (i', X', \eta', \pi')}$$ where $\texttt{lab}(i)$ is the labeled counterpart of i

Of course, in practice, this is more complicated, for two main reasons which are that: (1) we need to specify the context in which program P_+ is executed given that it might require taking values from parameters of an incoming message and (2) computing the follow-up symbolic interaction i' is not immediate, and, in particular, we might need to update parameters of received messages to reflect their evaluation when the message has been send.

Indeed, in this section, we have simply discussed the nature of symbolic interactions. A full formalisation and definition of a sound semantics may be the object of further work.

13.4 Exploration and analysis

In this section we discuss some more on the animation of symbolic interaction models in relation with the exploration of their execution trees and analysis graphs.

13.4.1 Unsatisfiable paths

The execution tree of a symbolic interaction i is a refinement of the execution tree of the corresponding labelled interaction $\texttt{lab}(i)$. Indeed, some paths that would be allowed in $\texttt{lab}(i)$ may correspond to unsatisfiable path conditions in the execution tree of i.

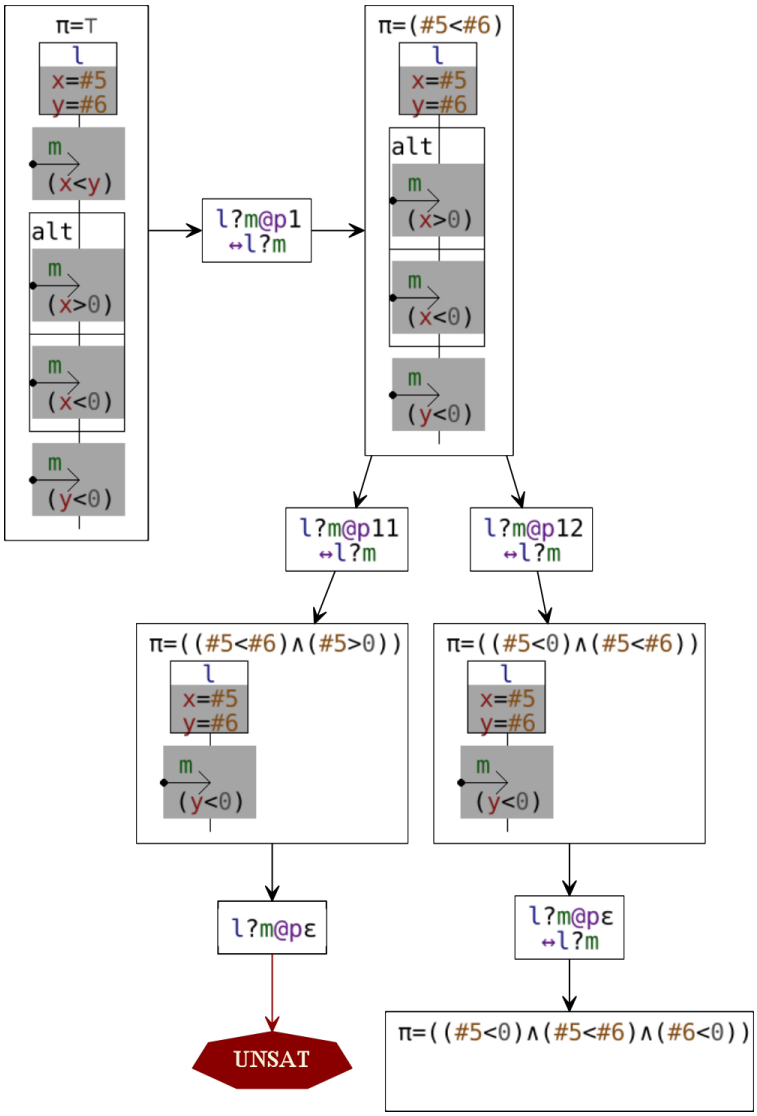


Figure 13.10: Example of an unsatisfiable path in a symbolic execution tree

We provide an example of such an unsatisfiable path on Fig.13.10. In this example, the lifeline `l` has two variables `x` and `y` of initial values #5 and #6. Then, the first action that is executed adds the condition

$\#5 < \#6$ to the path condition π. Then, two possible paths are explored depending on which branch of the alternative is taken. On the one on the left the additional condition $\#5 > 0$ is added to π whereas on the one on the right, it is $\#5 < 0$ which is added. Then, on both path we execute an action which adds the condition $\#6 < 0$. We can then see that from the left branch we have $\pi = (\#5 < \#6) \wedge (\#5 > 0) \wedge (\#6 < 0)$ which is an unsatisfiable formula. As a result, we add an UNSAT node to that path signifying that it is not feasible. On the path on the right however, we can see that the resulting path condition is satisfiable.

With this example, we have seen that unsatisfiable paths could be detected in the execution tree of a symbolic interaction i and therefore reduce its span with regards to the execution tree of the corresponding labelled interaction $\texttt{lab}(i)$. However, this notion of having unsatisfiable paths is particularly more useful when considering the analysis of multi-traces given that we may detect incoherent data in the concrete values provided as arguments of the analyzed multi-trace.

13.4.2 Message passing

As mentioned in Sec.13.3, whenever a message that is intended to carry data is send, we need to update the carried values according to the current interpretation of variables on the emitting lifeline at the moment of the emission. Let us consider for instance the example from Fig.13.11. We can see that initially, variable `x` on lifeline `l2` has a value set at $\#5$. It is then this value $\#5$ that is send with message `m` while the value of `x` on `l2` independently evolves to $\#5 - 1$. Then, it is the carried value $\#5$ in message `m` that is used when the message is received by `l1` to assign its value to the new variable `x` on `l1`. Afterwards, we have two distinct instances of variable `x`: the one on `l2` of value $\#5 - 1$ and the one on `l1` of value $2 * \#5$. Likewise, in the following step, when `l1` broadcasts the `m` message to `l2` and `l3`, the value that is carried is then $(2 * \#5) - 3$.

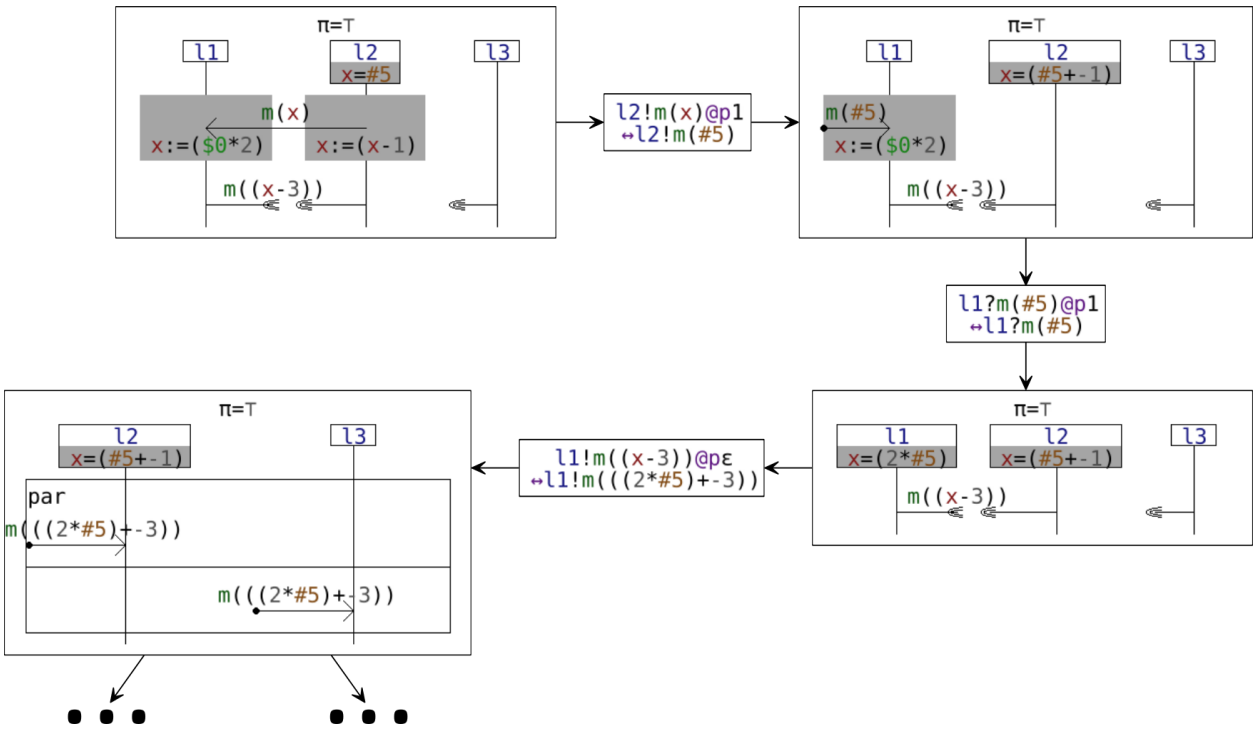


Figure 13.11: Example of message passing during execution

This mechanism of value passing ensures that, when a message is passed or broadcasted between lifelines, the values of message parameters that are originally send by the emitter are indeed the same values that are received by the receivers. With multi-trace analysis in mind, this allows the detection of foreign data being introduced, for instance, in the case of a man-in-the-middle attack, where an attacker alters information that is exchanged between parties.

13.4.3 Multi-trace analysis

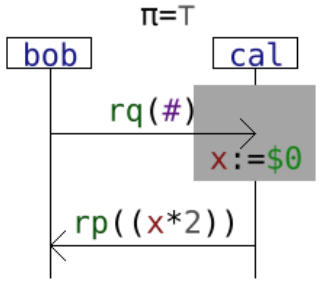
Let us consider the example given on Fig.13.12. On the left is represented a symbolic interaction which serves as a formal specification. On the right are represented three multi-traces that might be expressed by the distributed system that is modelled.

The system is composed of two lifelines `bob` who is a user and `cal` which is a calculator. `bob` can request, via the sending of a `rq` request message, that the `cal` calculator multiplies an integer value by 2 and returns the result to him via a `rp` response message.

As a result, the first multi-trace at the top right of Fig.13.12 corresponds to an accepted behavior. Indeed, here `bob` request having 4 being multiplied by 2. The `cal` indeed receives the value 4 and answers with the value 8.

However, in the second multi-trace, `cal` also receives the value 4 but answers with the value 2, having divided 4 by 2 instead of multiplying it. Hence, this multi-trace should yield a *Fail* verdict.

In the third multi-trace, even though `bob` sends the value 4, the value that is received by `cal` is 5. As a result, even though the computation is correct, given that `cal` sends 10 back, this behavior should also yield a *Fail* verdict. In practice, this could correspond to a man-in-the-middle attack, the transmitted data having been tempered with.



Accepted behavior

```
[bob] bob!rq(4);
[cal] cal?rq(4).cal!rp(8)
```

Wrong computation

```
[bob] bob!rq(4);
[cal] cal?rq(4).cal!rp(2)
```

Man-in-the-middle

```
[bob] bob!rq(4);
[cal] cal?rq(5).cal!rp(10)
```

Figure 13.12: Symbolic interaction & various conform & non-conform behaviors

Those three multi-traces can be analyzed against the symbolic interaction of Fig.13.12 so as to detect the non-conformities that we mentioned for the second and third multi-traces, as illustrated on Fig.13.13. When analyzing a concrete multi-trace, additional constraints are introduced in the path condition π so as to take into account the values of the concrete data:

- In the case of the first multi-trace, all the additional constraints are satisfiable and the path in the execution tree yields a *TooShort* verdict and hence the analysis yields *WeakPass*.

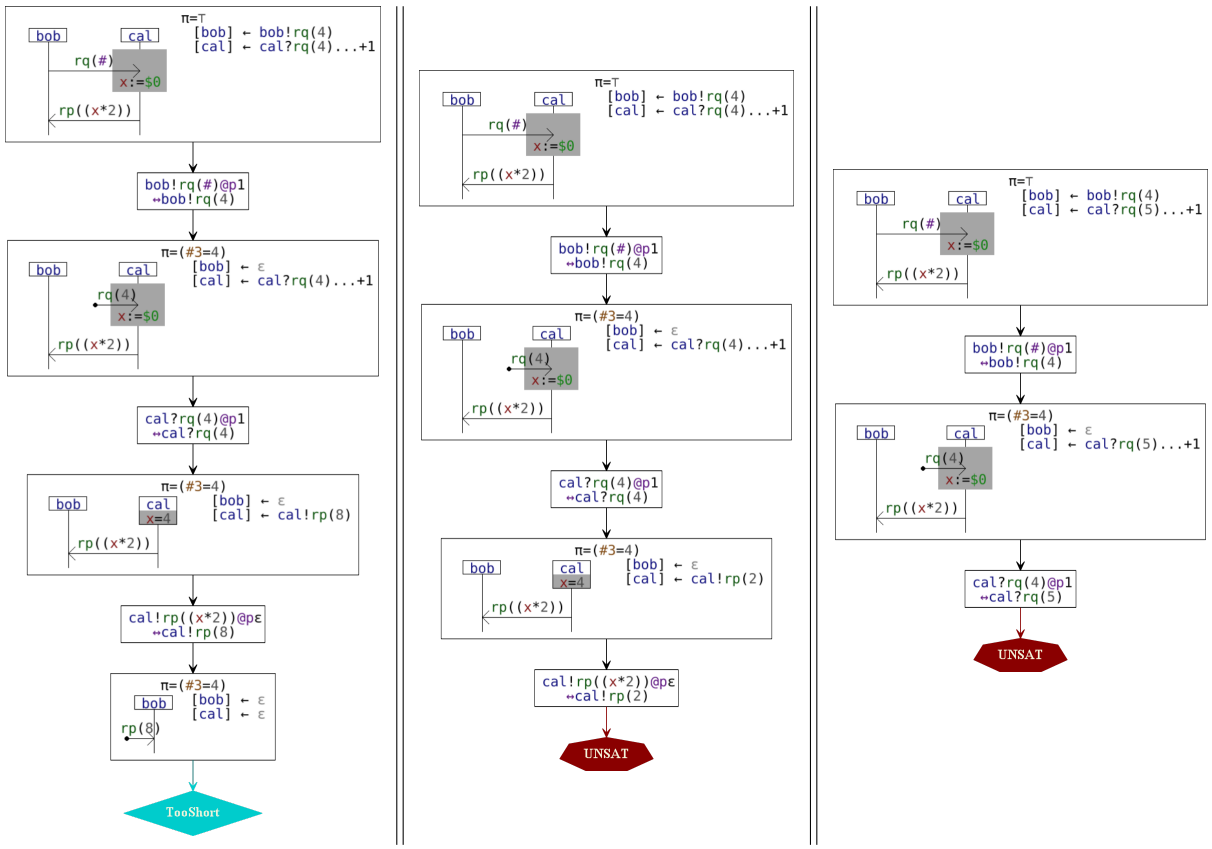


Figure 13.13: Analysis of the three multi-traces from Fig.13.12

- In the case of the second multi-trace, the concrete value 2 taken from message `rp`(2) does not satisfies $2 = 2 * x$ given $x = 4$ and hence the path yields UNSAT. The analysis therefore yields a *Fail* verdict.

- In the case of the third multi-trace, as per the message passing mechanisms previously mentioned, we have, in the second vertex of the analysis graph the reception of `rq`(4) that is expected to occur on `cal`. However, we have `cal`?`rq`(5) on the multi-trace. The constraint $4 = 5$ not being satisfiable, the path yields UNSAT and hence the analysis yields *Fail*.

As exemplified in Fig.13.13, non-conformities related to the use of data in models can be detected. If we had used labelled interactions in that case, all three concrete multi-traces would have been abstracted into the same labelled multi-trace and no non-conformities would have been detected.

Conclusion

In this section we have motivated and described an extension of the core Interaction Language to data. We define "symbolic" interactions i.e. interactions enriched with data, as a refinement of "labelled" interactions. In such models, typed variables can be created and owned by individual lifelines. The execution of individual communication actions can then be preceded or followed by the execution of a small program, consisting of assignments and guards which may change the values of variables or constraints defined on their values while constraining the execution of actions to those that satisfy those added constraints. As a result, the execution

tree of a symbolic interaction is a refinement of that of the corresponding labelled interaction in so far as it has the shape of one of its sub-trees. Likewise, the analysis of a multi-trace with data against a symbolic interaction is a refinement of the analysis of the counterpart labelled multi-trace against the counterpart labelled interaction, the symbolic analysis graph having the shape of a sub-graph of the labelled analysis graph.

Chapter 14

The HIBOUX tool

Contents

In this chapter we present an extension of the HIBOU tool which we have introduced in Chap.12. This extension HIBOUX (for HIBOU with symbolic eXecution) allows the exploration of execution trees associated to interaction models enriched with data as well as the analysis of multi-traces in which messages may carry concrete data against such models. To do so, we use symbolic execution to execute those enriched interaction models.

The plan of this chapter is as follows:

- in Sec.14.1 we quickly present the software and relevant parts of its user interface,
- in Sec.14.2 we describe the entry language of the tool,
- and in Sec.14.3, we quickly mention an industrial use case on which HIBOUX has been used, as part of the FUI DisTA project.

14.1 Overview

HIBOUX (for HIBOU with symbolic eXecution) provides utilities for drawing and manipulating symbolic interaction models, exploring their semantics and analysing multi-traces enriched with concrete data with regards to formal specifications written as symbolic interaction models.

HIBOUX is also implemented in Rust [7]. Its full code is available in [90]. It contains (for version 0.1.1) 13657 lines in 120 files among which 10532 lines of code. The version of HIBOUX which we describe in this chapter is version 0.1.1.

HIBOUX takes the form of an executable which offers a user Command Line Interface (CLI). Precompiled binaries are available for Windows and Linux in [90].

The use of HIBOUX is associated to two kinds of input files:

- "hiboux specification files", in which symbolic interaction models are encoded and which correspond to the ".hxsf" filename extension
- "hiboux trace file", in which traces or multi-traces with concrete data are encoded and which correspond to the ".hxtf" filename extension

Then, the `hiboux` executable provides a CLI which includes (among others) the following commands:

- "`hiboux draw <.hxsf file>`" draws a symbolic interaction
- "`hiboux explore` <.hxsf file>" explores the semantics of a symbolic interaction
- "`hiboux analyze` <.hxsf file> <.hxtf file>" analyzes a multi-trace with data against a symbolic interaction

14.2 Entry language

HIBOUX accepts ".hxsf" and ".hxtf" files for respectively specifying symbolic interactions and multi-traces with concrete data. Those two file types are similar to the ".hsf" and ".htf" which we have seen in Chap.12.

In the following we provide some more details on the entry language of HIBOUX.

14.2.1 Header options & declarations

The option sections `@explore_option` and `@analyze_option` that can be defined in the headers of ".hxsf" files are similar to those that we have seen in Chap.14 for the ".hsf" files of HIBOU.

Similarly to ".hsf" files, we declare in the headers of ".hxsf" files various components that participate in the definition of the interaction term. On Fig.14.1 we provide an example. As in Chap.12, we declare lifelines under a `@lifeline` section and messages under a `@message` section.

However, given that messages can have typed parameters, we declare the types (and hence number) of those parameters in between parentheses as illustrated on Fig.14.1. The types supported by HIBOUX include `Bool` and `Integer` as illustrated on Fig.14.1. The parameters of a message constitute an ordered list, and, individual parameters can be referred to by their index on that list. For instance $0 may refer to the parameter of index 0.

Profiles of variables can be declares under a dedicated `@variable` section, as illustrated in Fig.14.1. A variable profile declares the type of variables that may be instantiated on some lifelines during execution or before the start of the execution, in a dedicated initialization stage. A variable profile may correspond to instances of variables that may exist on any lifeline. Let us recall that in HIBOUX there are no global variables. All variables are locally defined and we may create new instances of variables during execution.

```
@message{
    publish  (Integer,Integer,Integer);
    puback   (Integer,Bool);
    transmit (Integer)
}
@variable{
    mkey : Integer;
    key  : Integer;
    data : Integer;
    gdat : Integer;
    gkey : Integer;
    mid  : Integer;
}
@lifeline{
    pub1;broker;sub1;sub2
}
@init{
    broker.mkey = #;
    pub1.key    = #
}
```

Figure 14.1: Example header declaration in a ".hxsf" file

```
@loopH(
    @scope{gkey,gdat,mid}(
        @seq(
            pub1
              -- publish(#,#,key)
              -> broker{mid:=$0;gdat:=$1;gkey:=$2},
            @alt(
                @seq(
                    [(gkey=mkey)]broker
                      -- puback(mid,⊤)
                      -> pub1,
                    broker
                      -- transmit(gdat)
                      -> (sub1{data:=$0}, sub2{data:=$0})
                ),
                [(gkey!=mkey)]broker
                  -- puback(mid,⊥)
                  -> pub1
            )
        )
    )
)
```

Figure 14.2: Encoding of a symbolic interaction (continuation of Fig.14.1)

The declaration of lifelines, messages and variables are all independent from one another. A declared lifeline may emit or receive any declared message and own an instance of any declared variable.

In addition, we can initialize variables on some lifelines and set values to them. This can be done under the dedicated `@init` section, as illustrated on Fig.14.1.

14.2.2 Specifying symbolic interactions

Symbolic interactions are encoded in ".hxsf" files in the same manner than labelled interactions are encoded in ".hsf" files. We illustrate this encoding with the example interaction from Fig.14.3 which encoding is presented in Fig.14.1 and on Fig.14.2.

We can remark that the overall structure of interaction terms is identical. The main difference comes with the definition of actions in which:

- the associated message may carry data and:
  - in the case of receptions this data may either be a raw value or a symbol # to signify random input data
  - in the case of emissions this data may be a term formulated on variables carried by the lifeline
- we may add some programs written in the language evoked in Chap.13 as either or both a preamble or postamble.
- there is an additional `scope` constructor which serves as making variables unique and have a local scope w.r.t. a certain sub-behavior.

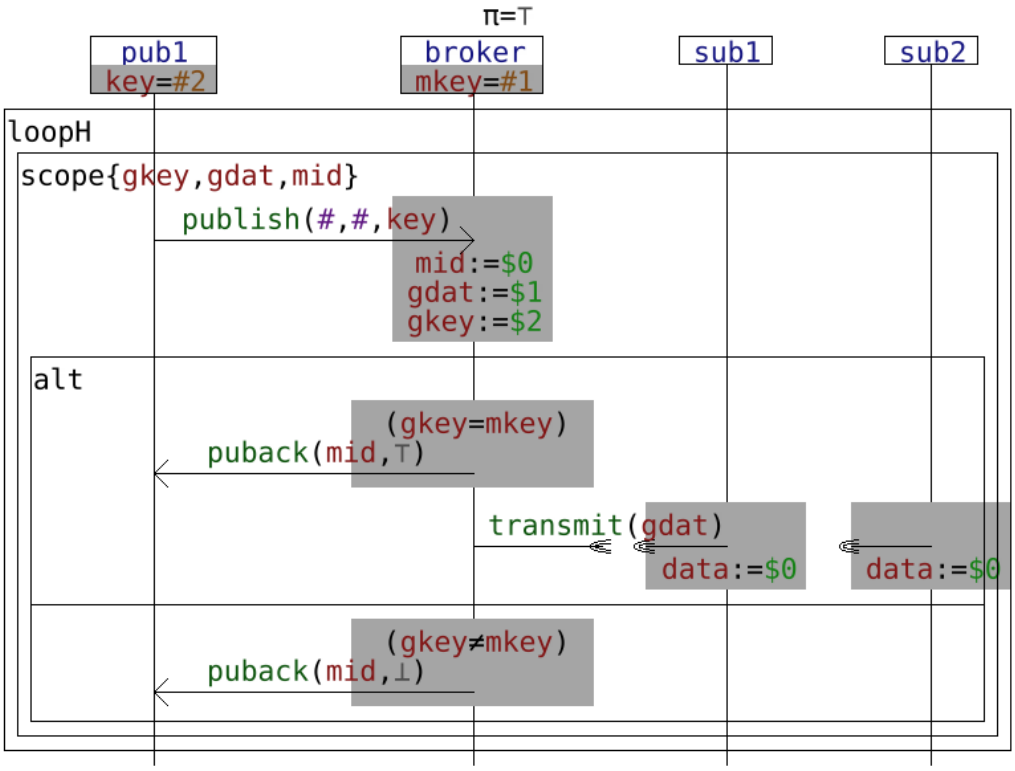


Figure 14.3: Symbolic interaction specified by the ".hxsf" file from Fig.14.1 and Fig.14.2

In the toy example from Fig.14.3, we describe a DS composed of four sub-systems `pub1` which publishes new information, `broker` which receives and may transmit this information and `sub1` and `sub2` which receive the transmitted information.

The sending of new information by `pub1` can be repeated; in the model we use a `loopH` constructor. Concretely, `pub1` sends a `publish` message to `broker` with three integer arguments which are a random message identifier ($0), random data ($1) and its authentication key which serves as identifying it as an certified published or information.

When the `broker` receives the message, it creates new temporary variables mid, gdat and gkey to store the received message identifier, data ("got data") and key ("got key"). Those variables are uniquely defined for each instantiation of the `loopH` thanks to the use of the `scope` constructor. For instance, if the `pub1` sends two different `publish` messages and the `broker` receives them, it will create a `mid_1` variable for the message id of the first message and a `mid_2` variable for the second.

In a given instance of the loop, after having received the `publish` message, the `broker` compares the key that it received and stored in its temporary `gkey` variable to its master key `mkey`. Then:

- If they are equal then the publication is authenticated and the `broker` confirms it to `pub1` by sending `puback` with the corresponding message identifier `mid` and a Boolean ⊤ to signify success. Then, the received data, stored in the temporary `gdat` variable is transmit to the subscribers `sub1` and `sub2` via the `transmit` message.

- If they are different then the `broker` refuses to transmit the data and informs the publisher `pub1` by

answering with a puback message that carries the corresponding message identifier mid and a Boolean $\perp$ to signify failure.

14.2.3 Specifying multi-traces with concrete data

Multi-traces with concrete data are encoded in ".hxtf" files for use with HIBOUX. This encoding is similar to the one used to encode labelled multi-traces in ".htf" files presented in Chap.12.

We provide an example on Fig.14.4. It suffices to add the raw values of concrete parameters in between parenthesis after the name of the corresponding message.

```
{
  [pub1] pub1!publish(1,999,456);
  [broker] broker?publish(1,999,456).broker!puback(1,⊤);
  [sub1,sub2]
}
```

Figure 14.4: Multi-trace with concrete data in a ".hxtf" file

14.3 Proof of concept use case

The thesis has been carried-out as part of the DisTA (Distributed Test Automation) project, financed as a Fond Unique Interministériel (FUI) by the French Ministry of the Economy and Finance. This project is a collaborative R&D project between several academic and industrial actors which are represented on Fig.14.5.

Figure 14.5: The DisTA project

As part of the thesis we worked closely with the CEA so as to implement the HIBOUX tool. As a proof of concept, we have also participated in a use case with Thales. We used HIBOUX to test log files extracted from a prototype distributed system made by Thales.

On Fig.14.7 is represented the interaction that models the prototype. It consists of three moving vehicles hkb1, hkb2 and hkb3 which regularly transmit their positions (x and y coordinates) via a position message to a station sta1. This station keeps track of the last known positions of each vehicle via the h1x, h1y,

h2x, h2y, h3x and h3y variables. Then, an operator ope1 may send a request to the sta1 station to put to sleep for a given amount of time any vehicle that is within a specific square area. This square area is delimited by four coordinates. Hence the message zsleep contains four arguments. When the station sta1 receives it, it stores the received values into temporary xmin, ymin, xmax, ymax and duration variables. Then depending on whether or not each of the three vehicles are in the designated area (for instance we must have $(\texttt{xmin} \leq \texttt{h1x} \leq \texttt{xmax}) \wedge (\texttt{ymin} \leq \texttt{h1y} \leq \texttt{ymax})$ satisfied for vehicle hkb1 to be in the area), a sleep command hsleep, with the corresponding duration is transmitted to those that are. Any vehicle that receives one such sleep command request confirmation to the operator ope1 before entering into sleep mode.

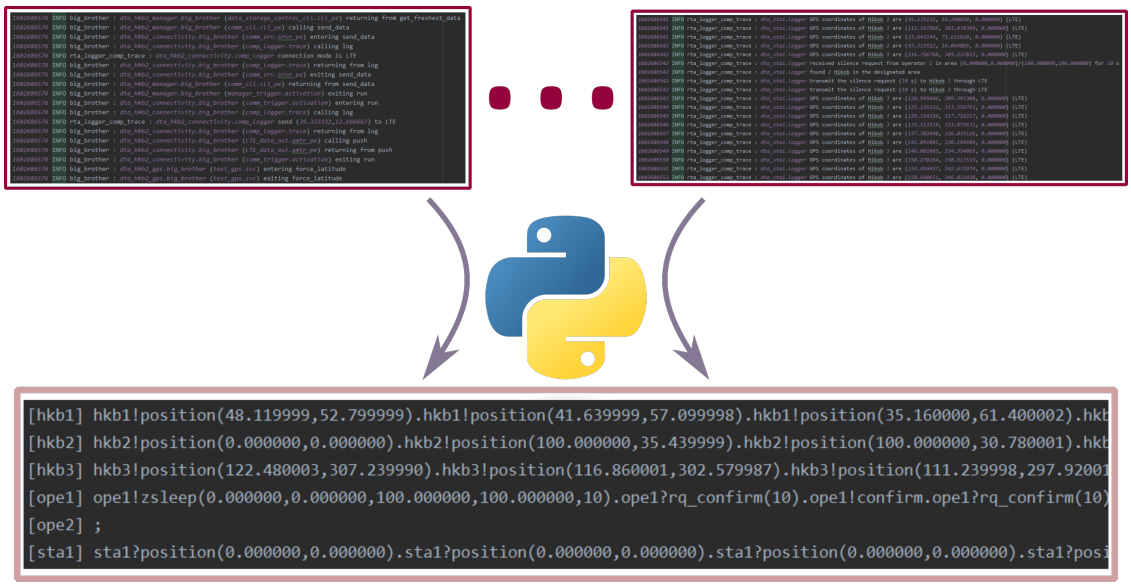


Figure 14.6: Parsing log files collected from the Thales prototype into multi-traces with data

In order to analyze the outputs of this distributed system with HIBOUX, we have collected log files produced by each sub-system. Parts of two such log files are represented at the top of Fig.14.6 (the one on the left corresponds to hkb2 and the one on the right corresponds to sta1). Those log files (".log" format) consists in several thousands lines containing logging information, not all of which is pertinent to our analysis. In order to transform those files into a multi-trace, we have written a parser in Python. Each log file, corresponding to an individual lifeline is parsed and transformed into a local trace component. The sum of those local trace component then is reconstituted into a multi-trace as shown on Fig.14.7.

Finally, those multi-traces were analyzed against the interaction model from Fig.14.7. In total we have analyzed several multi-traces both accepted and with non-conformities (introduced willfully) and which sizes (in total number of actions) varied between 408 and 572. As for the time required for the analysis, it ranged from 16.67 seconds to 91.05 seconds on a Dell Latitude 7480 with a 2 cores 2.00GHz Intel i5-6360U processor. Let us however remark, that more recent works on HIBOU such as the checking of local frontiers have not been ported to HIBOUX at the time of those experiments.

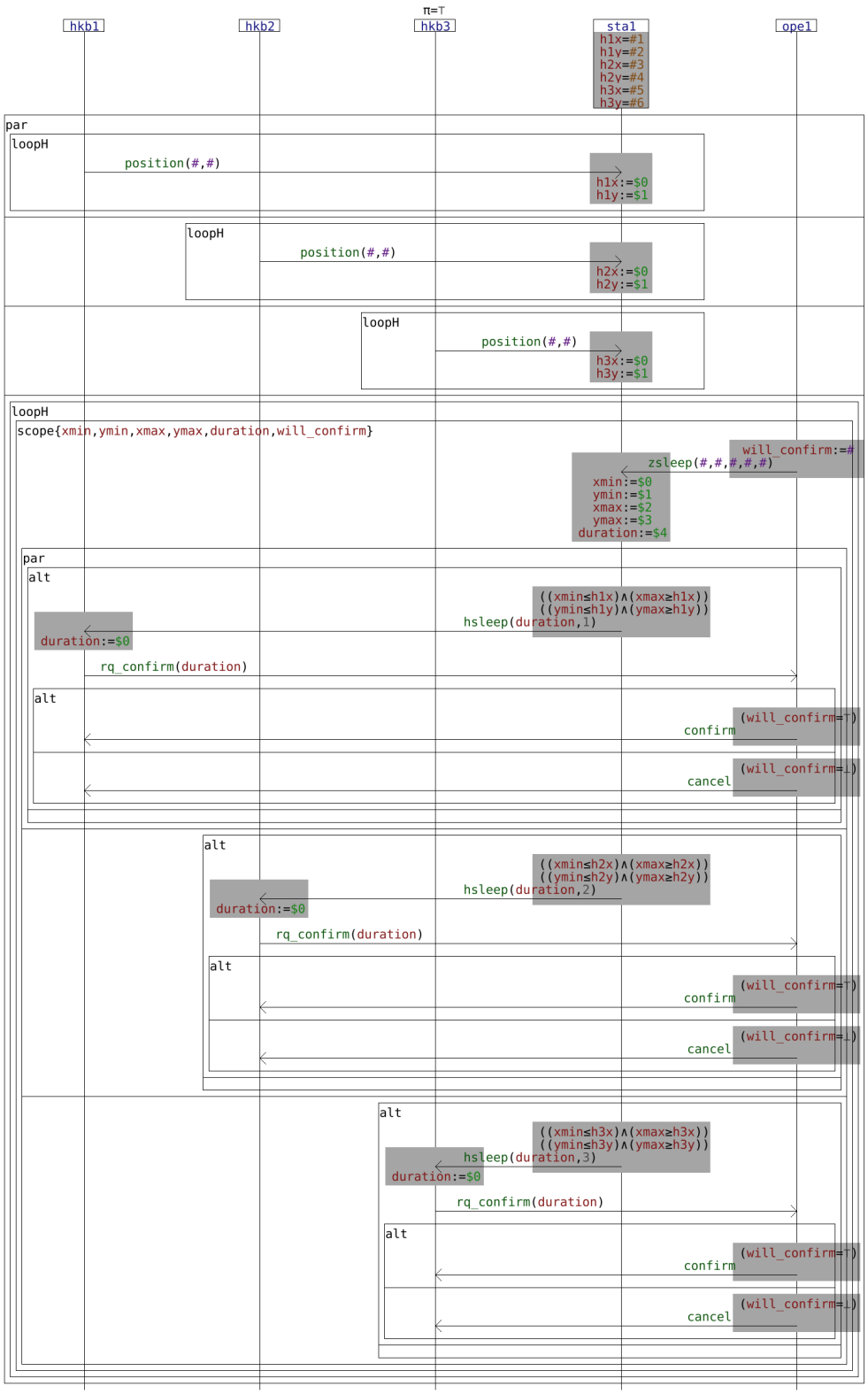


Figure 14.7: Symbolic interaction model for the Thales prototype use case

Conclusion

In this chapter we have presented the HIBOUX tool, which extends HIBOU by handling data in interaction models thanks to symbolic execution. We have notably used this tool as part of a prototype use case with an industrial partner.

Chapter 15

Conclusion

15.1 Summary

In this thesis we have defined an Interaction Language (IL) for modelling the behavior of Distributed Systems (DSs). This expressive language, formulated as a term algebra, includes operators for strict and weak sequencing, alternative and parallel composition and four kinds of loops so as to express nuances in the repetition of behaviors. The mathematical foundation of the meaning of that language is a trace semantics formulated in denotational style as a homomorphism between the term algebra of interactions and an algebra of sets of traces. Each interaction model is associated to a set of sequences of communication actions called a trace. This formulation of the semantics serves as a reference and allows for proving interesting properties. It is notably used to justify the definition of normal forms of interaction terms (computed via term rewriting) which are unique, compact and minimal represent of classes of equivalence containing syntactically distinct but semantically equivalent interaction terms.

However the denotational formulation of the semantics is cumbersome for use in some applications in Formal Verification (FV). As a consequence, we define a structural operational semantics for our IL which is inspired by that which can be found in the domain of process calculus. We then prove that this operational formulation is equivalent to the denotational formulation of the trace semantics of interactions, which constitutes, as mentioned before, a mathematical foundation.

The operational semantics then enables the definition of various algorithms for analyzing partial and distributed observations of the executions of DSs against formal specifications written as interaction models. The distributed nature of the observation, with each sub-system producing their own logs, combined with the absence of synchronization mechanism (or e.g. global clock), which prevents the global reordering of events, makes so that the objects to analyze are sets of local traces that we call multi-traces. The partial nature of observation makes so that we may need to analyze prefixes or slices (i.e. "sub-words") of multi-traces in which, on each component traces, events may be missing at the beginning (due to the observation having started too late) or at the end (due to the observation having ended too early). The various algorithms which

we have defined for identifying and discriminating between those diverse cases of partial observation either rely on simulation steps to complete missing events corresponding to unobserved behavior or on hiding steps. Hiding steps consist in restricting the analysis of the multi-trace to a smaller interaction model. Instead of continuing using the whole interaction specification, once a sub-system stops from being observed, we rather use a sub-specification that only concerns the sub-systems that are still observed.

Using an algorithmization of the operational semantics called the "execution semantics", which we have proven to be correct, we implemented the HIBOU tool which features most if not all the theoretical results developed during the thesis. This includes the ability to specify, manipulate and draw interaction models, compute their normal forms, explore their semantics and analyze multi-traces against them using any of the multi-trace analysis algorithms which we defined.

In addition, in order to meet challenges presented by an industrial use-case, we have enriched our IL with data via the implementation of the HIBOUX (HIBOU with symbolic eXecution) which allows the execution of interaction models enriched with data in the form of variables, message parameters, guards and so on. HIBOUX uses symbolic execution so as to treat in an efficient manner the analysis of multi-traces that contains typed data as parameters of send and received messages. We have used this tool in order to test outputs (sets of logs) of a prototype implementation provided by an industrial partner as part of the FUI DisTA project.

15.2 Perspectives

The work presented in this thesis could be extended in various ways. It can also be related to more distant research. In this section, we discuss research directions that could extend or benefit from our work.

15.2.1 Formalizing value passing & symbolic interactions

With the presentation of HIBOUX we have touched upon the notion of value passing because the data in a message that is send must be the same when that message is received. In HIBOUX this is taken care of by rewriting the symbolic interaction model so as to write the values computed when the message is send onto the message arguments of the corresponding receptions. However we have not formalized those mechanisms of value passing. Value passing can be defined and formalized independently of the addition of data or the use of symbolic execution. As a result, a first step towards the formalization of the theory behind HIBOUX could be to formalize an Interaction Language with value passing.

Then, of course, a follow-up research would be to propose a full formalisation of a symbolic interaction language and its semantics.

15.2.2 Towards testing & extended formal verification

In this thesis, we have presented various algorithms as solutions to the membership problems associated with specific semantics. Although the analysis of a multi-trace shares similarities with the notion of a test,

we have not explicitly touched upon this subject on the thesis. This can be explained by the absence of a notion of conformance relation associated with interaction models and multi-traces, which naturally calls for further work. Additionally, we have not yet discussed methods such as the automatic generation of test cases which are associated to partices of model-based testing.

The application of other Formal Verification techniques to our approach could also be investigated. This notably includes monitoring or online testing. In the context of this thesis, with multi-trace analysis, we have carried out a form of passive testing, in which a pre-existing finite multi-trace, that is collected after the execution of the DS, is analysed. With monitoring, the analysis occurs while the system is being executed, with the monitor listening for the occurrence of new events so as to feed the execution of the formal specification model which intends to reflect that of the concrete system implementation. This approach of monitoring aims at making sure of the smooth running of deployed implementations. With online-testing, the monitor is replaced by a tester, which also listens for the occurrence of events, but is also allowed to interact with the system and stimulate its execution, for instance, so as to force the expression of a certain behavior. Those two approaches of monitoring and online-testing could be the object of further work.

15.2.3 Enriching the interaction language

It is always possible to increase the expressivity of a language. This is therefore also the case for our Interaction Language. As many different and sometimes incompatible extensions can be investigated, further work related to the enrichment of the language with new constructs may be various and heterogeneous in nature, maturity and potential interest. In the following we present some such extensions.

We have seen in Chap.11, with the co-region constructs, that other operators could be added to the language for a gain in expressivity. In the particular case of co-regions, it would be interesting to explore their inclusion into the core language. Moreover, the nature of co-regions as configurable scheduling operators makes so that we could define an entirely new family of loops based around co-regions. In addition, given that co-regions may coincide with both weak sequencing and interleaving, we could define a formalism in which there are only co-regions. In this formalism, the two aforementioned operators would be included as edge cases of the co-region operators.

In the language presented in this thesis, we have define loops as Kleene closures, which allow repetitions up to any arbitrary high number of times. We could extend the IL so as to enable the restriction of loops in several manners:

- either by specifying a maximum number of repetitions, a minimum number of repetitions or both
- or by defining "`break`" and "`continue`" constructs attached to the evaluation of actions and which would destroy the current instance and (in the case of break) the initial loop when evaluated.

Another interesting extension to the IL would be the addition of modality. In UML-SD, modality takes the form of the *assert* and *negate* combined fragments which allow the specification of valid and invalid traces. However, as explained in [63, 100], this manner of defining modality comes with many issues. In

[63] the notion of hot (universal) and cold (existential) modalities associated to individual actions is rather proposed.

The language which we have presented has a static structure by nature. Indeed, any term is defined up to a signature (L, M) which determines which messages and lifelines are defined. A possible extension to the language would be to allow the creation and destruction of lifelines associated to the evaluation of some actions. Lifeline "templates" could be defined so as to create new lifelines during execution, that are expected to express specific behaviors.

Another possible extension to the IL would be that of considering groups of lifelines which contents could vary in time. This can be linked to the previous point, with, for instance, newly created lifelines becoming members of a certain group. Then, we could for instance define broadcasts that target members of these groups. Instead of having an alternative to cover all cases of which lifelines may or may not receive the broadcast, this would then be covered by the membership of the lifeline to a certain group.

Finally, in our IL we have only considered non-deterministic choice for the taking of alternative branches or the instantiation of loops. This non-determinism however is in fact rather linked to the choice of which action is fired rather than to the specific alt or $loop_k$ constructors. A possible extension to the language would then be to consider probabilistic interactions (by analogy to probabilistic automata) in which the execution of specific actions can be associated with probabilities.

15.2.4 Towards leaning interaction models

In this thesis, we have seen that we could analyse multi-traces collected from executions of a Distributed System against a specification of that DS written as an interaction model. However, in practice, it might be so that, on the one hand, we have no such interaction model, and, on the other hand, we might have a large dataset of collected multi-traces. Then, reconstructing an interaction model from this set of multi-traces would consists in a sort of reverse analysis. Doing so would then allow the automatic generation of models which could, for instance, help the understanding of complex systems which were not designed but rather emerged from the cooperation of various sub-systems.

An ideal tool for this kind of endeavour would be machine learning as it has already been used in the literature for learning automata-like models from traces [110]. The works presented in this thesis would help two aspects of learning interaction models:

- the multi-trace analysis algorithms that we have defined would constitute a criteria for whether or not a generated model is valid
- with the normalization of interaction terms, a machine learning algorihtm would only need to search for terms in the space of normal forms instead of the space of all interaction terms

On the first point, we could notably define a dedicated multi-trace analysis algorithm that is able to analyse several multi-traces at once and stops if any one of them is not accepted.

Appendices

Appendix A

Synthèse en français

Le caractère distribué de certains systèmes complexifie les problématiques liées à leur modélisation et à la vérification de leurs comportements. Dans le cadre de cette thèse nous nous intéressons à un type de formalisme particulier: les interactions, afin de modéliser le comportement de systèmes distribués. On y spécifie les échanges asynchrones de messages pouvant être observés aux interfaces des différents sous-systèmes d'un système distribué. Nous pouvons associer aux interactions une notation graphique intuitive qui permet une compréhension et une prise en main facile. Dans cette thèse nous formalisons ce type de modèles avec un Langage d'Interactions (LI) défini sous la forme d'une algèbre de termes. Ces termes sont construits à partir d'interactions atomiques qui peuvent correspondre à un comportement vide ou l'expression d'évènements de communication atomiques i.e. émissions ou réceptions de messages. Des opérateurs permettent la construction de termes plus complexes par composition. Un opérateur de séquencement strict est utilisé pour signifier qu'un évènement doit avoir lieu après un autre. Un opérateur de composition parallèle permet à deux évènements d'avoir lieu dans n'importe quel ordre. Le séquencement faible, qui est particulier aux systèmes distribués, consiste à ce que les évènements se produisent séquentiellement s'ils ont lieu sur le même sous-système mais en parallèle dans le cas contraire. La composition alternative permet de spécifier un choix non-déterministe entre deux comportements. Différentes nuances dans la répétitions de comportements peuvent être exprimés en utilisant quatre opérateurs de boucle distincts.

Nous définissons ensuite une sémantique de trace, associant à chaque terme d'interaction un ensemble de séquences d'évènements qui représentent les comportements pouvant être exprimés par le système distribué qui est modélisé. Cette sémantique est formulée dans un premier temps de manière dénotationnelle. La sémantique d'interactions complexes est définie par composition à partir de celles de ses sous-termes, composées à l'aide d'opérateurs algébriques sur les ensembles de traces. Cette formulation prenant la forme d'un morphisme d'algèbre nous pouvons ensuite utiliser les propriétés algébriques des opérateurs sur les ensembles de traces (associativité, commutativité, etc.) pour obtenir des équations reliant des termes d'interactions ayant la même sémantique. Grâce aux techniques de réécriture nous pouvons ensuite définir des formes normales de terme d'interaction. Ces formes normales correspondent à des termes qui expriment d'une manière

la plus compacte possible, un certain comportement.

Dans un deuxième temps, nous proposons une formulation opérationnelle de la sémantique de trace, qui permet de rendre les modèles d'interaction exécutables et ouvre la voie à des applications plus poussées en vérification. Cette formulation opérationnelle consiste en la détermination de transformations qui, en partant d'une interaction initiale, et via l'expression d'un certain évènement atomique, aboutie à une nouvelle interaction qui spécifie exactement toutes les continuations des comportements exprimables par la première et qui commencent par l'évènement en question. Ces deux formulations de la sémantique des interactions sont prouvées équivalentes, la preuve étant supportée par le prouveur de théorème Coq.

Les exécutions d'un système distribué peuvent être observées au travers de logs des évènements de communication collectés localement. Sans horloge globale il n'est pas possible de réordonner ces évènements globalement. Analyser une exécution revient donc à analyser un ensemble de traces, chacune correspondant à une observation locale, qu'on appelle une multi-trace. De plus, sur chaque composante locale, il se peut que l'observation ait commencé trop tard ou ait cessé trop tôt. Ainsi, une multi-trace peut correspondre à une observation partielle d'une exécution. Tirant parti de la formulation opérationnelle ainsi que de propriétés et transformations prouvées par rapport à la formulation dénotationnelle nous proposons des algorithmes d'analyse permettant d'identifier une multi-trace comme étant une observation d'un comportement spécifié par une interaction.

Notre approche a été implémentée au sein d'un outil appelé HIBOU qui permet de spécifier et dessiner des interactions, d'explorer leur sémantiques, de calculer des formes normales ou d'analyser des multi-traces. Nous avons étendu notre LI pour inclure des données sous la forme de variables définies localement. Des gardes, expressions booléennes sur les variables, peuvent conditionner l'exécution d'actions et les messages peuvent porter des données exprimées à l'aide des variables. L'extension aux données a été implémentée en utilisant les techniques d'exécution symbolique. Cet outil étendu a été utilisé pour un cas d'usage industriel dans le cadre du projet FUI DisTA.

Notations

Trace Semantic Domain (basics)	
L	a set of lifelines
l	a lifeline l element of L i.e. $l \in L$
M	a set of messages
m	a message m element of M i.e. $m \in M$
Ω	a signature $\Omega = (L, M)$
!	symbol for discrete emission
?	symbol for discrete reception
Δ	either of ! or ?
$\mathbb{A}_\Omega$	set of all actions $\mathbb{A}_\Omega = L \times \{!, ?\} \times M$ up to signature Ω
a	an action a element of $\mathbb{A}_\Omega$ i.e. $a \in \mathbb{A}_\Omega$
$\theta(a)$	lifeline (in L) on which action a occurs i.e. a of the form $\theta(a)\Delta m$
ϵ	empty word (in general) or empty trace (empty word of $\mathbb{T}_\Omega$) or empty position (empty word of $\{1, 2\}^*$), etc.
.	concatenation law on words
$\mathbb{T}_\Omega$	set of all (global) traces $\mathbb{T}_\Omega = \mathbb{A}_\Omega^*$ up to signature Ω
t or ς	we may denote by t or ς traces i.e. elements of $\mathbb{T}_\Omega$
$\lvert t \rvert$	length of a trace t
Trace Semantic Domain (algebraic operators)	
$t \between l$	predicate stating that a trace t has some conflict w.r.t. a lifeline l i.e. contains actions occurring on l
$t_1; t_2$	strict sequencing of two traces t_1 and t_2
$T_1; T_2$	strict sequencing of two sets of traces T_1 and T_2
$t_1 ;_\between t_2$	weak sequencing of two traces t_1 and t_2
$T_1 ;_\between T_2$	weak sequencing of two sets of traces T_1 and T_2

$t_1 \| \| t_2$	interleaving of two traces t_1 and t_2
$T_1 \| \| T_2$	interleaving of two sets of traces T_1 and T_2
$\diamond$	any scheduling operator i.e. any of either ; or $;_*$ or $\|\|$
$T^{;*}$	the strict Kleene closure of set of traces T
$T^{;_**}$	the weak Kleene closure of set of traces T
$T^{\|\|*}$	the interleaving Kleene closure of set of traces T
$t_1 ;_*^{\curvearrowleft} t_2$	the restricted weak sequencing of two traces t_1 and t_2
$T_1 ;_*^{\curvearrowleft} T_2$	the restricted weak sequencing of traces T_1 and T_2
$T^{;_*^{\curvearrowright}*}$	the weak Head-First closure of set of traces T
Interaction syntax & Denotational Semantics	
$\mathbb{I}_\Omega$	set of all interactions up to a signature Ω
i	an interaction i element of $\mathbb{I}_\Omega$ i.e. $i \in \mathbb{I}_\Omega$
$\sigma_d(i)$	set of traces accepted by interaction i according to the denotational semantics σ_d
Equational Logic	
$\mathcal{F} = \bigcup_{n\in\mathbb{N}} \mathcal{F}_n$	a set of operation symbols of arities 0 (constants) or more ($n > 0$) in particular it may refer to operation symbols of our interaction language
$f \in \mathcal{F}_n$	an operation symbol of arity n
$\mathcal{X}$	a set of variables
$\mathcal{T}_\mathcal{F}(\mathcal{X})$	the free term algebra associated to $\mathcal{F}$ and $\mathcal{X}$
$\mathcal{T}_\mathcal{F}$	the ground term algebra associated to $\mathcal{F}$
$t \in \mathcal{T}_\mathcal{F}(\mathcal{X})$	a term of the term algebra
$pos(t)$	set of positions of the nodes of term t seen as a tree
$p \in pos(t)$	the position of a node in the tree associated to t
$t_{\|p}$	the sub-term of t at position p
$t[s]_p$	replacing the sub-term at position p in t by s
$\mathcal{A} = (\mathbf{A}, \mathcal{F}^\mathcal{A})$	a $\mathcal{F}$-algebra of carrier $\mathbf{A}$ in particular it may refer to $\mathcal{A} = (\mathcal{P}(\mathbb{T}_\Omega), \mathcal{F}^\mathcal{A})$ the $\mathcal{F}$-algebra of sets of traces
$f^\mathcal{A} \in \mathcal{F}^\mathcal{A}$	the interpretation of a symbol $f \in \mathcal{F}$ in the $\mathcal{F}$-algebra $\mathcal{A}$
ρ	a mapping $\rho : \mathcal{X} \to \mathbf{A}$
$\overline{\rho}$	an environment, extending a mapping ρ to terms as a homomorphism between $\mathcal{T}_\mathcal{F}(\mathcal{X})$ and $\mathcal{A}$

ϕ	a substitution $\phi : \mathcal{T}_{\mathcal{F}}(\mathcal{X}) \to \mathcal{T}_{\mathcal{F}}(\mathcal{X})$, replacing within a term all occurrences of some variables by some corresponding terms
$x \approx y$	an equation, relating two terms of a term algebra
E	a set of equations on terms, called an axiom system
$\approx_E$	congruence relation on terms induced by E through the rules of equational logic
$=_{\mathcal{A}}$	congruence relation on terms induced by $\mathcal{A}$ through semantic equivalence
	Term Rewriting
$l \rightsquigarrow r$	a rewrite rule
R	a Term Rewriting System i.e. a set of rewrite rules
$\to_R$	the one-step rewrite relation induced by R
$E >$	an Ordered Rewriting System i.e. an axiom system E and a rewrite ordering $>$
$\to_{E>}$	the one-step rewrite relation induced by $E >$
T	an equational theory i.e. another name (in this context) for an axiom system
R/T	a Class Rewriting System i.e. a TRS R and an equational theory T
$\to_{R/T}$	the one-step rewrite relation induced by R/T
$\prec_{\mathbb{I}}$	a total rewrite ordering on (ground) interaction terms
$R^1(i)$	normal form modulo AC of interaction i w.r.t. phase 1 of the process
$R^2(i)$	normal form modulo AC of interaction i w.r.t. phase 2 of the process
$R_{\prec}(i)$	normal form of interaction i w.r.t. the total order $\prec$ and rules of AC
$R(i)$	normal form of interaction i with $R(i) = R_{\prec}(R^2(R^1(i)))$
	Operational Semantics
$i\downarrow$	predicate stating that an interaction i terminates i.e. accepts the empty trace
$i \downarrow^{\divideontimes} l$	predicate stating that an interaction i evades a lifeline l i.e. accepts a trace t with no conflict w.r.t. l i.e. such that $\neg(t \divideontimes l)$
$i \overset{l}{\divideontimes\!\!\to} i'$	predicate stating that i' is the unique interaction that exactly accepts traces $t \in \sigma_d(i)$ s.t. $\neg(t \divideontimes l)$ of which there is at least one
$i \xrightarrow{a} i'$	predicate stating that action $a \in \mathbb{A}_\Omega$ found at a certain leaf of $i \in \mathbb{I}_\Omega$ is immediately executable and that what remains to be executed is specified by interaction i'
$\sigma_o(i)$	set of traces accepted by interaction i according to the operational semantics σ_o
	Execution Semantics
p	position i.e. word of $\{1,2\}^*$ with ϵ the empty position
$pos(i)$	set of the positions of the nodes of interaction i seen as a binary tree

$i_{\|p}$	sub-interaction of i at position $p \in pos(i)$
$\texttt{prn}(i,l)$	unique interaction such that $i \overset{l}{\nrightarrow} \texttt{prn}(i,l)$
$\texttt{frt}(i)$	positions of all immediately executable actions in i (frontier of execution)
$\texttt{exe}(i,p)$	follow-up interaction that specifies what remains to be executed after the execution of action $i_{\|p}$ at position $p \in \texttt{frt}(i)$
$i \xrightarrow{a@p} i'$	short notation for $p \in \texttt{frt}(i)$ and $i_{\|p} = a$ and $\texttt{exe}(i,p) = i'$
$empty(i)$	is the set $\{\epsilon\}$ if $i \downarrow$ and is the set $\emptyset$ if $i \not\downarrow$
$\sigma_e(i)$	set of traces accepted by interaction i according to the execution semantics σ_e
$\sigma_n(i)$	normalizing execution semantics i.e. variant of σ_e in which we normalize intermediate terms using rewriting
$\texttt{prn}_{\approx}(i,l)$	variant of "prn" with on-the-fly term simplification s.t. $\texttt{prn}_{\approx}(i,l) \approx_{E_{\mathbb{I}}} \texttt{prn}(i,l)$
$\texttt{exe}_{\approx}(i,p)$	variant of "exe" with on-the-fly term simplification s.t. $\texttt{exe}_{\approx}(i,p) \approx_{E_{\mathbb{I}}} \texttt{exe}(i,p)$
$\sigma_{\underset{e}{\approx}}(i)$	simplifying execution semantics i.e. variant of σ_e in which we simplify intermediate terms on the fly when computing them
$\sigma(i)$	trace semantics of interaction i according to any of σ_d, σ_o, σ_e, σ_n or $\sigma_{\underset{e}{\approx}}$ which are proven equivalent
	Multi-traces (basics)
$\text{Part}(L)$	set of all partitions of set L
$\check{L}$	the discrete partition of set L i.e. $\check{L} = \{\{l\} \mid l \in L\}$
C	a partition of set L i.e. $C \in \text{Part}(L)$
c	a co-localization c element of a partition C i.e. $c \in C$
$\theta_C(a)$	co-localization (in C) on which a occurs i.e. unique $c \in C$ s.t. $\theta(a) \in c$
$\mathbb{A}_{\Omega\|c}$	subset of $\mathbb{A}_\Omega$ containing actions a occurring on c i.e. such that $\theta_C(a) = c$
$\mathbb{T}_{\Omega\|c}$	trace components on co-localization c i.e. $\mathbb{T}_{\Omega\|c} = \mathbb{A}^*_{\Omega\|c}$
$\mathbb{T}_{\Omega\|C}$	set of all multi-traces $\mathbb{T}_{\Omega\|C} = \prod_{c\in C} \mathbb{T}_{\Omega\|c}$ up to signature Ω and partition C
μ	a multi-trace μ element of $\mathbb{T}_{\Omega\|C}$ i.e $\mu \in \mathbb{T}_{\Omega\|C}$
$\mu_{\|c}$	the trace component $\mu_{\|c}$ of multi-trace μ on co-localization c i.e. $\mu = (\mu_{\|c})_{c\in C}$
$\mu[t]_c$	substitution of the component $\mu_{\|c}$ of μ by $t \in \mathbb{T}_{\Omega\|c}$
ϵ_C	empty multi-trace in $\mathbb{T}_{\Omega\|C}$ i.e. $\epsilon_C = (\epsilon)_{c\in C}$
	Multi-traces (algebraic operators)
$\overset{\rightarrow}{\odot}_C$	left concatenation law so that $a \overset{\rightarrow}{\odot}_C \mu = \mu[a.\mu_{\|\theta_C(a)}]_{\theta_C(a)}$
$\overset{\leftarrow}{\odot}_C$	right concatenation law so that $\mu \overset{\rightarrow}{\odot}_C a = \mu[\mu_{\|\theta_C(a)}.a]_{\theta_C(a)}$
$\odot_C$	concatenation law on multi-traces up to C

$⦶_C$	strict sequencing operator on multi-traces
$⊛_C$	weak sequencing operator on multi-traces
$⫴_C$	interleaving operator on multi-traces
$⦶_{C*}$	strict Kleene closure on multi-traces
$⊛^{↰}_{C*}$	weak Head-First closure on multi-traces
$⊛_{C*}$	weak Kleene closure on multi-traces
$⫴_{C*}$	interleaving Kleene closure on multi-traces
$\overline{T}$	prefix-closure of a set of traces or multi-traces T
$\underline{\overline{T}}$	slice-closure of a set of multi-traces T
	Morphisms between types
$\mathbf{proj}_C$	projection operator from traces to multi-traces up to C
$\mathbf{hide}_c(i)$	interaction where (actions occurring on) lifelines from c have been removed from i by the hiding operator $\mathbf{hide}_c$
$\mathbf{elim}_c$	elimination operator on multi-traces erases the trace component on co-localization c
	Multi-trace semantics
$\sigma_{\|C}(i)$	multi-traces exactly accepted by i (projections of accepted traces)
$\sigma^{\dagger}_{\|C}(i)$	multi-traces that are projections of prefixes of traces accepted by i
$\overline{\sigma_{\|C}}(i)$	multi-traces that are prefixes (in the sense of multi-traces) of multi-traces accepted by i
$\underline{\overline{\sigma_{\|C}}}(i)$	multi-traces that are slices (in the sense of multi-traces) of multi-traces accepted by i
$⓪_C(i)$	algebraic multi-trace semantics of i
$\overline{⓪_C}(i)$	algebraic multi-prefix semantics of i
	Analysis algorithms
$\mathbb{G}$	analysis graph, of which traversals constitute the analysis algorithm
$\mathbb{V}$	vertices of the analysis graph
$v \in \mathbb{V}$	a vertex of the analysis graph
$v \rightsquigarrow v'$	a transition between two vertices in graph $\mathbb{G}$
$\|v\|$	measure of vertex v
ω_C	basic algorithm to recognize $\sigma_{\|C}$
$\overline{ⓦ_L}$	algorithm with hiding steps to recognize $\overline{\sigma_{\|\check{L}}} = \overline{⓪}_{\check{L}}$
$\overline{\omega_C}$	algorithm with simulation steps to recognize $\overline{\sigma_{\|C}}$

ω_C^*	algorithm with simulation steps to recognize and discriminate between elements of $\sigma_{\|C}$, $\sigma_{\|C}^\dagger$, $\overline{\sigma_{\|C}}$ and $\underline{\overline{\sigma_{\|C}}}$

Bibliography

[1] Official coq website. https://coq.inria.fr/, . Accessed: 2021-01-26.

[2] Official documentation on coq tactics. https://coq.inria.fr/refman/proof-engine/tactics.html, . Accessed: 2021-01-26.

[3] Papyrus. https://www.eclipse.org/papyrus/, . Accessed: 2021-02-05.

[4] Planttext tool for drawing uml diagrams from plantuml. https://www.planttext.com/, . Accessed: 2021-02-09.

[5] Plantuml. https://plantuml.com/, . Accessed: 2021-02-09.

[6] Raspberry pi. https://www.raspberrypi.org/, . Accessed: 2021-05-26.

[7] Rust programming language. https://www.rust-lang.org/, . Accessed: 2021-05-26.

[8] STMicroelectronics. https://www.st.com/, . Accessed: 2021-05-26.

[9] Cinderella tool. http://www.cinderella.dk/, . Accessed: 2021-02-09.

[10] Diversity tool. https://projects.eclipse.org/projects/modeling.efm, . Accessed: 2021-02-09.

[11] Estelle tool. http://www-lor.int-evry.fr/idemcop/uk/est-lang/download/, . Accessed: 2021-02-09.

[12] Jade tool. https://homepages.dcc.ufmg.br/~coelho/jade.html, . Accessed: 2021-02-09.

[13] Lotos tool. http://cadp.inria.fr/man/lotos.html, . Accessed: 2021-02-09.

[14] Opengeode tool. https://github.com/esa/opengeode, . Accessed: 2021-02-09.

[15] Pragmadev studio. https://www.pragmadev.com/product/studio.html, . Accessed: 2021-02-09.

[16] Spin tool. http://spinroot.com/spin/whatispin.html, . Accessed: 2021-02-09.

[17] Uppaal tool. https://uppaal.org/, . Accessed: 2021-02-09.

[18] Visual paradigm. https://www.visual-paradigm.com/, . Accessed: 2021-02-05.

[19] Web service choreography business process execution language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, . Accessed: 2021-02-09.

[20] Web service choreography description language. `https://www.w3.org/TR/ws-cdl-10/`, . Accessed: 2021-02-09.

[21] Websequencediagrams tool for drawing diagrams. `https://www.websequencediagrams.com/`, . Accessed: 2021-02-09.

[22] Recommendation z.120 message sequence chart. Technical report, International Telecommunication Union, 2 2011.

[23] MQTT version 3.1.1. Technical report, OASIS, 12 2015.

[24] Unified Modeling Language v2.5.1. `omg.org/spec/UML/2.5.1/PDF`, 12 2017.

[25] L. Aceto, W. Fokkink, A. Ingolfsdottir, and B. Luttik. *Finite Equational Bases in Process Algebra: Results and Open Questions*, pages 338–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32425-6. doi: 10.1007/11601548_18.

[26] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, volume 2076 of *Lecture Notes in Computer Science*, pages 797–808. Springer, 2001. doi: 10.1007/3-540-48224-5_65.

[27] C. Andrés, M. Cambronero, and M. Núñez. Formal passive testing of service-oriented systems. In *IEEE International Conference on Services Computing, SCC*, pages 610–613. IEEE, 2010.

[28] F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *IEEE Communications Surveys Tutorials*, 4(1):2–20, 2002. doi: 10.1109/COMST.2002.5341329.

[29] O. Badreddin and T. C. Lethbridge. Model oriented programming: Bridging the code-model divide. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, MiSE '13, page 69–75. IEEE Press, 2013. ISBN 9781467364478.

[30] J. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2):131–146, 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.07.036. Process Algebra.

[31] J. Baeten and J. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205 – 245, 1988. ISSN 0890-5401. doi: 10.1016/0890-5401(88)90027-2.

[32] T. Balabonski, P. Courtieu, R. Pelle, L. Rieg, S. Tixeuil, and X. Urbain. Continuous vs. Discrete Asynchronous Moves: a Certified Approach for Mobile Robots. Research report, Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France, 2018.

[33] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. ISSN 0360-0300. doi: 10.1145/3182657.

[34] B. Bannour, C. Gaston, and D. Servat. Eliciting unitary constraints from timed sequence diagram with symbolic techniques: Application to testing. In *2011 18th Asia-Pacific Software Engineering Conference*, pages 219–226, 2011.

[35] D. Basile, P. Degano, G. Ferrari, and E. Tuosto. Relating two automata-based models of orchestration and choreography. *Journal of Logical and Algebraic Methods in Programming*, 85(3):425–446, 2016. ISSN 2352-2208. doi: 10.1016/j.jlamp.2015.09.011. Interaction and Concurrency Experience.

[36] A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods Syst. Des.*, pages 46–93, 2016.

[37] N. Benharrat, C. Gaston, R. M. Hierons, A. Lapitre, and P. Le Gall. Constraint-based oracles for timed distributed systems. In N. Yevtushenko, A. R. Cavalli, and H. Yenigün, editors, *Testing Software and Systems*, pages 276–292, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67549-7.

[38] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984. ISSN 0019-9958. doi: 10.1016/S0019-9958(84)80025-X.

[39] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, 2010. ISBN 3642058809.

[40] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. *SIGCOMM Comput. Commun. Rev.*, 35(4):265–276, Aug. 2005. ISSN 0146-4833. doi: 10.1145/1090191.1080123.

[41] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, Apr. 1983. ISSN 0004-5411. doi: 10.1145/322374.322380.

[42] M. V. Cengarle and A. Knapp. An institution for uml 2.0 interactions. Technical report, Institut für Informatik, Technische Universität München, 1 2008.

[43] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 10 2007.

[44] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated certified proofs with cime3. volume 10, pages 21–30, 01 2011. doi: 10.4230/LIPIcs.RTA.2011.21.

[45] M. Conti, N. Dragoni, and V. Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, 2016. doi: 10.1109/COMST.2016.2548426.

[46] V. Crespi, A. Galstyan, and K. Lerman. Top-down vs bottom-up methodologies in multi-agent system design. *Autonomous Robots*, 24(3):303–313, 4 2008. ISSN 1573-7527. doi: 10.1007/s10514-007-9080-5.

[47] J. A. Custodio Soares, B. Lima, and J. Pascoal Faria. Automatic model transformation from uml sequence diagrams to coloured petri nets. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2018, page 668–679, 2018. ISBN 9789897582837. doi: 10.5220/0006731806680679.

[48] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. ISSN 1572-8102.

[49] H. Dan and R. M. Hierons. Conformance testing from message sequence charts. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 279–288. IEEE, 2011.

[50] H. Dan and R. M. Hierons. The oracle problem when testing from mscs. *Comput. J.*, 57(7):987–1001, 2014. doi: 10.1093/comjnl/bxt055.

[51] N. Dershowitz and J.-P. Jouannaud. *Rewrite Systems*, page 243–320. MIT Press, Cambridge, MA, USA, 1991. ISBN 0444880747.

[52] C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. Compositional semantics for uml 2.0 sequence diagrams using petri nets. In A. Prinz, R. Reed, and J. Reed, editors, *SDL 2005: Model Driven*, pages 133–148, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31539-1.

[53] A. Engels, S. Mauw, and M. Reniers. A hierarchy of communication models for message sequence charts. *Science of Computer Programming*, 44(3):253 – 292, 2002. ISSN 0167-6423. doi: 10.1016/S0167-6423(02)00022-9.

[54] J. P. Faria and A. C. R. Paiva. A toolset for conformance testing against uml sequence diagrams based on event-driven colored petri nets. *International Journal on Software Tools for Technology Transfer*, 18(3):285–304, 2016.

[55] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz. Timed sequence diagrams and tool-based analysis - A case study. In *UML'99: The Unified Modeling Language - Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 645–660. Springer, 1999.

[56] F. Fondement, P.-A. Muller, L. Thiry, B. Wittmann, and G. Forestier. Big metamodels are evil. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, pages 138–153, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-41533-3.

[57] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04,

page 111–122, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964011.

[58] M.-C. Gaudel. *Software Testing Based on Formal Specification*, pages 215–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14335-9. doi: 10.1007/978-3-642-14335-9_7.

[59] T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An algebraic semantics for message sequence chart documents. 4 1999.

[60] B. Genest and A. Muscholl. Pattern matching and membership for hierarchical message sequence charts. *Theory Comput. Syst.*, 42(4):536–567, 2008. doi: 10.1007/s00224-007-9054-1.

[61] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. *Papyrus: A UML2 Tool for Domain-Specific Language Modeling*, pages 361–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16277-0. doi: 10.1007/978-3-642-16277-0_19.

[62] G. Gonthier. *The Four Colour Theorem: Engineering of a Formal Proof*, page 333. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 9783540878261. doi: 10.1007/978-3-540-87827-8_28.

[63] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling*, 7(2):237–252, 2008.

[64] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, ISR, 2000.

[65] O. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4(4):355–367, 2005.

[66] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., USA, 1985. ISBN 0131532715.

[67] M. Hussein, R. Nouacer, A. Radermacher, A. Puccetti, C. Gaston, and N. Rapin. An end-to-end framework for safe software development. *Microprocessors and Microsystems*, 62:41 – 49, 2018. ISSN 0141-9331. doi: 10.1016/j.micpro.2018.07.004.

[68] M. Jantzen. *Basics of Term Rewriting*, pages 269–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-642-59126-6. doi: 10.1007/978-3-642-59126-6_5.

[69] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9 (3):213–254, 6 2007. ISSN 1433-2787. doi: 10.1007/s10009-007-0038-x.

[70] J.-P. Jouannaud. Associative-commutative rewriting via flattening, 2005.

[71] J.-P. Jouannaud. *Higher-Order Rewriting: Framework, Confluence and Termination*, pages 224–250. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32425-6. doi: 10.1007/11601548_14.

[72] S. Kenneth and L. K. Barry. *Formal syntax and semantics of programming languages - a laboratory based approach*. Addison-Wesley, 1995. ISBN 978-0-201-65697-8.

[73] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252.

[74] C. Kirchner and H. Kirchner. Equational logic and rewriting. In D. M. Gabbay, J. H. Siekmann, and J. Woods, editors, *Handbook of the History of Logic*, volume 9 of *History of Logic and Computation in the 20th Century*. Elsevier, Mar. 2014. URL `https://hal.inria.fr/hal-01183817`.

[75] A. Knapp and T. Mossakowski. UML Interactions Meet State Machines - An Institutional Approach. In *7th Conf. on Algebra and Coalgebra in Computer Science (CALCO 2017)*, volume 72 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[76] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In R. Treinen, editor, *Rewriting Techniques and Applications*, pages 295–304, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02348-4.

[77] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In D. Malkhi, editor, *Concurrency: the Works of Leslie Lamport*, pages 179–196. ACM, 2019.

[78] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction and process-oriented choreographies. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 323–332, 2008. doi: 10.1109/SEFM.2008.11.

[79] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 12 1997. ISSN 1433-2779. doi: 10.1007/s100090050010.

[80] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *SIGPLAN Not.*, 51(4):517–530, Mar. 2016. ISSN 0362-1340. doi: 10.1145/2954679.2872374.

[81] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 7 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814.

[82] J.-L. Lions, L. Luebeck, J.-L. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[83] D. Longuet. Global and local testing from message sequence charts. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012*, pages 1332–1338. ACM, 2012.

[84] M. S. Lund and K. Stølen. A fully general operational semantics for uml 2.0 sequence diagrams with potential and mandatory choice. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, pages 380–395, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37216-5.

[85] M. S. Lund, A. Refsdal, and K. Stølen. *4 Semantics of UML Models for Dynamic Behavior*, pages 77–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16277-0. doi: 10.1007/978-3-642-16277-0_4.

[86] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*, pages 378–393. ACM, 2015.

[87] E. Mahe. Coq proof for the correctness of the multi-trace analysis algorithm. `https://erwanm974.github.io/coq_hibou_label_multi_trace_analysis/`, . Accessed: 2021-02-21.

[88] E. Mahe. Coq proof for the equivalence of the semantics. `https://erwanm974.github.io/coq_hibou_label_semantics_equivalence/`, . Accessed: 2021-02-21.

[89] E. Mahe. Example coq proof for a property on a toy process algebra. `https://erwanm974.github.io/coq_toy_process_algebra/`, . Accessed: 2021-05-20.

[90] E. Mahe. Hiboux tool. `https://github.com/erwanM974/hibou_efm`, . Accessed: 2021-02-21.

[91] E. Mahe. Hibou tool. `https://github.com/erwanM974/hibou_label`, . Accessed: 2021-02-21.

[92] E. Mahe. Small experiment using hibou and hiboux on an example mqtt interaction. `https://github.com/erwanM974/hibou_mqtt_benchmark_experiment`, . Accessed: 2021-05-20.

[93] E. Mahe, C. Gaston, and P. L. Gall. Revisiting semantics of interactions for trace validity analysis. In H. Wehrheim and J. Cabot, editors, *Fundamental Approaches to Software Engineering*, pages 482–501, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45234-6.

[94] E. Mahe, B. Bannour, C. Gaston, A. Lapitre, and P. Le Gall. A small-step approach to multi-trace checking against interactions. SAC '21, page 1815–1822, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381048. doi: 10.1145/3412841.3442054.

[95] E. Mahe, C. Gaston, and P. Le Gall. A structural operational semantics for interactions with a look at loops, 2021.

[96] C. Marché and X. Urbain. Modular and incremental proofs of ac-termination. *Journal of Symbolic Computation*, 38(1):873–897, 2004. ISSN 0747-7171. doi: 10.1016/j.jsc.2004.02.003.

[97] S. Mauw and M. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37, 3 1996. doi: 10.1093/comjnl/37.4.269.

[98] S. Mauw and M. A. Reniers. High-level message sequence charts. In *SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Proceedings*, pages 291–306. Elsevier, 1997.

[99] S. Mauw and M. A. Reniers. Operational semantics for msc. *Computer Networks*, 31(17):1785–1799, 1999.

[100] Z. Micskei and H. Waeselynck. The many meanings of uml 2 sequence diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[101] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982. ISBN 0387102353.

[102] Muan Yong Ng and M. Butler. Towards formalizing uml state diagrams in csp. In *First International Conference onSoftware Engineering and Formal Methods, 2003.Proceedings.*, pages 138–147, 2003. doi: 10.1109/SEFM.2003.1236215.

[103] F. Neves, N. Machado, and J. Pereira. Falcon: A practical log-based analysis tool for distributed systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 534–541. IEEE, 2018.

[104] H. N. Nguyen, P. Poizat, and F. Zaïdi. Passive conformance testing of service choreographies. In *Proceedings of the ACM Symposium on Applied Computing, SAC*, pages 1528–1535. ACM, 2012.

[105] R. O'Connor. Essential incompleteness of arithmetic verified by coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31820-0.

[106] R. M. O'Keefe and D. E. O'Leary. Expert system verification and validation: a survey and tutorial. *Artif. Intell. Rev.*, 7(1):3–42, 1993. doi: 10.1007/BF00849196.

[107] U. Ozeer, L. Letondeur, G. Salaün, F.-G. Ottogalli, and J.-M. Vincent. F3ARIoT: A Framework for Autonomic Resilience of IoT Applications in the Fog. *Internet of Things*, pages 1–54, Dec. 2020. doi: 10.1016/j.iot.2020.100275.

[108] G. Pedroza, P. Le Gall, C. Gaston, and F. Bersey. Timed-model-based Method for Security Analysis and Testing of Smart Grid Systems. In *19th International Symposium on Real-Time Distributed Computing (ISORC) 2016*, York, United Kingdom, May 2016. doi: 10.1109/isorc.2016.15.

[109] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, Apr. 1981. ISSN 0004-5411. doi: 10.1145/322248.322251.

[110] A. Pferscher and B. K. Aichernig. Learning abstracted non-deterministic finite state machines. In V. Casola, A. De Benedictis, and M. Rak, editors, *Testing Software and Systems*, pages 52–69, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64881-7.

[111] S. Pickin and J.-M. Jézéquel. Using uml sequence diagrams as the basis for a formal test description language. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, pages 481–500, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24756-2.

[112] G. D. Plotkin. An operational semantics for CSP. In *Formal Description of Programming Concepts : Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts- II*, pages 199–226. North-Holland, 1983.

[113] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web, WWW*, pages 973–982. ACM, 2007.

[114] M. Rocha, A. Simão, T. Sousa, and M. Batista. Test case generation by efsm extracted from uml sequence diagrams. pages 135–140, 7 2019. doi: 10.18293/SEKE2019-133.

[115] J. E. Rooda, D. A. van Beek, and J. C. M. Baeten. Process algebra. In P. A. Fishwick, editor, *Handbook of Dynamic System Modeling*. Chapman and Hall/CRC, 2007. doi: 10.1201/9781420010855.ch19.

[116] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag, Berlin, Heidelberg, 1997. ISBN 3540604200.

[117] B. J. Sauser, R. R. Reilly, and A. J. Shenhar. Why projects fail? how contingency theory can provide new insights – a comparative analysis of nasa's mars climate orbiter loss. *International Journal of Project Management*, 27(7):665–679, 2009. ISSN 0263-7863. doi: 10.1016/j.ijproman.2009.01.004.

[118] T. J. Schaefer. The complexity of satisfiability problems. In R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho, editors, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 216–226. ACM, 1978. doi: 10.1145/800133.804350.

[119] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering, ICSE*, pages 418–427. IEEE, 2004.

[120] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, pages 32–46, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04425-0.

[121] K. Shintani and N. Hirokawa. Coll: A confluence tool for left-linear term rewrite systems. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 127–136, Cham, 2015. Springer International Publishing. ISBN 978-3-319-21401-6.

[122] H. Störrle. Semantics of interactions in uml 2.0. In *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, pages 129–136, 10 2003. doi: 10.1109/HCC.2003.1260216.

[123] J. Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78917-8. doi: 10.1007/978-3-540-78917-8_1.

[124] H. Waeselynck, Z. Micskei, N. Rivière, Á. Hamvas, and I. Nitu. Termos: A formal language for scenarios in mobile computing systems. In P. Sénac, M. Ott, and A. Seneviratne, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 285–296, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29154-8.

[125] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993. ISBN 0262231697.

[126] A. Yamada, K. Kusakari, and T. Sakabe. Nagoya termination tool. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi*, pages 466–475, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08918-8.

[127] H. Zankl, B. Felgenhauer, and A. Middeldorp. Csi – a confluence tool. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 499–505, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22438-6.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Interfaces:
matériaux, systèmes, usages

Titre: Sémantique opérationnelle des interactions pour la vérification d'exécutions partiellement observées de systèmes distribués
Mots clés: systèmes concurrents et distribués, sémantique dénotationnelle et opérationnelle, réécriture de terme, analyse de logs distribués, observation partielle, exécution symbolique

Les interactions représentent des communications asynchrones dans un contexte distribué et sont associées à des diagrammes qui sont faciles à prendre en main tout en pouvant spécifier des comportements précis. Dans cette thèse nous formalisons ce type de modèles avec un Langage d'Interactions (LI) se présentant sous la forme d'une algèbre de termes incluant des opérateurs pour le séquencement strict et faible, la composition alternative et parallèle ainsi que quatre boucles distinctes.

Notre LI est équipé d'une sémantique associant à chaque interaction un ensemble de traces (séquences d'évènements) pouvant être exprimées. Nous proposons deux sémantiques dont nous avons prouvé l'équivalence. La première formulation dénotationnelle est donnée sous la forme d'un homomorphisme d'algèbres, donnant accès aux techniques de réécriture afin de définir des classes d'équivalence et des formes normales. Une seconde formulation, opérationnelle, permet diverses applications en vérification formelle.

Les exécutions d'un système distribué peuvent être observées au travers de logs des évènements de communication collectés localement. Sans horloge globale il n'est pas possible de réordonner ces évènements globalement. Analyser une exécution revient donc à analyser un ensemble de traces qu'on appelle une multi-trace. De plus, sur chaque composante locale, il se peut que l'observation ait commencé trop tard ou ait cessé trop tôt. Ainsi, une multi-trace peut correspondre à une observation partielle d'une exécution. Tirant parti de la sémantique opérationnelle nous proposons des algorithmes d'analyse permettant d'identifier une multi-trace comme étant une observation d'un comportement spécifié par une interaction.

Notre approche a été implémentée au sein d'un outil appelé HIBOU qui permet de spécifier et dessiner des interactions, d'explorer leur sémantiques, de calculer des formes normales ou d'analyser des multi-traces. Nous avons étendu notre LI pour inclure des données sous la forme de variables définies localement. Des gardes, expressions booléennes sur les variables, peuvent conditionner l'exécution d'actions et les messages peuvent porter des données exprimées à l'aide des variables. L'extension aux données a été implémentée en utilisant les techniques d'exécution symbolique. Cet outil étendu a été utilisé pour un cas d'usage industriel dans le cadre du projet FUI DisTA.

Title: An operational semantics of interactions for verifying partially observed executions of distributed systems
Keywords: distributed and concurrent systems, denotational and operational semantics, term rewriting, distributed log analysis, partial observation, symbolic execution

Interactions represent asynchronous communications in a distributed context and can be represented graphically in an intuitive manner while allowing the specification of precise scheduling policies for ordering events. In this thesis, we formalize such models as an Interaction Language (IL) taking the form of a term algebra which includes strict and weak sequencing, alternative and parallel composition and four semantically distinct loops to express nuances when repeating behaviors.

This IL is equipped with a semantics associating a set of traces (sequences of events) to each interaction. A denotational formulation as a homomorphism allows the use of rewriting to define equivalence classes and normal forms. An operational formulation, proven equivalent to the first, allows further applications in formal verification.

During the execution of a distributed system, communication logs might be collected locally. Without a global clock, events cannot be reordered globally. Hence analyzing an execution comes back to analyzing a set of local traces that we call a multi-trace. In addition one might start the observation too late or end it too early on any local component. As a result, a multi-trace might correspond to a partially observed execution. Taking advantage of the operational semantics we define algorithms for identifying multi-traces as being observations of behaviors specified by an interaction model.

We have implemented our approach in a formal verification tool: HIBOU, which allows the specification and drawing of interactions, the exploration of their semantics, the computation of normal forms or the analysis of multi-traces. The IL can be extended to include data in the form of locally defined variables. Guards, formulated as Boolean expressions on variables can condition the execution of individual actions, while messages can carry data. This extension has also been implemented and we use symbolic execution to animate models and perform the analyses. An industrial case study has been carried out in the context of the DisTA FUI project.

Maison du doctorat de l'Université Paris-Saclay
2ème étage aile ouest, Ecole normale supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France