Symbolic analysis of scenario based timed models for component based systems : Compositionality results for testing

Boutheina Bannour

ÉCOLE CENTRALE DES ARTS ET MANUFACTURES « ÉCOLE CENTRALE PARIS »

COMMISSARIAT À L'ÉNERGIE ATOMIQUE ET AUX ÉNERGIES ALTERNATIVES « CEA »

THÈSE

présentée par

Boutheina Bannour

pour l'obtention du

GRADE DE DOCTEUR

Spécialité : Informatique

Laboratoires d'accueil :
Laboratoire Mathématiques Appliquées aux Systèmes (MAS)
Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués (LISE)

SUJET :

Symbolic analysis of scenario based timed models for component-based systems: Compositionality results for testing

soutenue le : 14 juin 2012

devant un jury composé de :

Yves Le Traon	**Président**	**Université du Luxembourg**
Marc Aiguier	**Directeur de thèse**	**Ecole centrale de Paris – MAS**
Christophe Gaston	**Encadrant**	**CEA LIST – LISE**
Fatiha Zaidi	**Rapporteur**	**Université Paris-Sud XI – LRI**
Frédéric Mallet	**Rapporteur**	**Université Nice Sophia Antipolis – INRIA**
Pascale Le Gall	**Examinateur**	**Ecole centrale de Paris – MAS**

2012ECAP0027

Résumé en français
Dans cette thèse, nous décrivons comment on peut utiliser un diagramme de séquence UML avec des contraintes de temps MARTE pour spécifier complètement le comportement des systèmes à base de composants tout en faisant abstraction des rôles fonctionnels des composants. Nous avons proposé une approche qui permet d'analyser ces spécifications d'une manière modulaire. Pour cela, nous avons attribué une sémantique opérationnelle aux diagrammes de séquence en les traduisant vers les TIOSTS qui sont des automates symbolique et temporisé. Nous avons utilisé des techniques d'exécution symbolique pour calculer les exécutions du système sous la forme d'un arbre symbolique. Nous avons défini des mécanismes de projection pour extraire l'arbre d'exécution associé à un composant sous-jacent. L'arbre résultant de la projection caractérise les comportements attendus du composant et peut être utilisé comme une référence pour valider le système bout par bout. Pour ce faire, nous nous sommes intéressés à des techniques de test. Nous avons présenté un résultat qui ramène la conformité du système à la conformité des composants qui le composent. Sur la base de ces résultats, nous avons proposé une méthodologie incrémentale de test basé sur des spécifications décrites sous la forme de diagrammes de séquence.

Mots-clés
Diagramme de séquence UML, contraintes de temps MARTE, transformation de modèles, exécution symbolique, et test de conformité.

Résumé en Anglais
In this thesis, we describe how to use UML sequence diagrams with MARTE timing constraints to specify entirely the behavior of component-based systems while abstracting as much as possible the functional roles of components composing it. We have shown how to conduct compositional analysis of such specifications. For this, we have defined operational semantics to sequence diagrams by translating them into TIOSTS which are symbolic automata with timing constraints. We have used symbolic execution techniques to compute possible executions of the system in the form of a symbolic tree. We have defined projection mechanisms to extract the execution tree associated with any distinguished component. The resulting projected tree characterizes the possible behaviors of the component with respect to the context of the whole system specification. As such, it represents a constraint to be satisfied by the component and it can be used as a correctness reference to validate the system in a compositional manner. For that purpose, we have grounded our validation framework on testing techniques. We have presented compositional results relating the correctness of a system to the correctness of components. Based on these results, we have defined an incremental approach for testing from sequence diagrams.

Key words
UML sequence diagrams, MARTE timing constraints, model transformation, symbolic execution, and conformance testing.

Contents

Chapter 1

Introduction

Contents

1.1 Context of the thesis

As the complexity of systems grows, new specification paradigms have emerged in recent years. Component-based software engineering (CBSE) is among the most popular paradigms adopted in the industry. CBSE consider systems to be a collection of many functional units referred to, in a generic sense, as *components*. A component is encapsulated and communicates with other components (or with the system environment) only through clearly defined interfaces. Some of the components may be realized in-house or acquired from a third party (*e.g.* "commercial off-the-shelf" components (COTS)). Some others are created to be specifically used in the system. The whole system is designed by bringing together all these components into a coherent whole. Characterizing the system architecture is quite straightforward and done by specifying connections between components interfaces. Let us remark that component systems behaviors are often specified "only" by models presenting their component architectures together with more or less precise concurrent functional descriptions (including, sometimes, timing constraints). In particular, when dealing with COTS components, it may be difficult to obtain exhaustive formal specifications. The role of the specification is to define the intended system behaviors. But most specifications are not really system specifications: they are rather a collection of component specifications and the collaboration between component to form the system is left implicit in the semantics of connectors occurring in the system architecture descriptions. Such specifications do not truly specify explicitly behaviors of systems. Thus relating them to informal system behavioral requirements may be complicated. Moreover, such specifications are very close to implementations. Modern component systems, in a lot of domains, are often very big. Therefore, such specifications are likely to be very big, and thus, hard to analyze and full of bugs. We study another kind of modeling techniques (based on UML sequence diagrams [73]) to describe more abstract and more "system-focused" specifications. In this thesis, we suggest to use UML sequence diagrams with MARTE timing features [41] in order to fully specify the system intended behavior while abstracting as much as possible the roles of components (which are still seen as black boxes). Sequence diagrams are interaction-oriented specifications having the modeling power, in terms of structuring operators, capable of specifying the system behavior as a whole. Industry wants practical methods which do not require mathematical skills: the use of the sequence diagrams as being an acceptable technology to engineers seems adequate in that sense.

Now, validation of component systems may call for modular techniques. This is firstly because fully implementing such systems may take time. Waiting for the full implementation to begin validation may lead to discover bugs late and to question design choices made months ago which

may have harmful economic consequences. Moreover, big component systems may be hard to test "as a whole" because they may involve lots of subsystems destributed over a network of devices, which substantially complicates the building of the testing architecture. In this thesis, we study how to extract (we also say elicit), from systems specifications, constraints on components or subsystems behaviors, and how they can be used in a modular validation process.

In the sequel, we illustrate all these points on a concrete case. We present, in Section 1.2, an example of a component-based system specified using UML MARTE composite structures [73, 39]. It will serve as a running example in the introduction. We discuss, in Section 1.3, our use of sequence diagrams. Then, we discuss, in Section 1.4, the elicitation of subsystems behaviors from systems specifications and their utility in a modular testing process of systems.

1.2 Component-based system specifications

As glimpsed previously, components are functional units interacting through the communication architecture. A component has interaction points called *ports* which constitute the component interface. An example of a component system architecture, taken from the MARTE standard, is illustrated in Figure 1.1. It is a simplified Flight Management System (FMS) in a modern aircraft. The system computes the trajectory of the aircraft and generates continuous navigation commands to other equipments. In Section 1.3, we show how we can specify the full system behavior as a sequence diagram.

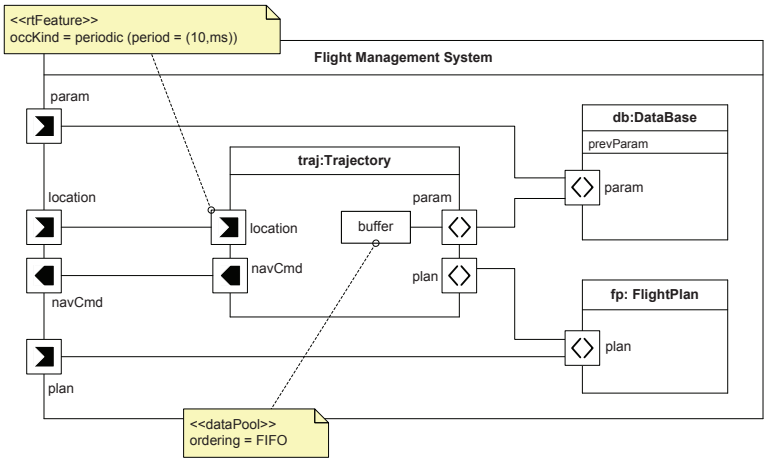


Figure 1.1: Flight management system as UML MARTE composite structure

Components receive input data and emit data through their ports. Internally, they compute results from received data in order to provide services. Connectors are abstractions of the different communication protocols between components (correspond to the lines in Figure 1.1) permitting to convey data between components. FMS is made of three components: $Trajectory$, $FlightPlan$, and $Database$. $Trajectory$ makes use of the flight plan data, as well as the current plane location to perform computations. It receives from the environment (specifically, from a sensor not depicted in the diagram) continuously the current location of the aircraft (on its port *location*). The frequency of locations measure (periodic every 10 milliseconds) is identified by the added MARTE annotation *rtFeature* [40]. This annotation may be applied to a port to specify its temporal behavior (here, it was applied to port *location*). $Trajectory$ explicitly asks to access the plan when needed (the port *plan* of $Trajectory$ is bound to the dedicated port *plan* of $FlightPlan$). $Trajectory$ also makes use of the performance and fuel consumption parameters stored in its cache. A pilot may change these parameters, initially stored in the database, when the FMS is in operation. The pilot is in the environment of the system (and interacts with

components *FlightPlan* and *Database* respectively through ports *plan* and *param*). If so, the *Database* component notifies *Trajectory* that the new parameters values need to be taken into account (their respective ports *param* are connected). The parameters are stored in the cache of *Trajectory*, in their arrival order (see the MARTE annotation *dataPool* stating that data is inserted/consumed in a FIFO order). When computations are completed, *Trajectory* generates navigation commands transmitted (through a dedicated port *navCmd*) to external equipment.

In spite of the presence of functional annotations, sometimes component-connectors specifications are still poor, in the sense that they focus only on the composition of components in terms of provided and required services (by connecting ports). Obviously, all the logical sequence of interactions described previously cannot be inferred from such specifications: the system behaviors which result from the interactions of components (and the system environment) through the communication architecture are not specified. We address, in the next section, the question of how to produce specifications of the system overall behaviors, that is, behaviors of the system which are considered to be correct at this level of abstraction (i.e. the one of component-connectors specifications).

1.3 Produce correct specifications

> "A final difficulty encountered in modeling is the frequent lack of good requirement documents associated with the project. Most of the time, industrial requirement documents are either almost nonexistent or far too verbose."
> Jean-Raymond Abrial, Theory becoming practice, Journal of Universal Computer Science, 2007

Implementing a correct system starts with a specification which can be considered as correct. Usually, the relation between early requirements and the specification is informally stated. This is because the notion of formal correctness relates two mathematical objects: an object to evaluate against a reference object from which one can define behaviors that are considered to be correct. Here, the object to evaluate is indeed the specification of the system and there exists necessarily at some phase of the system development cycle, a specification of all possible executions of the system which can be considered as the most abstract. The evaluation of the correctness of such a specification may be done by human proof-reading the requirements. Another alternative is to execute the specification on simulators in order to construct scenarios of executions and analyze them manually. Sometimes, requirements are formal (e.g. using temporal logics) in which case formal verification and validation techniques can be applied to evaluate the satisfaction of such requirements (e.g. using model checking). Unfortunately, most abstract available specifications of the system behaviors are described in terms of components and connectors as shown in Figure 1.1. In such specifications, we specify the components behaviors (e.g. using state diagrams) and the system behaviors as a whole are left implicit (simply obtained by connecting the components). As a matter of fact, we never explicitly characterize the intended behaviors of the system. This complicates the understanding of what behaviors have been really specified at the system level. That is why engineers often use simulation platforms to exercise the system behaviors. However, conducting these simulations while covering a proper part of the system behaviors is a hard task because of the voluminous size of the specification, which shares the same level of abstraction as the implementation.

Sequence diagrams are visual formalisms for scenario-based specifications and is a specification of the OMG consortium (Object Management Group, for modeling object-oriented systems) as part of the Unified Modeling Language (UML). The predecessors of sequence diagrams are SDL (Specification and Description Language) [49] and MSC (Message Sequence Chart) [51]. MSC

is an ITU-T standard (ITU Telecommunication Standardization Sector). Earlier versions of MSC were an input to make UML 2.0 sequence diagram. There are few technical differences [45] between the MSC and sequence diagrams in their modeling power. Our motivation is to use rather the UML technology very tightly related to model-driven development (MDD). Moreover, we use sequence diagrams with time properties formulated using MARTE::VSL [41]. The Modeling and Analysis of Real-Time and Embedded systems (MARTE) profile is also an OMG standard which extends UML with capabilities for model-driven development of embedded and real-time systems. An example of a sequence diagram is depicted in Figure 1.2. It specifies the intended behaviors of the Flight Management System illustrated in Figure 1.1.

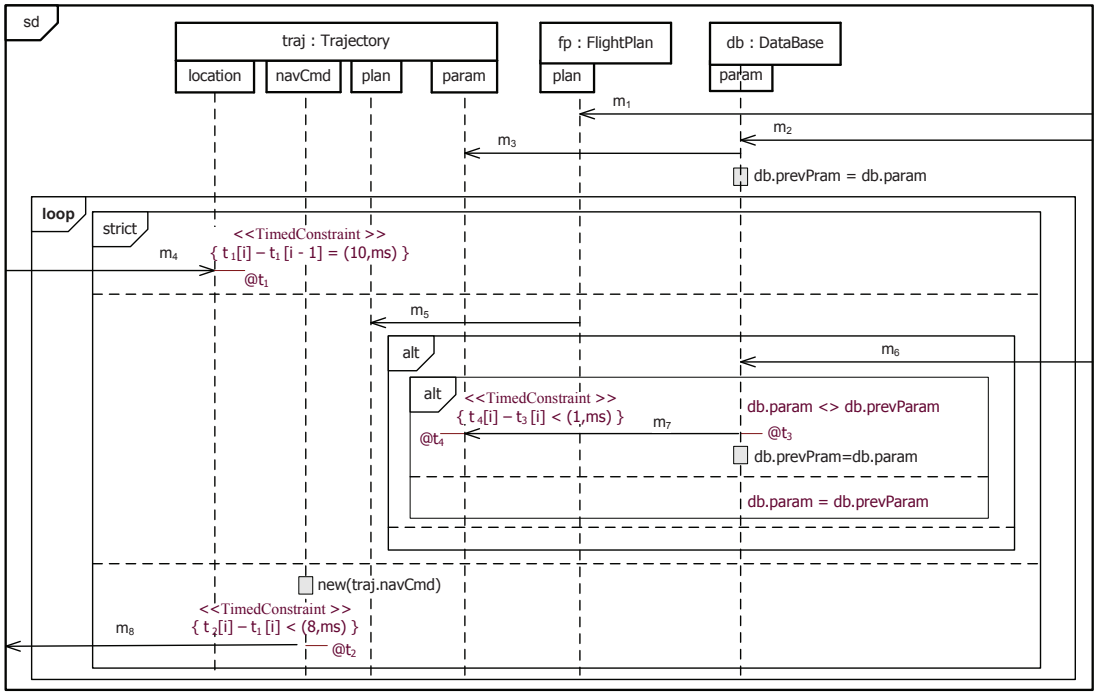


Figure 1.2: Flight Management System behavior as UML MARTE sequence diagram

In sequence diagrams, one may describe execution scenarios in terms of partially-ordered sequences of messages exchanged between basic interaction entities which represent ports owned by components to communicate with their environment. Figure 1.2 depicts messages (arrows) $m_1, \ldots,$ m_8 exchanged between all connected component ports (see Figure 1.1). For example, consider the initialization messages m_1, m_2 and m_3: m_1 and m_2 convey information about the flight coming from the environment (the pilot) respectively towards components $FlightPlan$ and $DataBase$; and m_3 notifies $Trajectory$ of the flight parameters sent by the $DataBase$. Message descriptions may include constraints on the data transmitted and the time at which the message occurs. An example of a *data constraint* is *db.param* <> *db.prevParam*, which states that the parameters computed by $DataBase$ carried by m_7 arriving at port *param* of $Trajectory$ are different form the previously sent parameters. An example of a *time constraint* is $t_1[i] - t_1[i-1] = (10, ms)$ states that the delay between the previous occurrence of m_4 conveying the location measure from the environment and the current one is exactly 10 milliseconds.

The usual practice is to use sequence diagrams to describe particular system scenarios in terms of message exchanges. However, sequence diagrams introduce structuring operators to define precedence or concurrency between occurrences of messages. This allows one to specify completely the system behavior. Figure 1.2 shows how the FMS behaviors may be entirely specified thanks to these operators. Typically, the cyclic behavior of the system during the flight is captured by the iterative operator *loop*: $Trajectory$ receives locations measures and generates navigations

commands (see messages $m_4, \ldots, m_8$) while being notified of any new parameters (see messages m_6 and m_7) from the *DataBase*. Notifications are random behaviors captured by the (most external) *alt* operator which specifies, here, that either the behavior corresponding to notifications occurs or nothing happens. The whole cyclic behavior is sequenced by the (most external) *strict* operator, which introduces synchronization points in the system behavior at the level of its dashed horizontal lines. Finally, note that ports are considered as particular variables to store data in transit. Their assigned values change if a message is received or if the *new* action is applied. The effect of *new* is to assign a random value to the port. We have introduced *new* simply to control values stored on ports when computations are abstracted.

What we have presented in Figure 1.2 is actually a subset of the sequence diagram constructs with slight enrichments added in order to better control the behaviors of the underlying components in the system (e.g. the *new* action). In fact, we have defined a subset of the sequence diagram language that allowed us to specify many examples, in particular to specify a railway use case, whose natural language description has been provided by ALSTOM in the Context of the ITEA project VERDE [1]. The objective of the use case is to operate a train along the tracks, in the requested direction of traction, while ensuring that all safety parameters and delays are always preserved.

Now we want to automatically analyze timed sequence diagrams which specify the intended behaviors of component-based systems. While defining tools based on sequence diagrams as a popular industrial standard is quite straightforward, the key obstacle to using sequence diagrams has been demonstrated ([70]) to be its informal semantics ambiguously defined in natural language.

We start by giving an operational semantics to the subset of the sequence diagrams that we have selected. The semantics was given to sequence diagrams by translating them into a formalism of communicating symbolic automata. This formalism is an extension, defined in this thesis, of IOSTS (Input/Output Symbolic Transition Systems) to support timing features. IOSTS have been widely used in black box testing approach based on symbolic execution [1, 36, 33, 32, 30, 34]. Several works (e.g. [57, 66, 78, 92, 72]) have addressed the issue of formal semantics for UML sequence diagrams (or similar ones like MSC [51]). Our proposal of semantics is related to our use of symbolic techniques for (timed) black box testing. IOSTS are automata where transitions are labeled by guards over variables, by symbolic communication actions and by variable assignments which capture state evolutions. Our extension called *TIOSTS (Timed Inout/Output Symbolic Transition Systems)*, allows to define in addition timing constraints on transitions and particular variables to store dates according to sequence diagrams semantics. We may have used other formalism for the same purpose, for example the Timed Automata ([3]); but we have chosen TIOSTS because IOSTS are already associated with symbolic execution techniques [1, 36] and tools. The tools are gathered in the *Diversity* platform [76, 26] [2] developed in our laboratory (of the CEA LIST). We had simply to extend these existing techniques and tools with capabilities to handle time and hence be able to execute symbolically timed sequence diagrams. Symbolic execution [52] is an analysis technique which was initially defined for programs. Symbolic execution represents values of (program) variables with symbols introduced on the fly and accumulates conditions on them throughout the execution. These conditions are the so-called *path conditions*. This technique is used to check the behavior of code (e.g. using Java Path Finder [23] with Java code) or of symbolic specifications (e.g. using Diversity) in order to detect livelocks and deadlocks. It is used as well to generate test inputs by solving the path conditions. Therefore, our operational semantics (jointly defined with tools extensions) enables this kind of analysis on sequence diagrams. Now, we are interested in extracting intended behaviors of components in the context of their use in the system.

[1] *http://www.eclipse.org/modeling/mdt/papyrus/*
[2] formerly called *AGATHA*

1.4 Extract unitary requirements for system components

> "There is general consensus that the most significant problems in software development are due to inadequate requirements, especially where these concern what one component or subsystem may expect of another."
> John Rushby, Automated formal methods enter the mainstream, Journal of Universal Computer Science, 2007

We study how to use symbolic techniques to elicit requirements for any component (or for any subsystem) from the sequence diagram specifying all the possible executions of the system as a whole. As discussed in the previous section, basic components are seen in this kind of specification as black boxes, only observable by sequences of incoming and outgoing values through their ports. However, by characterizing precedence and concurrency rules on the messages exchanged between components in the system, the designer implicitly characterizes constraints on the component behaviors themselves: if a message has to transit from a component to another at a given instant, then the sending component should produce the requested value at the appropriate instant. Consider again the FMS example as described in Figure 1.2. Here is an example of a specified behavior for the *FlightPlan* component in the context of FMS system that we can deduce from the sequence diagram: when *DataBase* receives new parameters (through m_6), it sends them to $Trajectory$ within 0.8 milliseconds (through m_7). Or more evidently, we can deduce that when $Trajectory$ receives the location measure (message m_4), it reacts by sending navigation commands (message m_8) within 8 milliseconds.

As glimpsed previously, based on symbolic techniques we want to derive such unitary behaviors (relative to one component) from the sequence diagram. For that purpose, we adapted the projection introduced in [33] to project the symbolic behaviors associated with the sequence diagram on the interface of components. Those behaviors obtained by projection can be used as unitary test purposes to select a COTS component or as guideline to produce the component code. Within a testing context, the following question arises naturally: if we decompose the system into subsystems and if each subsystem conforms to its requirements obtained by projection, what about the conformance of the system as a whole to the sequence diagram? For the purpose of answering this question, we need a formal definition of the correctness. We use the *tioco* conformance relation defined in [20, 54, 81]. *tioco* defines when an implementation is correct with respect to some given specification: *tioco* states that after a specified behavior, any reaction or delay observed of the implementation is intended in the specification. We establish in the frame of *tioco*, a theorem which relates the conformance of a system to the conformance of the subsystems composing it. This kind of result is interesting because waiting until the system is finished to test means errors are discovered too late in the development cycle. This may be costly. In addition, one may question the different design choices since it is difficult to figure out where these errors come from. Far better is to test components as they are implemented. The elicited unitary behaviors by our projection mechanism are exactly the behaviors expected from components (or subsystems) in the context of the system. Our result states that any fault discovered by testing the system as a whole, can be discovered by unitary testing the subsystems. Therefore, it is sufficient to test components earlier in the development cycle, before being assembled to realize the system. Clearly, testing components separately is more practical, in the sense that the tester has full observability, than developing a complicated testing architecture, especially in a distributed context, and test the entire system while running. This result is a first step towards a fully automated process of test generation in which the system is tested in a modular way subsystem-by-subsystem.

Outline of the thesis

The thesis is structured as follows:

Chapter 2: UML MARTE sequence diagram presents the subset of the timed sequence diagram that we use to specify the system intended behaviors.

Chapter 3: Formal preliminaries gives preliminaries about typed equational logic and automata formalisms related to our work.

Chapter 4: Formalizing UML MARTE sequence diagram provides a formalization of sequence diagrams in a logical framework were time and data are handled as first order structures.

Chapter 5: Timed Input Output Transition Systems (TIOSTS) defines TIOSTS formalism and its associated symbolic execution.

Chapter 6: Operational semantics of UML MARTE Sequence Diagram gives operational semantics to the selected subset of the sequence diagrams as translation rules into the TIOSTS formalism.

Chapter 7: Application to testing establishes compositional results relating the correctness of systems to the correctness of components composing them and links these results to sequence diagram as a reference for testing.

Chapter 8: Related Work reviews the state of the art relevant to the thesis: the synthesis of automata formalism from sequence diagrams; the use of sequence diagram in testing; and the analysis of sequence diagram to obtain requirements by projection.

Chapter 9: Implementation and experiments describes the prototype of our approach evaluated on a railway use case.

Chapter 10 is the conclusion.

Chapter 2

UML MARTE sequence diagram

Contents

In this chapter, we present our use of the UML sequence diagrams specialized with the MARTE profile to handle timing constraints. First, we informally define sequence diagrams based on their graphical representation. Then we introduce the syntax of sequence diagrams as being a part of the UML language given with a metamodel (called abstract syntax in the UML parlance): The metamodel describes the concepts of the language and the relations between them, sequence diagrams are just a concrete notation to depict them. Hence the graphical notation is called *concrete syntax* in the OMG specification.

2.1 Basics and graphical representation

A sequence diagram is made up of two dimensions : participants are distributed along the horizontal dimension and the vertical dimension is temporal. A sequence diagram is a graphical depiction of information exchange between participants over time. We give in the following some background on the key concepts of a sequence diagram. As we go along this section, we also discuss point by point the different restrictions and choices we impose in the use of sequence diagrams.

2.1.1 Lifelines and messages

Sequence diagrams essentially rely on two concepts, *lifelines* and *messages*. Consider the elementary sequence diagram illustrated in Figure 2.1. A *lifeline* is depicted as a vertical line along which time flows from top to bottom. Specifically, a *lifeline* represents a single port in our case.

In the example, there are three lifelines representing respectively from left to right ports p_1, p_2 and p_3. Participants communicate through *message* passing. A *message* is depicted as an arrow from the sending to the receiving *lifelines*. An arrow with an open head corresponds to an asynchronous *message*. Asynchronous means that the behaviors are not blocked until a message sent is received. In the sequel, we only consider asynchronous messages. Continuing the example, two messages are depicted namely m_1 and m_2. At port p_2 level, the fact that the message m_1

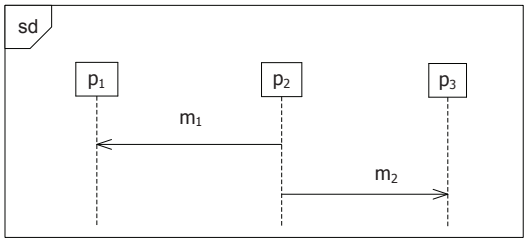
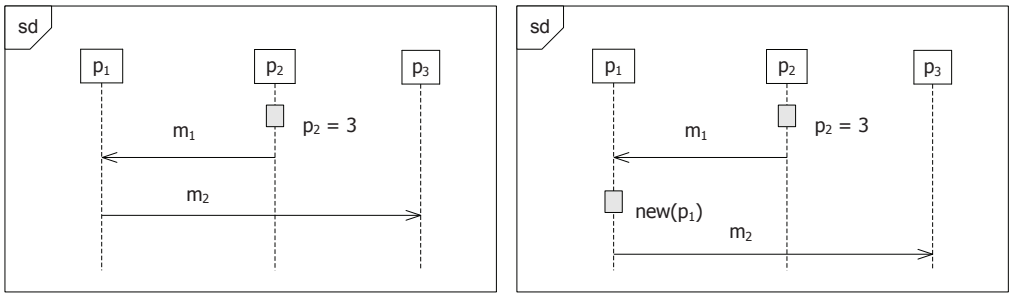


Figure 2.1: Elementary sequence diagram

is asynchronous implies that : once m_1 was sent, the emission of the message m_2 may happen before the reception of the message m_1 by the port p_2. Each lifeline is associated with its own time scale and by default an instant corresponding to a point on a lifeline cannot be compared (in terms of precedence) to an instant of another lifeline. Only messages bridging two lifelines induce a partial order on the instants of different lifelines. In the example, the receptions of the messages m_1 and m_2 by respectively the ports p_1 and p_3 may occur in any order, however the emission of m_1 by the port p_1 happens necessarily before the emission of m_2 by p_2 and therefore the reception of m_2 by the port p_3.

Notice that the message labels (figuring on the message arrow) which appear in our diagram are the messages identifiers and not the conveyed data. We consider a piece of data conveyed by a message as an abstraction of all kinds of piece of information that can be exchanged between lifelines. The nature of the entities that the involved lifelines represent (ports in our context), decides of the nature of the piece of data in transit. In order to control the exchanged data, we use the port as a locale variable to store these data. The question now is where is stored the data in transit before being received by the lifeline? In fact, in the case of asynchronous messages, the UML does not precise the underlying communication model. We choose to associated an implicit queue to every message with a FIFO selection policy to store the data conveyed by the message before the target lifeline is ready to process it.

2.1.2 Local actions



(a) (b)

Figure 2.2: Local actions

We use another feature of sequence diagrams which is a unit of behaviors or an action within a lifeline in order to capture concise interactions. We show in Figure 2.2 examples of such actions. We call them *local actions* in the following. They are drawn as labeled boxes covering the lifelines. The local action $p_2 = 3$ is an *assignment action* performed on the port p_2. The assignment action is specific to our use of actions in sequence diagrams. We assume that it is executed atomically. Recall that we consider ports as particular variables. Unless modified locally, say by an assignment action, the value associated with a port is the last one received. In

figure 2.2a after the reception of the message m_1, the value 3 is associated with port p_1. Moreover, this same value will be associated with the port p_3 after the reception of the message m_2. In fact, messages carry the value contained on the emitting port and this value is stored on the receiving port. We do not make any assumption on the value associated with a port if none has been received or no local action has been performed yet on that port. This is quite unusual, as conventionally in sequence diagrams, no assumption at all is made on the value associated with a port. For this reason, we introduce local actions of the form $new(p)$ that may occur on lifelines and whose effect is to randomly associate a new value with p. We call such an instruction an *underspecification action*. This action allows us to distinguish messages consisting in plain data forwarding from those depicting unspecified value exchanges. We illustrate in Figure 2.2b the use of the underspecification action. The action is applied on port p_1. Then, the message m_2 provides p_3 with the random value assigned to p_1.

2.1.3 Combining operators

One may compose behaviors in sequence diagrams thanks to a number of combining operators. In this thesis, we consider three of them namely : the *loop*, *alt* and *strict* operators. Those operators are graphically associated with rectangles (covering portions of lifelines and messages), as in Figures 2.3—2.5. In the rest of the paper, we will call that range a *region* (it is an *operand* in UML parlance). In Figure 2.3, the region associated with the *loop* operator is called o as indicated optionally in the upper right part of the rectangle (this is an addition w.r.t. UML graphical notation, usually no sign is added to operands in the diagram).

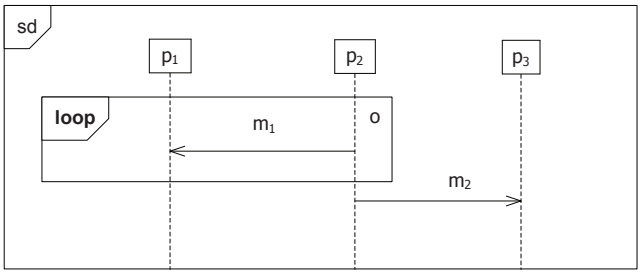


Figure 2.3: *loop* operator

• The *loop* operator is a repetition operator: all behaviors inside its region occur cyclically. The number of iterations is unknown beforehand and may be infinite. In the example, the message m_1 may occur many times before m_2 occurs or m_1 may never occur (*zero* iteration of the *loop*). When the number of iterations is finite, the execution of a lifeline behavior can leave the *loop* region, while it is perfectly possible that other lifelines still have some behaviors to be executed in that region. For example, we may have the following situation : p_2 sends twice m_1 then leaves region o and proceeds with the behavior outside (sending m_2). Due to the asynchronism of messages, it may happen that by the time p_2 sends m_2, p_1 has not received yet any of the occurrences of m_1. Recall that we use FIFO-buffering to encode asynchronous messages where the values issued by a sender are buffered and then consumed later by the receiver.

We use the loop operator without guard which corresponds implicitly according to the OMG specification, that the number of iteration is unknown beforehand (between 0 and $infinity$). The loop may not be executed at all or execute any number of times. These are excerpts from the OMG specification concerning this point:

> "[...] The loop operand will be repeated a number of times. The Guard may include a lower and an upper number of iterations of the loop as well as a *Boolean* expression. The

semantics is such that a loop will iterate minimum the *minint* number of times (given by the iteration expression in the guard) and at most the *maxint* number of times." (p. 472)

" If only loop, then this means a loop with $infinity$ upper bound and with 0 as lower bound." (p. 473)

Each lifeline may decide to leave the loop at its own. However, the number of iteration is the same for all lifelines which may quit the loop at different time instants (since each lifeline evolve at its own rate).

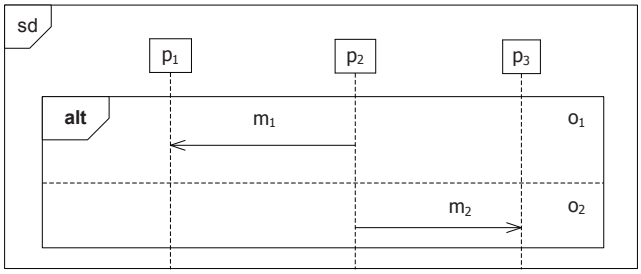


Figure 2.4: *alt* operator

• The *alt* operator illustrated in Figure 2.4 is a non deterministic choice among a set of possible behaviors. This is because we use it without explicit guards usually associated with the execution of behaviors : an implicit *true* guard is implied if the behavior has no guard according to the OMG specification. Those behaviors are depicted within sub-regions, horizontally delimited by dotted lines. Only one of the two sub-region behaviors occurs. In Figure 2.4, one such example is shown, where either the emission of m_1 or the emission of m_2 occurs. If a *loop* operator encloses the *alt* operator then the sub-region behaviors would be executed in any order within that loop (one of them per cycle) but no assumption is made concerning priorities between them. If a *loop* encloses the *alt* depicted in Figure 2.4 then executions may consist in many occurrences of m_1 before m_2 transits.

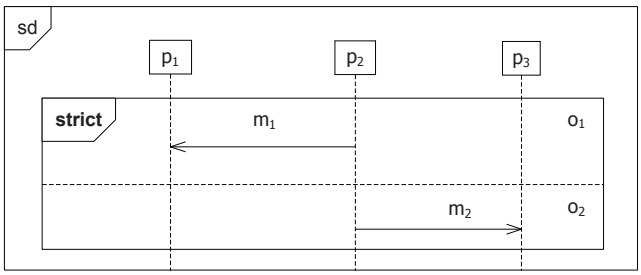


Figure 2.5: *strict* operator

• The *strict* operator allows to specify instants at which all lifeline behaviors are forced to leave a region. To do so, a synchronization point is introduced as an horizontal dotted line between two regions of a *strict* operator, as for example, between o_1 and o_2 in Figure 2.5. All execution fragments of the first region o_1 must be finished before any lifeline execution enters the second region o_2. In particular, thanks to the *strict* operator, the reception of m_1 by p_1 happens necessarily before the emission of m_2 by p_2. Note that by default, these two receptions have no implied ordering as discussed earlier about the Figure 2.1.

2.1.4 Data constraints

We distinguish two kinds of constraints : data constraints and timing constraints. Data constraints restrain the allowed values which transit between lifelines. While timing constraints define restraints on execution instants. An example of data constraint is given in Figure 2.6. It states that the value available on the port p_1 has an upper bound of 10 after the reception of the message m_1.

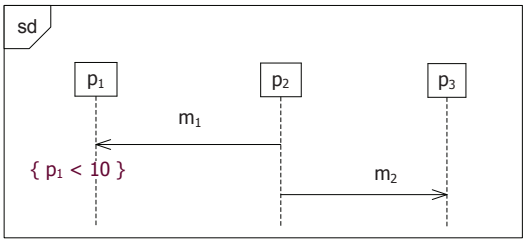


Figure 2.6: Data constraint

2.1.5 Timing constraints with MARTE

A typical example of the kind of timing constraint supported by our framework, is illustrated in Figure 2.7. They are a subset of those allowed by the *MARTE profile* from which we borrow the notations introduced by the *VSL* language. Typically our approach does not use *clock constraint language* [67, 38] to deal with time. This was done so as to reduce the assumptions imposed on the modeling of behavior (the same is true for our choice of using operators with no guards –alt,loop–).

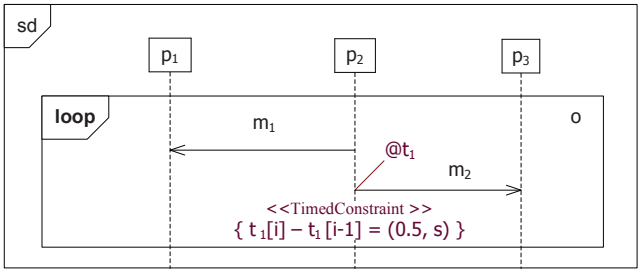


Figure 2.7: Timing constraint

Notations of the form $t_1[i]$ or $t_1[i-1]$ are introduced in the *VSL* language of the *MARTE* specification and are syntactic constructions capturing time instants. t_1 is a so-called *time observation* in UML (associated here with the sending of m_2). In VSL, a time expression denotes either a duration or a time instant. We consider rather timing expressions constraining instants. We interpret *time observations* as unbounded array variables capturing consecutive time instants. We designate these special variables *time variables*. Thus, $t_1[i]$ is the time instant of the i^{th} occurrence of m_2 stored at the index i of t_1. The constraint $t_1[i] - t_1[i-1] = (0.5, s)$ is used to signify that the delay between the previous occurrence of message m_2 (whose instant is stored at location $i-1$) and the current one (whose instant is stored at location i) is exactly $0.5s$ (s means seconds, we do not consider units of measurement in our analysis). The location i is implicitly incremented at each new occurrence of m_2. i is incremented several times because of the *loop* operator that defines an iterative behavior. The following excerpts show that our interpretation is compliant with the MARTE specification :

> "[...] One single instant or duration observation can be expressed with an occurrence

> index. For instance, we can express the "$i-th$" occurrence of a given event. [...] recurrent interaction fragments represented by a single sequence diagram, such as periodic or loop fragments may require time assertions comparing different instance traces of the sequence diagram. For instance, the duration between the i-th and i+1-th occurrence of an event that triggers a periodic scenario." (p. 433)
>
> " Index i enables comparisons between different occurrences of the same event that may not be consecutive (e.g. burstiness)." (p. 434)
>
> " $t1[i]$ returns the instant time of an event observation $t1$ declared in a UML model element *time observation*. The index i is a modifier that indicates that the instant time refers to whatever of the occurrences of the observed event." (p. 459)
>
> " $(t1[i+1]-t1[i])$ returns the duration between any two successive occurrences of an observed event whose occurrence instants are labeled by $t1$." (p. 459)

Notice that we rewrite constraints as following : $t_1[i+1]-t_1[i]=0.5$ becomes $t_1[i]-t_1[i-1]=0.5$. Anyway, the MARTE specification does not indicate what happens when an occurrence index is not defined yet. Typically, it is not clear for example when does the first occurrence happen considering the constraint $t_1[i]-t_1[i-1]=0.5$ on the occurrences of m_2. In full generality, if $t[term]$ appears in a constraint expression where t is a time variable and $term$ is any integer term denoting a location in t then whenever $t[term]$ is not defined we consider that the constraint is true. This makes for instance the first occurrence of m_1 happens at any time (since $t_1[-1]$ is not defined in the evaluation of $t_1[0]-t_1[-1]=0.5$).

2.2 UML Metamodel

We discuss in this section the metamodel architecture of UML which is related to our use of sequence diagrams. In fact, the UML metamodel as a whole consists of a variety of modeling elements to describe the system structure and behavior from multiple views. In this context, the sequence diagram is a high level view of system behavior as interactions between entities of the system. The metamodel provides a syntax for sequence diagrams combined with graphic and natural language description.

2.2.1 Modeling elements

The UML metamodel of a sequence diagram defines modeling elements and relations between them. It is shown in figure 2.9. In the following, the main constructs of the metamodel are discussed by incrementally making a focus on some parts of it for illustration. In figure 2.8, below the line we introduce the main modeling elements of UML which are useful to define sequence diagrams. For each of them, above the line, we introduce either their graphical denotation if it is unique or an example when the element is more abstract and corresponds to several concrete notations.

The graphical elements above the line are indeed the representation of the concepts below the line introduced in the UML metamodel to describe interactions. An *Interaction* element corresponds to the frame of the sequence diagram itself. *Lifeline*, *Message* and *CombinedFragment* of the metamodel correspond respectively to a lifeline, a message and a combining operator in the sequence diagram. The element *OccurrenceSpecification* denotes a point on the lifeline when an execution occurs : E.g. the message m_1 reception by the lifeline of p_2 defines a point in the diagram which is the intersection between the head of the arrow of m_1 and the vertical line of the lifeline of p_2. Finally, the element *ExecutionSpecification* corresponds to a local action. A local

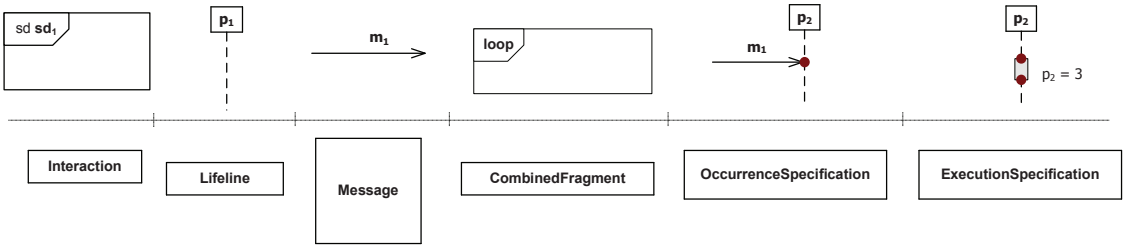


Figure 2.8: Modeling elements and graphical denotations

action is identified by two points on the lifeline that is : an *OccurrenceSpecification* denoting the beginning of the action and another one when it ends. This precise identification is not relevant for our analysis because we assume the execution of a local action to be instantaneous.

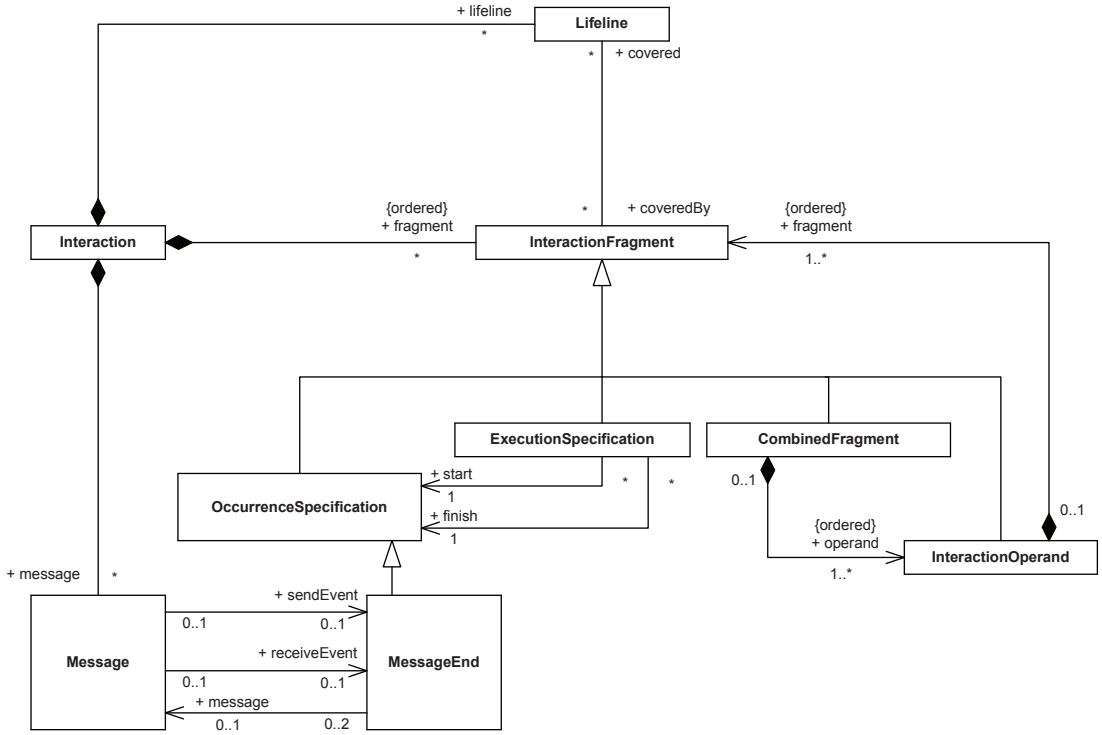


Figure 2.9: UML sequence diagram metamodel: a simplified view

2.2.2 Element relationships

When two elements in the metamodel are related then a line is drawn between them (see all the lines linking boxes in Figure 2.9). There are different kinds of relationships:

• If the line ends with a solid white arrowhead then the source element inherits all the relationships (and properties) defined in the target element. In this case, the relation is called *generalization*. Let us comment the generalization relationship linking elements in Figure 2.10.

The element *InteractionFragment* is introduced to generalize among others the elements *CombinedFragment*, *OccurrenceSpecification* and *ExecutionSpecification* (see Figure 2.10a). In turn, an *OccurrenceSpecification* is an generalization of the element *MessageEnd* (see Figure 2.10a) when it denotes a point on the lifeline of a reception or an emission of a message. This kind of relation is used to factorize the metamodel.

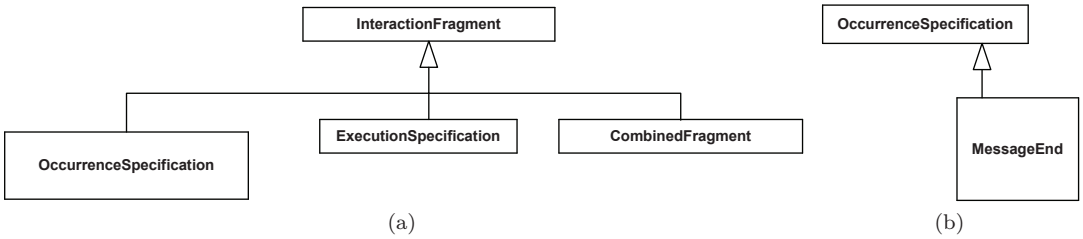


Figure 2.10: Generalization

• When it is not a generalization, the line denotes a relation, called *association*, which may be annotated by roles. Note that the association line may be annotated also by multiplicities. A multiplicity denotes the number of individuals/instances of the target or source element which may participate in the association. Figure 2.11 depicts examples of associations.

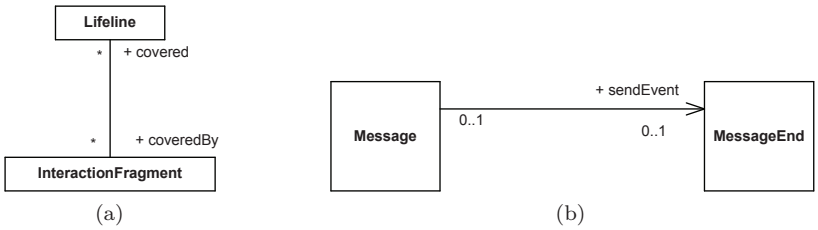


Figure 2.11: Association

A *Lifeline* may be associated with a set of *InteractionFragments* (has a multiplicity $*$) : We say a *Lifeline* is *covered by* each *InteractionFragment*. Conversely, an *InteractionFragment* covers a set of *Lifelines* (see Figure 2.11a). This is the case when the *InteractionFragment* denotes a combining operator and thus some/all lifelines may traverse the regions of that operator. The kind of association discussed previously is bidirectional. But, if the line of the association ends with an open arrowhead then the association maps the source element to the target element and said to be *navigable from the source element*. The *Message* is associated with at most (multiplicity 0..1) one *MessageEnd* with the role *sentEvent* (see Figure 2.11b). Recall that the latter is actually a point on the lifeline and thus when a message is not be emitted by a lifeline, it comes from the environment of a sequence diagram (hence the zero multiplicity).

When the association line ends with a filled-in diamond then the element at the opposite side of the diamond exists only if the the source element exists and is said to be a composition :

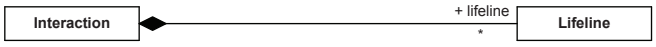
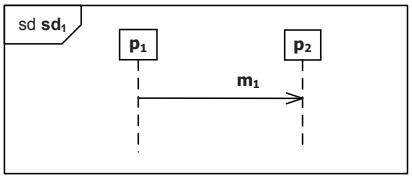


Figure 2.12: Composition

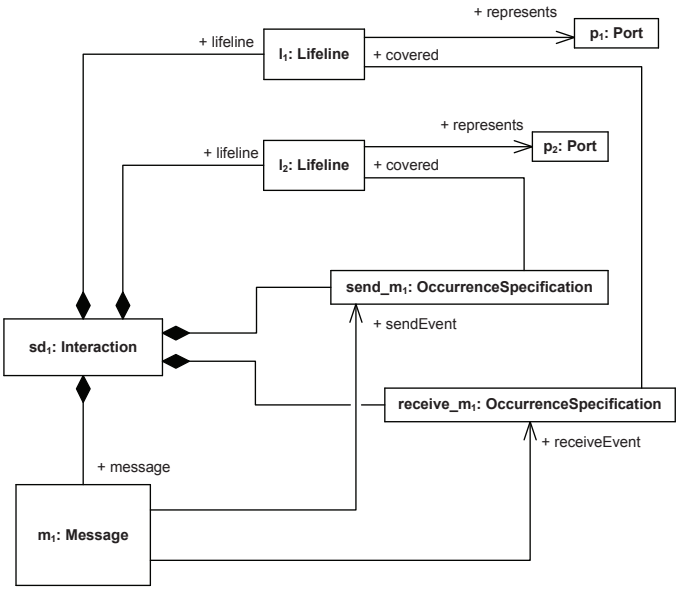
The Figure 2.12 indicates that an *Interaction* is composed of a set of lifelines (considering the multiplicity $*$). Recall the metamodel illustration of a sequence diagram in figure 2.9. The main element of a sequence diagram is an *Interaction* (represented with the top most box on the left). Besides the set of *Lifeline* elements, an *Interaction* is composed, following the outgoing association lines with diamonds, as well of a set of *Message* and a set of *InteractionFragment* elements.

Example 1 *An elementary sequence diagram is given in Figure 2.13 together with its structure using the metamodel elements. The* Interaction sd_1 *is composed of two* Lifeline *elements* l_1, l_2 *associated respectively with ports* p_1, p_2*. Also,* sd_1 *has a set of* Message *elements, here reduced to the singleton containing* m_1*. The set of fragments composing* sd_1 *contains two* OccurrenceSpecifi-

cation *elements named send_m_1 and receive_m_1 corresponding respectively to the emission and reception of the message m_1.*



(a)



(b)

Figure 2.13: An elementary sequence diagram and its structure with metamodel elements

Conclusion

We have presented the syntax and semantics of the subset of sequence diagrams as a part of the UML language that will be considered in the rest of this thesis. This subset is quite complete and includes the main commonly used constructs of the language. We have also discussed our usage of sequence diagrams in terms of semantics choices and restrictions. As we have seen from some excerpts of the OMG specification of UML, the semantics of sequence diagrams is defined informally using natural language. In this thesis, we give an operational semantics to sequence diagrams by mapping them on a particular kind of automata TIOSTS (that will be discussed in Chapter 5). For that purpose, we introduce a concise formalized textual representation of sequence diagrams which reflects the metamodel in Chapter 4. Links between this textual representation used in the semantics attribution and the metamodel are stated in the implementation Chapter 9 where is discussed the model transformation which generates automata from sequence diagrams.

Before the formalization of sequence diagrams, mathematical preliminaries will be given in the next chapter in order to be used for the formalization and also for the definition of TIOSTS automata.

Chapter 3

Formal preliminaries

Contents

3.1 Typed equational logic

As glimpsed in the introduction, sequence diagrams introduce pieces of data denoted in a symbolic manner. In the following, we use the classical typed equational logic [1] to represent and reason about data according to the UML and MARTE standards. Herein we introduce this logic whose syntactic part will be the basis of sequence diagram textual representation introduced later (in Chapter 4). Moreover, this logic will be used also in the next section as the mean to define data in IOSTS and later in TIOSTS (refer to chapter 5). In a classical manner, we begin by presenting *syntax* of the logic, and then we define the mathematical meaning of that syntax, that is its associated semantics.

3.1.1 Syntax

The syntax of Typed Equational Logic is defined is several steps. The first step consists in defining a syntactical structure used to declare function symbols and types. Such a structure is called a *signature*. A signature is simply a couple whose first component is a set of type names and the second component is a set of typed function names. In order to represent types associated with a function, each function name is provided with a profile consisting in a sequence of type names.

Definition 1 (Signature) *A (data type) signature is a pair $\Omega = (S, Op)$ where S is a set of type symbols and Op is a set of function names, each one provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$, $s_i \in S$).*

A function name of the form f associated with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ is latter denoted $f : s_1 \cdots s_{n-1} \rightarrow s_n$ and represents a function taking $n-1$ arguments of respective types $s_1 \cdots s_{n-1}$ and computing a value of type s_n. A function name of the form $f :\rightarrow s_n$ denotes a constant value of type s_n.

[1] The *Typed Equational Logic* restricts the first order logic to the only use of the predicate equality (=).

Example 2 (Primitive data types) *We use the usual basic data types in the sequence diagram for conveyed values. Number of primitive types have been defined by the specification of the UML standard. These include primitive types such as Integer and Boolean.*

The signature $\Omega_{Integer} = (S_{Integer}, Op_{Integer})$ *is associated with the specification of integer arithmetic.*

- $S_{Integer} = \{Integer, Boolean\}$
- $Op_{Integer} = \{0 :\rightarrow Integer,$
  $true :\rightarrow Boolean,$
  $false :\rightarrow Boolean,$
  $succ : Integer \rightarrow Integer,$ *(successor)*
  $pred : Integer \rightarrow Integer,$ *(predecessor)*
  $+ : Integer \times Integer \rightarrow Integer,$ *(addition)*
  $- : Integer \times Integer \rightarrow Integer,$ *(subtraction)*
  $* : Integer \times Integer \rightarrow Integer,$ *(multiplication)*
  $<: Integer \times Integer \rightarrow Boolean\}$ *(inequality less than).*

Note that noted $\Omega_{Integer}$ to signify that it is a signature where one of the sorts is $Integer$.

Example 3 (FIFO queue) *A queue is a FIFO (First In, First Out) structure. Any value inserted first, will be the first to be consumed. We use the queuing to capture underlying communication mechanism in sequence diagrams between component of the system where the exchanged messages will be stored in the queue in the same order of receipt. The signature* $\Omega_{Queue} = (S_{Queue}, Op_{Queue})$ *is such that :*

- $S_{Queue} = S_{Integer} \cup \{Queue\}$
- $Op_{Queue} = Op_{Integer} \cup \{emptyQueue :\rightarrow Queue,$
  $top : Queue \rightarrow Integer,$
  $pop : Queue \rightarrow Queue,$
  $push : Queue \times Integer \rightarrow Queue\}.$

The second step of the syntax definition consists in representing executions over functions whose names are declared in a signature. Such executions are represented as so-called *terms* which are defined as follows over a signature and a set of variables.

Definition 2 (Term) *Let* $\Omega = (S, Op)$ *be a signature and* $V = \cup_{s \in S} V_s$ *be a set of so-called typed variables satisfying* $\forall s, s' \in S,\ s \neq s' \Rightarrow V_s \cap V_{s'} = \emptyset$. *The* set of Ω-terms with variables in V *is denoted* $T_\Omega(V) = \cup_{s \in S} T_\Omega(V)_s$ *and is inductively defined as follows:*

- *if* $x \in V_s$ *then* $x \in T_\Omega(V)_s$,
- *if* f *has a profile* $\rightarrow s_n$ *then* $f \in T_\Omega(V)_{s_n}$,
- *if* f *has a profile* $s_1 \cdots s_{n-1} \rightarrow s_n$ *and* $(t_1, \ldots, t_{n-1}) \in T_\Omega(V)_{s_1} \times \ldots \times T_\Omega(V)_{s_{n-1}}$ *then* $f(t_1, \ldots, t_{n-1}) \in T_\Omega(V)_{s_n}$.

Example 4 *Using the signature* $\Omega_{Integer} = (S_{Integer}, Op_{Integer})$, *let us consider the typed variable names* $V = V_{Boolean} \cup V_{Integer}$ *where* $V_{Boolean} = \emptyset$ *and* $V_{Integer} = \{x, y\}$. *The following are some* Ω*-terms with variables in* V *(*$T_{\Omega_{Integer}}(V)$*):*

terms 0, x, y, $succ(0)$, $pred(0)$,$succ(x)$, $succ(y)$, $-(0,0)$ *and* $*(x,y)$ *are in* $\in T_{\Omega_{Integer}}(V)_{Ineteger}$; *and the term is in* $T_{\Omega_{Integer}}(V)_{Boolean}$.

Now we can define the formulas associated with a signature and a set of variables: they denote properties concerning executions. Basic formulas are simply equalities between executions (*i.e.* terms) and more complex formulas are obtained by connecting formulas by means of the conjunction or disjunction operators, or by the negation of a formula.

Definition 3 (Formula) *Let (S, Op) be a signature and V be a set of variables typed in S. The* set $Sen_\Omega(V)$ of typed equational Ω-formulas over V *is inductively defined as follows:*

- $True$ *and* $False$ *are in* $Sen_\Omega(V)$,
- *for any* s *in* S, *for any* t *and* t' *in* $T_\Omega(V)_s$, *we have* $t = t'$ *is in* $Sen_\Omega(V)$,
- *for any* φ_1 *and* φ_2 *in* $Sen_\Omega(V)$, *we have* $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ *and* $\neg\varphi_1$ *are in* $Sen_\Omega(V)$,
- *for any* x *in* V *and* φ *in* $Sen_\Omega(V)$, *we have* $\forall x\varphi$ *and* $\exists x\varphi$ *are in* $Sen_\Omega(V)$.

Example 5 *Based on the signature and terms of Examples 2 and 4, we can define the following formulas:*

$x = y$, $\neg(0 = succ(x))$, $x + 0 = x$, $succ(x + y) = x + succ(y)$.

Notation 1 *In the sequel* $Sen^{qf}_\Omega(V)$ *(qf stands for quantifier free) is the subset of* $Sen_\Omega(V)$ *such that its elements contain no occurrences of* $\forall$ *and* $\exists$.

The syntax of the typed equational logic is then fully characterized by the set of all possible triples $(\Omega, V, Sen_\Omega(V))$ that can be built by means of the previous definitions. In the remaining of the Section we characterize some syntactical operations that will be useful in the sequel. We now define the notion of substitution which is used to assign computations to variables. Substitutions simply consist in functions associating terms with variables while preserving types.

Definition 4 (Substitution) *A* Ω-substitution over V *is a function* $\sigma : V \to T_\Omega(V)$ *preserving types, that is, associating with each variable* v *of type* s, *a term* $t \in T_\Omega(V)$ *also of type* s. *In the following, we note* $T_\Omega(V)^V$ *the set of all* Ω*-substitutions of the variables* V . *Any substitution* σ *may be canonically extended to terms (with* $\sigma(f(t_1, \ldots, t_{n-1})) = f(\sigma(t_1), \cdots, \sigma(t_{n-1}))$*).*

Example 6 *Consider now the signature* $\Omega_{Integer}$ *and the set of variables* $V_{Integer} = \{x, y\}$. *We can define the following* $\Omega_{Integer}$*-substitution* $\sigma : V_{Integer} \to T_{\Omega_{Integer}}(V_{Integer})$ *such that* $\sigma(x) = x + 1$ *and* $\sigma(y) = y + 1$. *For example, we have then* $\sigma(x + y) = (x + 1) + (y + 1)$.

Finally we give a notation useful to update values of some variables in a substitution.

Notation 2 *The identity* Ω*-substitution over the variables* V , id_V , *is defined as* $id_V(v) = v$ *for all* $v \in V$

Let $x_1 \cdots x_n$ *be variables of respectively* $V_{s_1} \cdots V_{s_n}$, *let* $t_1 \cdots t_n$ *be terms of respectively*

$T_\Omega(V)_{s_1} \cdots T_\Omega(V)_{s_n}$. $[(x_i \leftarrow t_i)_{i \leq n}]$ *is the substitution* $T_\Omega(V)^V$ *such that for all* $j \leq n$ *we have* $[(x_i \leftarrow t_i)_{i \leq n}](x_j) = t_j$ *and for all* $y \in V \setminus \{x_1, \cdots, x_n\}$ *we have* $[(x_i \leftarrow t_i)_{i \leq n}](y) = y$.

3.1.2 Semantics

Semantics of Typed equational logic is based on the notion of *model*. A model associated with a signature is a mathematical structure used to interpret all symbols of the signature. It is defined

as a set whose elements are typed data and which is provided with a function for each function name of the signature.

Definition 5 (Model) *A* Ω***-model*** *is a set* M *whose elements are associated with a type in* S*, and we note* $M_s \subseteq M$ *the subset of* M *whose elements are associated with* s*. Each* $f : s_1 \cdots s_n \rightarrow s \in Op$*, is interpreted as a function* $f_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_s$*.*

Models give a semantical counterpart to type and function names. We now introduce the notion of interpretation *to give a semantical counterpart to variables.*

Definition 6 (Interpretation) *We define* Ω***-interpretations*** *as applications* ν *from* V *to* M *preserving types and extended to terms in* $T_\Omega(V)$*.* M^V *is the set of all* Ω*-interpretations of* V *in* M*. Any interpretation* ν *can be extended to terms in a canonical way* $\nu(f(t_1, \ldots, t_n)) = f_M(\nu(t_1), \ldots, \nu(t_n))$*.*

We now proceed to define the notion of *satisfaction of a formula.* Roughly, an Ω-model being given, the satisfaction relation is a mathematical relation associating interpretations and formulas. A variable interpretation is associated with a formula whenever that formula is true for the interpretation.

Definition 7 ***An interpretation*** ν ***satisfies a formula*** φ *Let* $\Omega = (S, Op)$ *be a signature and* V *be a set of variables typed in* S*. Let* M *be a* Ω*-model. For any* $\nu \in M^V$ *and* $\varphi \in Sen_\Omega(V)$*, we say that* ν satisfies φ *denoted* $\nu \models \varphi$ *if and only if:*

- *if* φ *is* $True$ *then we have* $\nu \models \varphi$
- *if* φ *is* $False$ *then we do not have* $\nu \models \varphi$
- *whenever* φ *is of the form* $t = t'$*, we have* $\nu(t) = \nu(t')$*,*
- *whenever* φ *is of the form* $\neg\psi$*, we do not have* $\nu \models \psi$*,*
- *whenever* φ *is of the form* $\varphi_1 \wedge \varphi_2$*, we have* $\nu \models \varphi_1$ *and* $\nu \models \varphi_2$*,*
- *whenever* φ *is of the form* $\varphi_1 \vee \varphi_2$*, we have* $\nu \models \varphi_1$ *or* $\nu \models \varphi_2$*,*
- *whenever* φ *is of the form* $\forall x\psi$*, we have for all* ν' *such that for all* y *in* $V \setminus \{x\}$*,* $\nu'(y) = \nu(y)$*, we have* $\nu' \models \psi$*,*
- *whenever* φ *is of the form* $\exists x\psi$*, we have exists* ν' *such that for all* y *in* $V \setminus \{x\}$*,* $\nu'(y) = \nu(y)$*, we have* $\nu' \models \varphi$*,*

3.2 Input Output Symbolic Transition System

Herein we present Input Output Symbolic Transition Systems (IOSTS). IOSTS are widely employed in black box testing based on symbolic techniques [1, 36, 33, 32, 30, 34]. IOSTS are symbolic automata used to specify behaviors of reactive systems expressed as reactions by producing outputs to external stimuli. They form the basic formalism that we will extend in Chapter 5 to give an operational semantics to sequence diagrams in Chapter 6.

3.2.1 IOSTS syntax

IOSTS are defined over a *IOSTS signatures* which are used to introduce particular variables whose valuations define states of the IOSTS and to introduce the so-called *communication channels* through which values may be received or emitted. In the sequel, we suppose that a datatype signature $\Omega = (S, Op)$ is given. IOSTS signatures are defined as follows.

Definition 8 (*IOSTS* signature) *A IOSTS signature Σ is defined as a couple (A, C) where A is a set of variables of the form $\cup_{s \in S} A_s$ such that for all s and s' in S $s \neq s' \Rightarrow A_s \cap A_{s'} = \emptyset$. Variables of A are called* attribute variables *and elements of C are called* communication channels.

We now define the *communication actions* over channels. Communication actions can be inputs or outputs sent by communication channels, or they can be *internal actions*. Internal actions represent operations that do not involve any communication with the environment. They are all represented by a generic symbol τ.

Definition 9 (Communication actions) *Let Σ be an IOSTS signature.*
The set of communication actions *over Σ is defined as $Act(\Sigma) = I(\Sigma) \cup O(\Sigma) \cup \{\tau\}$, where:*

- $I(\Sigma) = \{c?x \mid x \in A, c \in C\}$
- $O(\Sigma) = \{c!t \mid t \in T_\Omega(A), c \in C\}$

Elements of $I(\Sigma)$ are called *inputs* and those of $O(\Sigma)$ are called *outputs*.

*IOSTS*s are composed of a set of *states*, an *initial state*, and *transitions* going from one state to another. Transitions are composed of: *guards*, which are conditions that have to be satisfied in order to fire the transition; *communication actions*, introduced in Definition 33; and *affectations*, representing the modifications on the attribute variables when firing the transition.

Definition 10 (*IOSTS*) *Let $\Sigma = (A, C)$ be an IOSTS signature.*
An IOSTS *over Σ is a tuple $\mathbb{G} = (Q, init, T)$ where:*

- *Q is a set of* state names
- *$init \in Q$ is the* initial state
- *$T \subseteq Q \times Sen_\Omega^{qf}(A) \times Act(\Sigma) \times T_\Omega(A)^A \times Q$ is a set of* transitions

Notation 3 *In the following, for any IOSTS $\mathbb{G}$ of the form $(Q, init, T)$ over Σ, we use the notations $state(\mathbb{G})$, $init(\mathbb{G})$, and $Trans(\mathbb{G})$ in order to refer, respectively, to Q, $init$, and T.*

In the same way, for any transition $tr \in Trans(\mathbb{G})$ of the form $(q, \varphi, act, \rho, q')$ we use the notations $source(tr)$, $guard(tr)$, $act(tr)$, $sub(tr)$, and $target(tr)$ in order to refer, respectively, to q, φ, act, ρ, and q'.

Example 7 *We represent an IOSTS in the standard way, that is, by a directed, edge-labeled graph where nodes represent states and edges represent transitions. Transitions are represented with an arrow $\rightarrow$ representing the flow of the communication from their source state to their target state.*

An example of an IOSTS is shown in Figure 3.1. It is defined over the signature $\Sigma = (A, C)$ where $A = \{x_{env}, x_{cmd}, x_{data}\}$ and $C = \{env, cmd, data\}$. Consider for example the transition

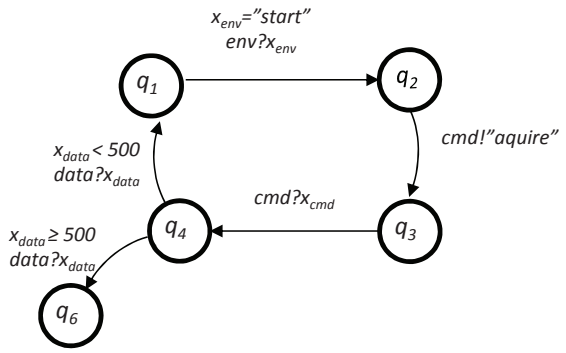


Figure 3.1: IOSTS $\mathbb{G}$

of $\mathbb{G}$ *which goes from* q_4 *to* q_1 *with the input* data?x_{data}*. It denotes the reception of a value on channel data and stored in the attribute variable* x_{data}*. The execution of the transition is constrained by the guard* $x_{data} < 500$*.*

3.2.2 IOSTS behavior

The behaviors of an $IOSTS$, also called its *semantics*, is defined by the notion of the interpreted traces that can be generated from it. Traces are possible successions of communication actions that are specified by an $IOSTS$. However, those succession of communication actions are to be interpreted in order to get real values. We suppose a Ω-model M is given. Therefore, we give a series of definitions that are needed in order to define the behavior of an $IOSTS$.

We start by defining the notion of *concrete actions*, which are the interpretations of the communication actions.

Definition 11 (Concrete actions) *Let* $\Sigma = (A, C)$ *be an* $IOSTS$ *signature.*

The set of concrete actions *over* Σ *is the defined as* $Act_M(\Sigma) = I_M(\Sigma) \cup O_M(\Sigma) \cup \{\tau\}$*, where:*

$I_M(C) = \{c?v \mid c \in C, v \in M\}$

$O_M(C) = \{c!v \mid c \in C, v \in M\}$

The value v is the interpretation of the received or emitted terms.

Traces of an $IOSTS$ are built from sequences of transitions. The semantics of an $IOSTS$ is the semantics that we give to the transitions.

Definition 12 (Semantics of a transition) *Let* $\mathbb{G} = (Q, init, T)$ *be an* $IOSTS$ *over* Σ*. The* semantics of a transition $tr \in T$ *of the form* $(q, \varphi, act, \rho, q')$ *is the relation* $Run(tr) \subseteq M^A \times Act_M(\Sigma) \times M^A$*, such that* $(\nu_i, act_M, \nu_f) \in Run(tr)$ *if and only if:*

- *if act is of the form* $c!t$*, then* $\nu_i \models \varphi$*,* $\nu_f = \nu_i \circ \rho$ *and* $act_M = c!\nu_i(t)$
- *if act is of the form* $c?x$*, then* $\nu_i \models \varphi$*, there exists* ν_a *such that* $\nu_a(z) = \nu_i(z)$ *for every* $z \neq x$*,* $\nu_f = \nu_a \circ \rho$*, and* $act_M = c?\nu_a(x)$
- *if act is of the form* τ *then* $\nu_i \models \varphi$*,* $\nu_f = \nu_i \circ \rho$ *and* $act_M = \tau$

Notation 4 *In the following, Run(tr) stands for the* run of a transition *and, for any run r of Run(tr) of the form (ν_i, act_M, ν_f), we use the notations source(r), act(r), and target(r) in order to refer, respectively, to ν_i, act_M, and ν_f.*

The application ν_i is the interpretation of variables *before* executing the transition, and ν_f is the interpretation of the variables *after* the execution of the transition. act_M is the interpretation of either the value sent or received in the communication action of the transition or the internal action τ.

Example 8 *Consider again the IOSTS $\mathbb{G}$ depicted in Figure 3.1. Recall that $\mathbb{G}$ is defined over $\Sigma = (A, C)$ where $A = \{x_{env}, x_{cmd}, x_{data}\}$ and $C = \{env, cmd, data\}$. Consider in particular the transition going from q_4 to q_1: $q_4 \xrightarrow{x_{data}<500 \quad data?x_{data} \quad id_A} q_1$. This is a possible run of the transition: $\nu_i \xrightarrow{data?200} \nu_f$, where ν_i and ν_f are defined as follows:*

interpretation of variables
$\nu_i(x_{env}) =$ "*start*", $\nu_i(x_{cmd}) =$ "*ack*" $\nu_i(x_{data}) = 999$
$\nu_f(x_{env}) =$ "*start*", $\nu_f(x_{cmd}) =$ "*ack*" $\nu_f(x_{data}) = 200$

Paths are sequences of transitions beginning at the initial state of the *IOSTS*.

Definition 13 (Paths of an *IOSTS*) *Let $\mathbb{G} = (Q, init, T)$ be an IOSTS over Σ. The set of* paths*, denoted Path($\mathbb{G}$), contains all the finite sequences $tr_1 \cdots tr_n$ of transitions of T such that:*

- *$source(tr_1) = init$*
- *for every i, $1 \leq i < n$, $target(tr_i) = source(tr_{i+1})$*

The *run of a path* is the sequence of runs of the transitions in the path, where the target state shares the variable interpretation with the source state of the consecutive transitions.

Definition 14 (Runs of paths) *Let $\mathbb{G}$ be an IOSTS over Σ. The set of* runs of a path *p, denoted Run(p), for a path $p = tr_1 \cdots tr_n$ in Path($\mathbb{G}$), are sequences $r_1 \cdots r_n$ such that:*

- *for all $i \leq n$, r_i is a run of tr_i, $r_i \in Run(tr_i)$*
- *for all $i < n$, $target(r_i) = source(tr_{i+1})$*

Now we define how to extract traces from a path.

Definition 15 (Concrete traces) *Let $\mathbb{G}$ be an IOSTS over Σ, and let $p \in Path(\mathbb{G})$. The set of* concrete traces *of a path p, denoted traces(p), is the set $\bigcup_{r \in Run(p)} \{traces(r)\}$, where traces(r) is inductively defined as follows:*

- *if r is ε, then $traces(r)$ is ε*
- *if p is of the form $p'.tr$ and r is of the form $r'.a$, where $r' \in Run(p')$ and $a \in Run(tr)$, then:*
  - *if $act(a)$ is τ, we have $traces(r) = traces(r')$*

– *if* $act(a)$ *is not* τ*, we have* $traces(r') = traces(r).act(a)$

Finally, the behaviors of an $IOSTS$, also called its *semantics*, are defined as the set of all the concrete traces that can be obtained from its paths.

Definition 16 (Semantics of an $IOSTS$) *Let* $\mathbb{G}$ *be an* $IOSTS$ *over* Σ*. The semantics of* $\mathbb{G}$ *is defined as* $Traces(\mathbb{G}) = \bigcup_{p \in Path(\mathbb{G})} traces(p)$*.*

Example 9 *The trace* $env?"start".cmd!"aquire".cmd?"ack".data?200$ *is in the semantics of* $\mathbb{G}$*.*

3.3 TIOLTS

Timed Input Output Labeled Transition Systems (TIOLTS or TIOTS) [21, 81, 55, 31]: they are simply automata whose transitions are labeled either by actions (inputs, outputs, or the internal action τ) or by delays. We have however slightly adapted the actions format of TIOLTS in order to better fit our needs of assigning semantics to sequence diagrams. First, inputs and outputs introduce channel names as in the case of IOSTS. Values exchanged between a TIOLTS and its environment are denoted as elements of a model M of a signature Ω that are considered given in the sequel of this section.

We begin by defining the actions occurring in TIOLTS.

Definition 17 (TIOLTS actions) *Let* C *be a set whose elements are called channels.*

The set of communication actions *over* C*, denoted* $Act_M(C)$*, is the set* $I_M(C) \cup O_M(C) \cup \{\tau\}$*, where:*

- $I_M(C) = \{c?v \mid v \in M, c \in C\}$
- $O_M(C) = \{c!v \mid v \in M, c \in C\}$

$c?v$ denotes the reception of a value v on channel c, $c!v$ denotes the emission of the value v on channel c and τ denotes the unobservable action. Elements of $I_M(C)$ (respectively $O_M(C)$) are called inputs *(respectively* outputs*).*

Transitions of a TIOLTS may introduce durations that may be denoted as integers or real numbers. In the sequel, we note I the type introduced in Ω to handle those durations. We suppose that Ω introduces an addition over durations $+ : I.I \to I$. M_I is either isomorphic to natural numbers or real numbers and is left implicit in the sequel. Moreover, $+ : I.I \to I$ is associated with $+_M : M_I \times M_I \to M_I$ the usual addition (either on integers or reals). In the following for readability sake, we note by abuse $+ : M_I \times M_I \to M_I$ instead of $+_M : M_I \times M_I \to M_I$.

Now TIOLTS are simply defined as triples introducing a set of states, an initial state and a set of transitions as follows :

Definition 18 (TIOLTS) *Let* C *be a set of channels. A* Timed Input/Output Labeled Transition System ($TIOLTS$) over C *is is a triple* (Q, q_0, T) *where:*

- Q *is a set of* states
- $q_0 \in Q$ *is the* initial state

- $T \subseteq Q \times Act_M(C) \cup M_I \times Q$ *is a set of* transitions

Notation 5 *For any TIOLTS* $\mathbb{A} = (Q, q_0, T)$ *over* C *and for any transition* $tr \in T$ *of the form* (q, act, q')*, we use the notations* $state(\mathbb{A})$*,* $init(\mathbb{A})$*,* $Trans(\mathbb{A})$*,* $Chan(\mathbb{A})$*,* $source(tr)$*,* $target(tr)$ *and* $act(tr)$ *in order to refer, respectively, to* Q*,* q_0*,* T*,* C*,* q*,* q' *and* act*.*

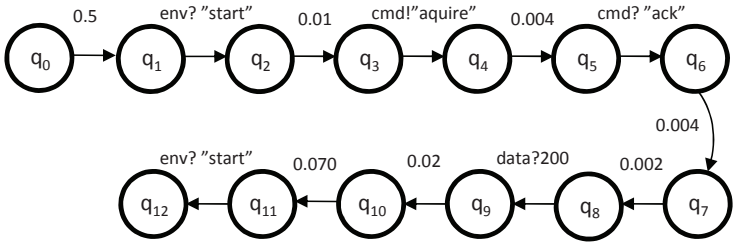


Figure 3.2: TIOLTS $\mathbb{A}$

Example 10 *We show an example of a TIOLTS in Figure 3.2. Note that it is defined over the set of channels* $C^{ctrl} = \{env, cmd, data\}$*. Consider for example the transition of* $\mathbb{A}$ *which goes from* q_0 *to* q_1 *with the concrete action as a delay* 0.5*, concrete in the sense that* 0.5 *is in* M_I *(not symbolic in* I*). It denotes time passing. Now consider the transition going from* q_8 *to* q_9 *labeled with the action data*?200*. It denotes the reception of the value* 200 *on the channel data.*

3.3.1 Trace semantics

TIOLTS specify sequences of actions separated by durations. Those sequences are called *timed traces*. The definition of a timed trace is based on the notion of paths of a TIOLTS which corresponds to a sequence of consecutive transitions.

Definition 19 (Paths of a TIOLTS) *Let* $\mathbb{A} = (Q, q_0, T)$ *be an TIOLTS over* C*. The* set of finite paths of $\mathbb{A}$*, denoted* $FP(\mathbb{A})$ *is the set of all sequences of transitions of* T *such that* $tr_1 \cdots tr_n \in FP(\mathbb{A})$ *if and only if:*

- $source(tr_1)$ *is* q_0*,*
- *for all* $i < n$ *we have* $target(tr_i) = source(tr_{i+1})$*.*

Any path of a TIOLTS can be associated with a so called trace that is simply defined as the sequence of actions and durations introduced in the path. Traces of a TIOLTS are then defined as the set of all traces of all its finite paths.

Definition 20 (Traces of a TIOLTS) *Let* $\mathbb{A} = (Q, q_0, T)$ *be a TIOLTS over* C*. Let* $fp \in FP(\mathbb{A})$*. The* trace of fp *denoted* $trace(fp) \in (Act_M(C) \cup M_I)^*$ *is defined as follows:*

- *if* fp *is* ε *then* $trace(fp)$ *is* ε*,*
- *if* fp *is of the form* $fp'.tr$ *where* tr *is a transition then:*
  - *if* $act(tr)$ *is of the form* $c!v$ *or* $c?v$ *or* d *then* $trace(fp)$ *is* $trace(fp').act(tr)$*,*
  - *if* $act(tr)$ *is* τ *then* $trace(fp)$ *is* $trace(fp')$*.*

The set of traces of $\mathbb{A}$*, denoted* $Traces(\mathbb{A})$*, is the set* $\bigcup_{fp \in FP(\mathbb{A})} \{trace(fp)\}$*.*

Example 11 *The following trace is associated with the path starting in q_0 and going to q_{12} in the TIOLTS depicted in Figure 3.2:*

$(0.5).env?"start".(0.01).cmd!"acquire".(0.004).cmd?"ack".(0.004).(0.002).data?200.(0.02).(0.07).env?"start"$

In the sequel, we need to be able to represent any given duration as any decomposition of small durations. For example, the duration 0.5 may be decomposed as the sum of delays 0.3 and 0.2 since $0.5 = 0.3 + 0.2$. It may also be represented as $0.1 + 0.1 + 0.1 + 0.2$, etc. In order to take into account all such durations, we define *timed traces* of a TIOLTS as the set of all sequences obtained by applying arbitrary decompositions of durations in any trace of the TIOLTS.

Definition 21 (Timed traces of a TIOLTS) *For any finite path fp in $FP(\mathbb{A})$, the set of* timed traces of *fp denoted $ttraces(fp)$ is defined as follows:*

- *$trace(fp)$ in $ttraces(fp)$*
- *for any timed trace in $ttraces(fp)$ of the form $\sigma.d_1.d_2.\sigma_2$ with σ_1 and σ_2 in $(Act_M(C) \cup M_I)^*$, d_1 and d_2 in M_I, we have $\sigma.d_1 + d_2.\sigma_2$ in $ttraces(fp)$,*
- *for any timed trace in $ttraces(fp)$ of the form $\sigma.d.\sigma_2$ with σ_1 and σ_2 in $(Act_M(C) \cup M_I)^*$, d in M_I, for any d_1 and d_2 in M_I such that $d = d_1 + d_2$, we have $\sigma.d_1.d_2.\sigma_2$ in $ttraces(fp)$.*

The set of timed traces of $\mathbb{A}$, denoted $TTraces(\mathbb{A})$ is defined as follows:

- *for any fp in $FP(\mathbb{A})$ and for any σ in $ttraces(fp)$, we have σ in $TTraces(\mathbb{A})$,*
- *for any σ in $TTraces(\mathbb{A})$ of the form $\sigma'.d$ with σ' in $(Act_M(C) \cup M_I)^*$ and d in M_I, we have σ' in $TTraces(\mathbb{A})$.*

Example 12 *This an example of a timed trace of the TIOLTS $\mathbb{A}$ defined in Figure 3.2 and corresponding to the trace depicted in Example 11 :*

$$\overbrace{(0.2).(0.3)}^{\substack{decomposition\\0.5}}.env?"start".(0.01).cmd!"acquire".(0.004).cmd?"ack".\ \underbrace{(0.006)}_{\substack{0.004+\\0.002\\additivity}}.data?200.(0.02).(0.07).env?"start"$$

By applying last item of Definition 21, we have that the following trace is also a timed trace of $\mathbb{A}$ being a prefix of the previously given trace:

$(0.2).(0.3).env?"start".(0.01).cmd!"acquire".(0.004).cmd?"ack".0.006)$

This trace does not correspond to any a timed trace of any path in $\mathbb{A}$ simply because the duration 0.006 at the end of the trace does not appear explicitly in $\mathbb{A}$ and was obtained by adding durations as illustrated previously. However it is a possible trace of $\mathbb{A}$ since it corresponds to an acceptable waiting time in the execution of $\mathbb{A}$. Stating that the timed traces of $\mathbb{A}$ correspond to the timed traces of all paths of $\mathbb{A}$ is hence not sufficient. Indeed the last item of Definition 21 ensures that such a trace is taken into consideration in the set of timed traces of $\mathbb{A}$.

3.3.2 TIOLTS composition

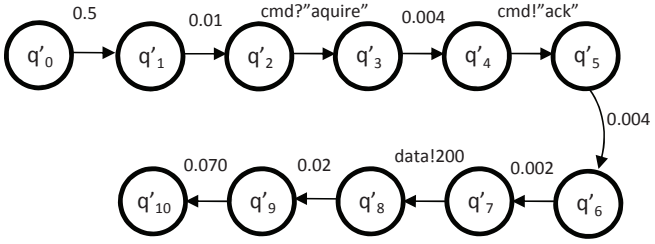
We define in the following how to compose two TIOLTS. In fact, two TIOLTS synchronize on actions performed on shared channels and time delays. Other actions are executed asynchronously.

Definition 22 (TIOLTS composition) *Let $\mathbb{A}_1 = (Q_1, q_0^1, T_1)$ and $\mathbb{A}_2 = (Q_2, q_0^2, T_2)$ be two TIOLTS respectively over C_1 and C_2. The composition of $\mathbb{A}_1$ and $\mathbb{A}_2$, denoted $\mathbb{A}_1 \parallel \mathbb{A}_2$, is an TIOLTS (Q, q_0, T) over $C_1 \cup C_2$ such that:*

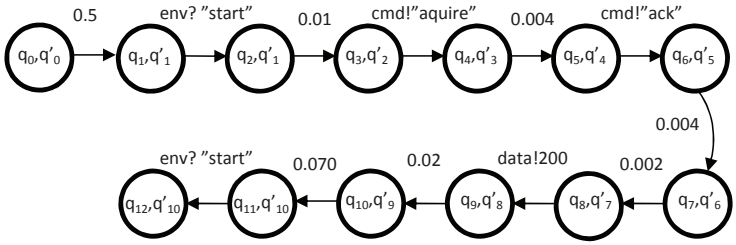
- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$
- *T is defined as follows:*
  - **synchronous execution***: if $(q_1, c!v, q_1') \in T_1$ and $(q_2, c?v, q_2') \in T_2$, such that $c \in C_1 \cap C_2$ then $((q_1, q_2), c!v, (q_1', q_2')) \in T$,*
  - **synchronous time passing***: if $(q_1, d, q_1') \in T_1$ and $(q_2, d, q_2') \in T_2$, such that $d \in M_I$ then $((q_1, q_2), d, (q_1', q_2)) \in T$;*
  - **asynchronous execution***: and for any $(q_1, a, q_1') \in T_1$ where a is of the form τ or $c?v$ or $c!v$ with $c \notin C_1 \cap C_2$, then for any $q_2 \in Q_2$, $((q_1, q_2), a, (q_1', q_2)) \in T$.*

  The role of $\mathbb{A}_1$ and $\mathbb{A}_2$ can be inverted.

Example 13 *Consider the TIOLTS in Figure 3.3b. It is the result of the composition of the TIOLTS $\mathbb{A}$, $\mathbb{A}'$ respectively in Figures 3.2,3.3a. To see this, first note that this TIOLTS is defined over the set of channels $C^{sys} = C^{ctrl} \cup C^{sens}$ that is $\{env, cmd, data\}$.*



(a) TIOLTS $\mathbb{A}'$



(b) TIOLTS $\mathbb{A}||\mathbb{A}'$

Figure 3.3: TIOLTS composition

For example, the transition $(q_5, q_4') \xrightarrow{cmd!"ack"} (q_6, q_5')$ corresponds to the synchronous execution of the transition $q_5 \xrightarrow{cmd?"ack"} q_6$ and the transition $q_4' \xrightarrow{cmd!"ack"} q_5'$: $\mathbb{A}'$ notifies $\mathbb{A}$ of the reception of its request for a new data by an acknowledgment message "ack".

The transition $(q_4, q_3') \xrightarrow{0.004} (q_5, q_4')$ corresponds to the synchronous execution of the transition $q_4 \xrightarrow{0.004} q_5$ and the transition $q_3' \xrightarrow{0.004} q_4'$: Time elapses of 0.004 delay since the acknowledgment message "ack" transited in the system.

Chapter 4

Formalizing UML MARTE sequence diagram

Contents

The UML syntax is precise thanks to meta-modeling, its semantics is informally defined. By means of translation of a subset of the UML metamodel, we carry out operational semantics for timed sequence diagrams: the target formalism is TIOSTS which are symbolic automata with time. In order to prepare this translation phase, we introduce a an intermediate representation as a textual syntax of such timed sequence diagrams. The textual representation is given as input to translation rules which generates a set of TIOSTS automata denoting the semantics of the sequence diagram. This intermediate representation has the advantage of describing concisely and formally the subset of UML that we use.

We begin by exemplifying our use of timed sequence diagrams on the component based system of a *Rain-sensing wiper control*. This example will be used through the whole chapter. We present the textual representation of timed sequence diagrams which is a simple way to delimit the subset of UML sequence diagram we consider. Besides, the formulation of the textual representation prepares the translation phase (see chapter 6).

4.1 Example: Rain-sensing wiper control system

Figure 4.1 specifies a Rain-sensor Wiper Controller in a car (RWC). This device automatically adjusts the frequency of the wipers according to the measured rain intensity. Every $0.5s$, the controller sends the rain intensity received from the environment (sensor) to the system calculator. The calculator computes the speed of the wiper. If the calculated speed changes, the system sends the new value to the wipers' engine.

The sequence diagram *sd RainSensingWiperControl* defines a cyclic behavior with the *loop* operator covering all the lifelines of the system ports. Upon execution, the lifeline associated with the port *speed* of the controller component *ctrl*, starts with an assignment action $ctrl.prevSpeed = 0$. In fact, a component may own *computation variables*. In the example, the component controller *ctrl* owns a computation variable *prevSpeed* which stores the speed that was last applied to the engine *eng*. This information serves later in the sequence diagram, to characterize interactions in which the wipers engine receives a new speed different from the previous applied one.

The system contains a controller *ctrl* which receives rain intensity values on its port *intensity*

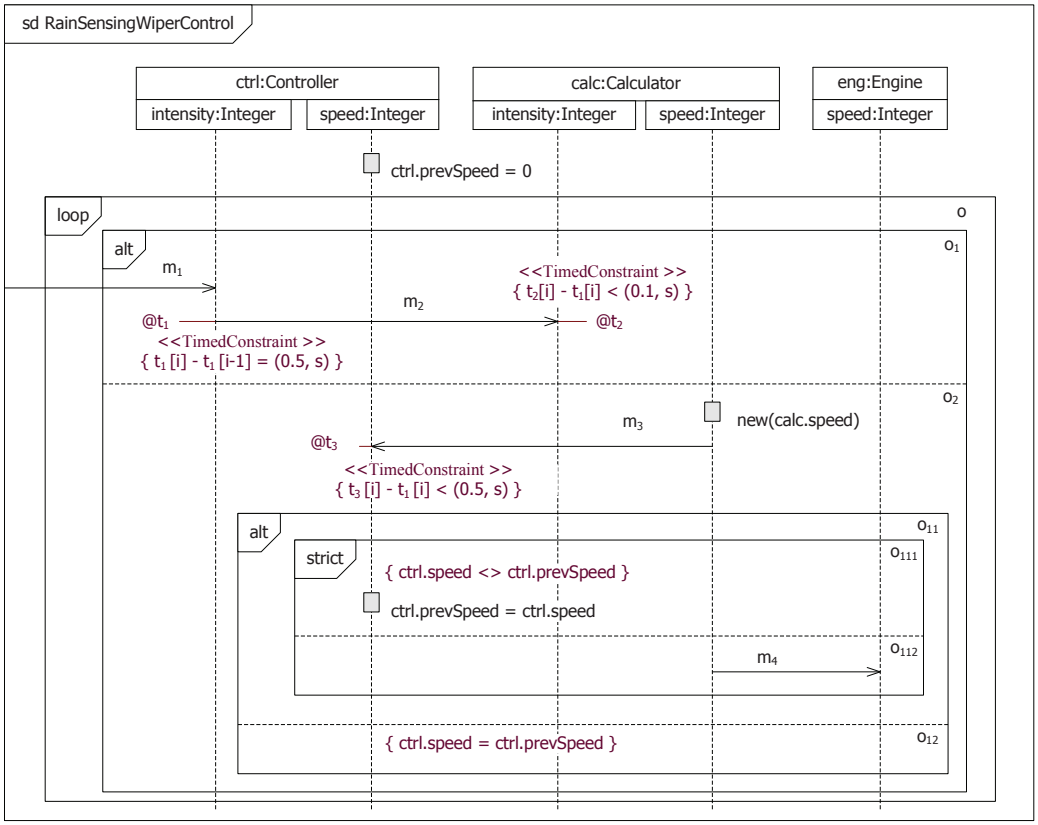


Figure 4.1: Rain-sensor Wiper Controller system (RWC) as a UML MARTE sequence diagram

from a sensor not depicted in the diagram: these values are conveyed by message m_1 whose source, supposed to be the sensor, is the environment of the sequence diagram. Every 0.5 seconds the received value is forwarded via m_2 to a calculator component *calc*, whose main purpose is to compute an appropriate speed for the wiper depending on the rain intensity. The frequency is identified by means of the constraint $t_1[i] - t_1[i-1] = (0.5, s)$. The message transmission is expected to take at most $0.1s$ (see the constraint $t_2[i] - t_1[i] < (0.1, s)$).

All this sequence occurs alternatively with another behavior (specified by the most external *alt* operator of the diagram): *ctrl* receives a new speed value computed by *calc* conveyed by message m_3. Note the formula at the target of m_3 which specifies that the reception occurs in between two occurrences of m_2. Then two alternative behaviors may occur depending on the new speed value (differentiated thanks to the most internal *alt* operator). If the new speed value is different from the previously computed speed ($ctrl.speed <> ctrl.prevspeed$) the $ctrl.prevspeed$ is updated ($ctrl.prevspeed = ctrl.speed$) and the new value is sent to the engine (message m_4). The updating and forwarding are sequentialized thanks to the *strict* operator. If $ctrl.speed = ctrl.prevspeed$ nothing happens.

4.2 Sequence diagram data signature

In order to represent types and operations of data in sequence diagrams, we consider a data signature $\Omega = (S, Op)$ and a set of typed variables $V = \cup_{s \in S} V_s$ as defined in Section 3.1, is given. S includes basic types such as *Integer* and *Boolean* and Op contains all operations names which relate to them. These types are used to define variables used in computations. Besides computation variables, sequence diagrams use special variables to handle instants in time hence the need to introduce more advanced types for that purpose.

Time modeling Time in the MARTE standard may be both of discrete or dense nature. In our setting, we introduce the type I (already discussed in Section 3.3 of Chapter 3) in order to type *time instants* whose associated model M_I may be either (a) the set of integers $\mathbb{Z}$ and so discrete isomorphic to positive natural numbers or (b) may be $\mathbb{R}$ the set of real numbers. Besides we introduce the type I^* for variables capturing time instants. Recall that we call them time variables as discussed in Section 2.1.5 of Chapter 2. In Figure 4.1, the time variable t_1 captures for example consecutive instants of the message m_2 emission, being in cyclic behavior. Typically this variable is considered of type I^*. The type I^* can be understood as a type of array of instants. Indeed a variable like t_1 as discussed before is used to capture successively instants (at each iteration of the loop going in region o_1).

In the sequel, we consider that Op contains the following operations:

- Op contains:
  $- : I \times I \to I$, (subtraction)
  $<: I \times I \to Boolean$ and (inequality less than)
  M_I associates respectively the function $-_{M_I} : M_I \times M_I \to M_I$ to $- : I \times I \to I$ and $<_{M_I}: M_I \times M_I \to M_I$ with $<: I \times I \to I$. These functions are respectively the subtraction and the inequality less than as they are classically defined on $M_I = \mathbb{Z}$ or $M_I = \mathbb{R}$.

- Op contains all the constant symbols associated with the type I. If I denotes discrete instants then it contains $0 :\to I$, $1 :\to I$, $2 :\to I$, etc. If I denotes real numbers then it contains all constants names of real numbers (e.g. $0.5 :\to I$).

- Op contains:
  $emptyArr :\to I^*$ (empty array)
  $[] : I^*.Integer \to I$ (access instant in array by location)
  $pushArr : I^*.I \to I^*$ (push instant onto array, add it to end of array)
  $len : I^* \to Integer$ (array length)

  These operations are interpreted in M_I as follows:

  - $emptyArr$ is associated with $emptyArr_{M_I}$ which is ϵ denoting the empty word;
  - $[] : I^*.Integer \to I$ is associated with a function $[]_{M_I} : M_I^*.M_{Integer} \to M_I$ such that $[]_{M_I}(d, a)$ is the value at location a in d whenever a denotes such an index. More precisely, if d is of the form $d_0 \dots d_n$ then we have $[]_{M_I}(d_0.d_1 \dots d_n, a)$ is d_a if $0 \leq a \leq n$ and has any value otherwise (whenever a falls beyond the length of d or is a negative integer);
  - $pushArr$ is associated with $pushArr_{M_I} : M_I^*.M_I \to M_I^*$ such that:
    $pushArr_{M_I}(d_0.d_1 \dots d_n, d_{n+1})$ returns $d_0.d_1 \dots d_n.d_{n+1}$ (adds an instant d_{n+1} to the end of d);
  - len is associated with the function $len_{M_I} : M_I^* \to M_{Integer_{\geq 0}}$ such that $len(d_0.d_1 \dots d_n)$ returns $n + 1$ the length of $d_0.d_1 \dots d_n$ and $len(\epsilon) = 0$.

Example 14 *In this example, we illustrate how some terms which relate to time, occurring in the sequence diagram of Figure 4.1 are built in our formal framework. Using the signature* $\Omega = (S, Op)$ *and over the sets of the typed variable names* $V_{Integer} = \{i\}$ *and* $V_{I^*} = \{t_1, t_2, t_3\}$*, one may built the following* Ω*-terms with variables in* $V = V_{Integer} \cup V_{I^*}$ *(*$T_\Omega(V)$*):*

0.1, 0.5, i, $-(i, 1)$, $[](t_1, i)$, $[](t_1, -(i, 1))$, $[](t_2, i)$, $[](t_3, i)$, $-([](t_1, i), []_I(t_1, i - 1)) = 0.5$, $<(-([](t_2, i), [](t_1, i)), 0.1)$ *and* $< (-([](t_3, i), []_I(t_1, i)), 0.5)$.

Let us discuss some of them. Regarding the the controller, the constant 0.5 *denotes the frequency of the transmissions of the rain intensity to the calculator and* 0.1 *denotes the maximum allowed propagation delay of the rain intensity to the calculator.* i *and* $-(i,1)$ *are integer terms denoting locations in time variables. The terms* $[](t_1,i)$ *and* $[](t_1,-(i,1))$ *denote the instants located respectively at* i *and* $-(i,1)$ *of the time variable* t_1 *related to the emissions of the rain intensity.*

Notation 6 *In the sequel for the sake of simplicity, we simply note* $t[x]$ *instead of* $[](t,x)$*. We often adopt as well an infix notation instead of the prefix one. For instance, we note* $t[x]-t[x-1]$ *rather than* $-([](t,x),[](t,x-1))$*.*

We introduce now a subset of formulas useful to the definition of timing constraints.

Definition 23 (Time formula) *The set of* time formula *denoted* $\mathscr{T}_\Omega(V \setminus V_I)$ *is inductively defined as follows:*

- *For any term* t_1 *and* t_2 *in* $\mathscr{T}_\Omega(V)$*, we have*
  - $t_1 = t_2$ *is in* $\mathscr{T}_\Omega(V)$*,*
  - $< (t_1,t_2) = True$ *is in* $\mathscr{T}_\Omega(V)$*,*
  - $< (t_1,t_2) = False$ *is in* $\mathscr{T}_\Omega(V)$*.*
- *For any* φ_1 *and* φ_2 *in* $\mathscr{T}_\Omega(V)$*, we have* $\varphi_1 \wedge \varphi_2$*,* $\varphi_1 \vee \varphi_2$ *and* $\neg\varphi_1$ *are in* $\mathscr{T}_\Omega(V)$*.*

Let us recall the meaning of time formula according to sequence diagrams. What is special about such formula, is that they may contain terms of the form $t[-1]$, that is for negative integer indexes where values are not significant. In which case, the formula is ignored (as if we have a *true* guard). Typically in a sequence diagram in Figure 4.1, the timing constraints $t_1[i]-t_1[i-1]=0.5$ states that between two successive emissions of m_2 there is a delay of 0.5, the first emission (where i equals to 0) occurs anyway: the evaluation of $t_1[0]-t_1[-1]=0.5$ is irrelevant and hence ignored in this case.

This validation of formulas as defined in Definition 7 does not reflect this semantical interpretation. However we can define from any time formula, a weaker formula that will be satisfiable exactly according to the semantic interpretation discussed above. In order to prepare this definition, consider the following definition which allows one to identify syntactically terms of the form $t[i]$ occurring in a time formula expression.

Definition 24 (Instants of a time formula) *Let* $\varphi \in \mathscr{T}_\Omega(V)$ *be a time formula. The set of* time instants associated with φ*, denoted* $InstantTerms(\varphi)$ *is defined as:*

- *if* φ *is of the form* $t[x]=d$ *where* $t \in V_{I^*}$*,* $x \in T_\Omega(V)_{Integer}$ *and* $d \in V_I$ *then* $InstantTerms(\varphi) = \{t[x]\}$
- *if* φ *is of the form* $t[x]=t'[y]$ *where* $t,t' \in V_{I^*}$ *and* $x,y \in T_\Omega(V)_{Integer}$ *then* $InstantTerms(\varphi) = \{t[x],t'[y]\}$
- *if* φ *is of the form* $\varphi_1 \wedge \varphi_2$ *and* $\varphi_1 \vee \varphi_2$ *then* $InstantTerms(\varphi) = InstantTerms(\varphi_1) \cup InstantTerms(\varphi_2)$*.*

Example 15 *Consider the time formula* $\varphi = t_1[i]-t_1[i-1]=0.5$*. The set of time instants associated with* φ *is* $\{t_1[i],t_1[i-1]\}$*.*

Now, we show how to define the formula discussed above (we call it "weak form" of the original formula). Firstly, we build a formula characterizing situations where the index occurring in a time instant term is out of bound of the corresponding time variable.

Definition 25 (Time index out of bound formula) *Let $t[x]$ be an instant term where $t \in V_{I^*}$ and $x \in T_\Omega(V)_{Integer}$. We note $IOB(t[x])$ (for Index Out of Bound) the formula $x < 0 \vee x > len(t)$.*

The weak form of a time formula is a formula which is true when φ is true or when some index of some time instant term is out of bound.

Definition 26 (Weak form of a time formula) *Let $\varphi \in \mathscr{T}_\Omega(V)$ be a time formula. The* weak form of φ, *denoted $WF(\varphi)$ is the formula $\varphi \vee \vee_{t \in InstantTerms(\varphi)} IOB(t)$.*

Example 16 *The weak form of $\varphi = t_1[i] - t_1[i-1] = 0.5$ is $\varphi = t_1[i] - t_1[i-1] = 0.5 \vee (i < 0 \vee i > len(t_1)) \vee (i - 1 < 0 \vee i - 1 > len(t_1))$.*

Specifically, $WF(\varphi)$ is evaluated to $True$ when $i = 0$. This means that when the first period of $0.5s$ starts (at $t_1[0]$) the formula is satisfied (regardless the value at $t_1[-1]$). Then, the only sub formula of $WF(\varphi)$ that matters when the second period starts (at $t_1[1]$, $i = 1$), is φ: it is obvious that the remaining sub formulas are not satisfied.

4.3 Sequence diagram syntax

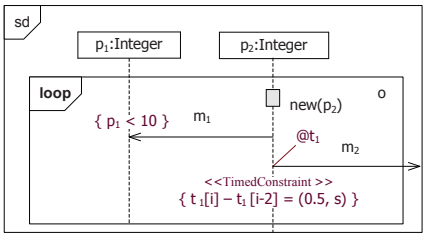
For any sequence diagram, we define the *sequence diagram signature* which structures all symbols introduced by the specifier that may occur in a sequence diagram.

Definition 27 (Sequence diagram signature) *A* Sequence diagram signature *(*signature *for short) is a 5-tuple $\Sigma = (P \cup \{e\}, Var \cup \{i\}, Msg, Obs, Reg)$ where*

- *P is a typed set of ports of the form $P = \cup_{s \in S} P_s$, e is distinct element representing the environment satisfying $e \notin P$,*
- *Var is a set of typed variables* called computation variables *$Var = \cup_{s \in S} Var_s$ such that $s \notin \{I, I^*\}$. We assume that Var can be partitioned as $Var = \cup_{p \in P} Var_p$ such that $p \neq p' \Rightarrow \emptyset$. Moreover i is a distinct variable of type $Integer$ satisfying $i \notin Var$.*
- *Msg is a set of typed message labels of the form $Msg = \cup_{(u,v) \in (P \cup \{e\})^2} Msg_{(u,v)}$, where:*
  - *for any $u, v \in P$ of different types, we have $Msg_{(u,v)} = \emptyset$*
  - *for any $u \in P \cup \{e\}$, we have $Msg_{(u,u)} = \emptyset$*
- *Obs is a set of time variables of type I^*,*
- *Reg is a set of regions names.*

In the sequel we note V the set of variables $P \cup Var \cup Obs \cup \{i\}$.

Example 17 *Consider as an example the signature Σ of the sequence diagram depicted in the upper part of Figure 4.2.*



$\Sigma = (P \cup \{e\}, Var \cup \{i\}, Msg, Obs, Reg)$ where:
$P = P_{Integer} = \{p_1, p_2\}$; $Var = \emptyset$;
$Msg = Msg_{(p_2,p_1)} \cup Msg_{(p_2,e)}$, where $Msg_{(p_2,p_1)} = \{m_1\}$ and $Msg_{(p_2,e)} = \{m_2\}$,
m_1 (respectively m_2) starts at port p_2 and ends at p_1 (respectively environment e);
$Obs = \{t_1\}$, t_1 is the unique time variable defined in the diagram capturing emission instants of m_2; and $Reg = \{o\}$ where o is the region of the loop operator.

Figure 4.2: Sequence diagram signature

Example 18 *The signature Σ_{RWC} of the sequence diagram of the Rain-sensing Wiper Control system (RWC) in Figure 4.1 defines these sets:*

$P = P_{Integer} = \{ctrl.intensity, ctrl.speed, calc.intensity, calc.speed, eng.speed\}$;
$Var = Var_{Integer} = Var_{ctrl.speed} = \{ctrl.prevSpeed\}$;
$Msg = Msg_{(e,ctrl.intensity)} \cup Msg_{(ctrl.intensity,calc.intensity)} \cup Msg_{(calc.speed,ctrl.speed)}$
$\cup Msg_{(calc.speed,eng.speed)}$,
where $Msg_{(e,ctrl.intensity)} = \{m_1\}$, $Msg_{(ctrl.intensity,calc.intensity)} = \{m_2\}$,
$Msg_{(calc.speed,ctrl.speed)} = \{m_3\}$, *and* $Msg_{(calc.speed,eng.speed)} = \{m_4\}$; $Obs = \{t_1, t_2, t_3\}$;
and $Reg = \{o, o_1, o_2, o_{11}, o_{12}, o_{111}, o_{112}\}$.

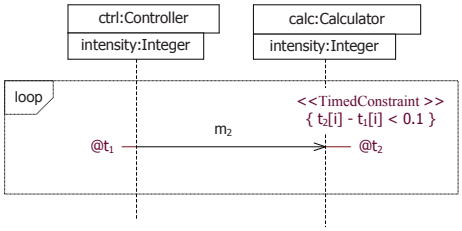
4.3.1 Messages

Recall that a message introduces a temporal order between two instants belonging respectively to the sending and receiving lifelines (which are by default incomparable) and hence allows one to constrain the transmission delay of the piece of data conveyed by the message. We encapsulate the message in a structure containing besides the message label, additional timing features. The *message* expression may contain two time variables which relate to the connection points of the message and also contains a timing constraint. Based on the sequence diagram signature definition Σ, the set of messages may be reformulated as follows:

Definition 28 (Messages) *The set of* messages *over Σ is the set $Msg(\Sigma)$ of all quadruples of the form: (t, ϕ_t, m, t') and $(\_, true, m, \_)$, where $t, t' \in Obs$ are time variables, $m \in Msg$, $\phi_t \in \mathscr{T}_\Omega(V)$ (the symbol $\_$ denotes the absence of a time variable).*

Note that a messages cannot be of form $(t, \phi_t, m, \_)$ or $(\_, \phi_t, m, t')$ because the intended use if to constrain the transmission delay and hence relate t and t' in the timing constraint expression. However constraining t or t' alone is possible at the lifeline level. In the following, we call a messages of the form (t, ϕ_t, m, t') (respectively $(\_, true, m, \_)$) a *a timed message* (respectively *simple message*).

Example 19 *Figure 4.3 depicts the expression of the message m_2 of the RWC system as specified in Figure 4.1. As discussed in the section, m_2 coveys cyclically (being in a loop operator region) the rain intensity from the controller to the calculator within at most* 0.1*s (stated by the timing constraint $t_2[i] - t_1[i] < 0.1$). The message $(t_1, t_2[i] - t_1[i] < 0.1, m_2, t_2)$ is an element of the set $Msg(\Sigma_{RWC})$ over the sequence diagram signature Σ_{RWC} defined in Example 18.*



$(t_1, t_2[i] - t_1[i] < 0.1, m_2, t_2)$

Figure 4.3: Syntax of a message of Rain-sensing Wiper Control system (RWC)

4.3.2 Lifelines

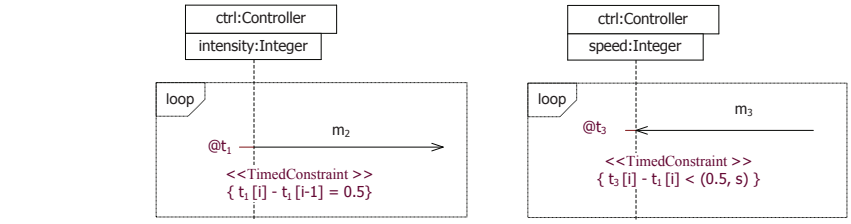
As we have seen in Chapter 2, there are three kinds of actions that may occur on a lifeline representing a given port, namely: an emission of a message, a reception of a message and a local action. These actions are performed atomically. They are the basic units of concurrency. These atomic actions occur at the port level, hence we call them *atoms of a port*. The instant at which the atom occurs may be constrained. Therefore, the expression of an atom may contain a time variable and a timing constraint. Besides, a data constraint is introduced to guard the action execution. This results in the encapsulation of the action in a richer structure, i.e. atom, such that its expression may contain data and timing features. We define the set of atoms of a given port as follows:

Definition 29 (Atoms of a port) *Let $p \in P$, the set of* atoms of p *over Σ is the set $Atom(p, \Sigma)$ of all quadruples of the form: (t, ϕ_t, ϕ_d, m), $(\_, true, \phi_d, m)$, $(t, \phi_t, \phi_d, new(x))$, $(\_, true, \phi_d, new(x))$, $(t, \phi_t, \phi_d, x = \varrho)$ and $(\_, true, \phi_d, x = \varrho)$, where $t, t' \in Obs$ are time variables, $\phi_t \in \mathscr{T}_\Omega(V)$, $\phi_d \in Sen_\Omega(Var_p \cup \{p\})$, $m \in \cup_{(u,v)\in(\{p\}\cup\{e\})^2\setminus\{(e,e)\}} Msg_{(u,v)}$, $x \in Var_p \cup \{p\}$, ϱ is term of $T_\Omega(Var_p \cup \{p\})$ (= is the assignment operation), the instructions of the form $new(x)$ randomly associates a new value with x.*

Similarly to messages, we distinguish two kinds of atoms, based on the presence or not of timing features, respectively called *timed atoms* and *simple atoms*.

Example 20 *Consider again the sequence diagram of RWC system. The Figure 4.4 illustrates expressions of lifeline atoms built over Σ_{RWC}: Atom in Figure 4.4a is in $Atom(ctrl.intensity, \Sigma_{RWC})$; Those in Figures 4.4b and 4.4d are elements of $Atom(ctrl.speed, \Sigma_{RWC})$; and the one in Figure 4.4c belongs to $Atom(calc.speed, \Sigma_{RWC})$. Note that the atom $(t_1, t_1[i]-t_1[i-1] = 0.5, true, m_2)$ in Figure 4.4a denotes a sending of a message since $m_2 \in Msg_{(ctrl.intensity, calc.intensity)}$ that is the message originates from the port ctrl.intensity and the atom is of the port ctrl.intensity. Similarly, $(t_3, t_3[i]-t_1[i] < 0.5, true, m_3)$ inFigure 4.4b denotes a reception of a message ($m_3 \in Msg_{(calc.speed, ctrl.speed)}$, the port ctrl.speed is the target of the message).*

Besides local actions, operators may be depicted on lifelines. Unlike actions which concern individual lifelines, an operator may cover some/all lifelines of the sequence diagram and so the behavior contained in its region concern all of them. However we translate each lifeline into a TIOSTS automaton. This allows us to characterize the sequence diagram as a set of communicating automata and classically obtain traces by synchronizing actions on values exchanges and interleaving the others. The translation requires to capture operators locally in lifeline expressions. From the point of view of the lifeline, it is sufficient to know the kind of the behavior (iterative with loop, choice with alt, etc.) and the portion of the behavior contained in the operator region which concerns that lifeline. Regions names are also kept in the lifeline



$(t_1, t_1[i] - t_1[i-1] = 0.5, true, m_2)$

(a) Sending atom

$(t_3, t_3[i] - t_1[i] < 0.5, true, m_3)$

(b) Reception atom

$(\_, true, true, new(calc.speed))$

(c) Underspecification atom

$(\_, true, ctrl.speed <> ctrl.prevSpeed, ctrl.prevSpeed = ctrl.speed)$

(d) Assignment atom

Figure 4.4: Syntax of lifeline atoms of the RWC system

expression because we make use of them in the translation mechanism as content of artifact messages for the lifelines to notify each others of some choice of behavior (e.g. when a lifeline chooses (non deterministically) to go in a given region of an alt operator, it notifies the other lifelines of the region choice, refer to the translation Chapter 6). This definition formulates the expression of a lifeline in an inductive manner:

Definition 30 (Lifeline of a port) *Let $p \in P$, the set $Lf(p, \Sigma)$ of* lifelines of p *is inductively defined as follows:*

- $\epsilon \in Lf(p, \Sigma)$
- *if* $lf \in Lf(p, \Sigma)$ *then* $(\textbf{\textit{seq}}, atom, lf) \in Lf(p, \Sigma)$ *where* $atom \in Atom(p, \Sigma)$,
- *if* $lf, lf' \in Lf(p, \Sigma)$ *then* $(\textbf{\textit{loop}}, o, lf, lf') \in Lf(p, \Sigma)$ *where* $o \in Reg$,
- *if* $lf, lf', lf'' \in Lf(p, \Sigma)$ *then* $(\textbf{\textit{alt}}|\textbf{\textit{strict}}, o, lf, o', lf', lf'') \in Lf(p, \Sigma)$ *where* $o, o' \in Reg$,

Example 21 *This is the expression of the lifeline associated with the port ctrl.intensity (see Figure 4.1) built over Σ_{RWC} (an element of $Lf(ctrl.intensity, \Sigma_{RWC})$):*
$lf_{calc.intensity} = (\textbf{\textit{loop}}, o, lf_0, lf_1)$, *where* $lf_1 = \epsilon$; $lf_0 = (\textbf{\textit{alt}}, o_1, lf_2, o_2, lf_3, lf_4)$; $lf_3 = \epsilon$; $lf_4 = \epsilon$; $lf_2 = (\textbf{\textit{seq}}, (\_, true, true, m_1), lf_5)$; $lf_5 = (\textbf{\textit{seq}}, (t_1, t_1[i] - t_1[i-1] = 0.5, true, m_2), lf_6)$; *and* $lf_6 = \epsilon$.

We define now sequence diagrams as couples of sets: the first set is the set of messages and the second one is the set of lifelines.

Definition 31 (Sequence diagram) *Let Σ be a sequence diagram signature. A sequence diagram sd over Σ is a couple (Msg, Lf), where Msg is a set of* messages *such that $Msg \subseteq Msg(\Sigma)$ and Lf is a set of lifelines of the form $\cup_{p \in P, lf \in Lf(p,\Sigma)}\{lf\}$.*

Example 22 *Consider again the sequence diagram in Figure 4.1 of the RWC system. Its textual expression is (Msg_{RWC}, Lf_{RWC}) where M_{RWC} is $\{m_1, m_2, m_3, m_4\}$ and Lf_{RWC} is $\{lf_{ctrl.intensity}, lf_{ctrl.speed}, lf_{calc.intensity}, lf_{calc.speed}, lf_{eng.speed}\}$.*

Chapter 5

Timed Input Output Symbolic Transition Systems (TIOSTS)

Contents

The goal of this chapter is to present Timed Input/Output Symbolic Transition Systems (TIOSTS) [12] that are later used to associate a formal counterpart with sequences diagrams with MARTE constraints. TIOSTS extend Input/Output Symbolic Transition Systems (IOSTS) (refer to Section 3.2). IOSTS are symbolic automata used to specify behaviors of reactive systems expressed as reactions by producing outputs to external stimuli. In few words, IOSTS are automata whose transitions are labeled by guards over variables called data constraints, by symbolic communication actions and by variable assignments which capture state evolutions. We define TIOSTS automata [12] which are similar to IOSTS except that they introduce timing constraints on transitions and particular variables to store instants as in sequence diagram semantics. Hence, TIOSTS formalism handles both data and time in a symbolic manner. This differentiates the TIOSTS from timed automata [3] (TA) used also in testing [29, 60, 54] where data is enumerated.

In this chapter, we present the syntax of TIOSTS automata and give their semantics. Then we discuss some closely related automata formalisms in the literature which treat time and data symbolically.

5.1 TIOSTS syntax

As glimpsed previously in the introduction, TIOSTS are automata in which one describes data manipulations in symbolic manner. In order to represent types and operations of these data, we use data signature and all along this chapter, we consider that a data signature $\Omega = (S, Op)$ as defined in section 3.1, is given. TIOSTS manipulate variables capturing instants as in sequence diagrams. For that purpose, we suppose that S contains the type I and I^* and Op contains all operation names introduced in Section 4.2.

In this section, we develop the syntax of TIOSTS and their composition which is used to represent communicating TIOSTS.

5.1.1 Basic definition

TIOSTS are defined upon *TIOSTS signatures* (or *signature* when the context is clear of confusion) which are used to define syntactical elements specific to a particular TIOSTS. A signature introduces two sets:

- a set of variables which are used either to abstract data handled in the TIOSTS or to stores instants and are of type I^*,
- a set of so-called *channels* which are used later to define communication actions.

A particularity of TIOSTS is that variables introduced in their signatures are partitioned into two sets. The first one is a set of so-called *read/write variables*, to which the TIOSTS may assign values or use associated values. The second one is a set of so-called *read only variables* whose values can be defined by the TIOSTS environment but not by the TIOSTS itself. In fact, what we call *environment* of a TIOSTS $\mathbb{G}$ corresponds to another TIOSTS communicating with $\mathbb{G}$. From the point of view of $\mathbb{G}$, the value of a read only variable of its signature may change without any control of $\mathbb{G}$. This will be discussed more precisely in Section 5.2 and the usefulness of such read only variables will made clear in Section 6.2.2.

Definition 32 (TIOSTS signature) *A* TIOSTS signature *(*signature *for short) is a couple* $\Sigma = (A, C)$ *where* A *is a set of variables typed in* S *and* C *is a set of* communication channels. *A is of the form* $A_{rw} \amalg A_r$ *where variables of* A_{rw} *are called* read/write variables *and variables of* A_r *are called* read only variables. *Moreover* A_{rw} *satisfies* $A_{rw} \cap A_{I^*} = A_{rw} \cap A_I = \emptyset$.

Notation 7 *We note* $Read(\Sigma)$ *the set* A_r *and* $Write(\Sigma)$ *the set* A_{rw}. *In the sequel,* A_{I^*} *is called the set of* time variables.

Example 23 *Let us define the signature* Σ^{ctrl} *as the couple* (A^{ctrl}, C^{ctrl}) *such that:*

- $A^{ctrl} = A^{ctrl}_{rw} \amalg A^{ctrl}_r$,
  where $A^{ctrl}_r = A^{ctrl}_{I^*} = \{t_1, t_2, t_3\}$ *(*$A^{ctrl}_I = \emptyset$*) and* $A^{ctrl}_{rw} = \{x_{env}, x_{cmd}, x_{data}, i_{t_1}, i_{t_3}\}$.
- $C^{ctrl} = \{env, cmd, data\}$

t_1, t_2, t_3 *are time variables and hence read only variables.* $x_{env}, x_{cmd}, x_{data}$ *are read/write variables assigned by data stored in communication or computation where* x_{env}, x_{cmd} *are of type* String *and* x_{data} *is of type* Integer. *Intuitively,* i_{t_1}, i_{t_3} *are read/write variables of type* Integer *which are used later to identify* places *where instants are stored respectively in* t_1, t_3. *Finally,* $env, cmd, data$ *are channels through which data inputs and outputs transit.*

TIOSTS are automata whose transition executions are associated with action occurrences. The set of actions that can be defined for a signature contains: input actions used to denote receptions from the TIOSTS environment; output actions which denote value emissions performed by the TIOSTS towards its environment; the action $new(x)$ which denotes an arbitrary updating of the data variable x and corresponds to a underspecification action introduced in sequence diagram (see section 2.1.2); and the invisible action τ which is classically used to denote the absence of an observable action during the execution of some transitions. Formally, the set of actions associated with a TIOSTS signature is defined as follows:

Definition 33 (TIOSTS actions) *Let* $\Sigma = (A, C)$ *be a TIOSTS signature.*
The set of TIOSTS actions *(*actions *for short) over* Σ *is defined as*

$$Act(\Sigma) ::= c?x | c!t | new(x) | \tau,$$

where $c \in C$, $x \in A_{rw}$, $t \in T_\Omega(A_{rw})$. $c?x$ denotes the reception of a value on channel c stored in x, $c!t$ denotes the emission of the value assigned to t on channel c.

Example 24 *Based on the signature Σ^{ctrl} of Example 23, we give some actions in $Act(\Sigma)$:*

- *The action $env?x_{env}$ represents an input received by the system on the channel env and stored in the variable x_{env}.*
- *The action $cmd!$"$acquire$" represents an output sent by the system on channel cmd as a string constant "$acquire$".*

As in the case of IOSTS, transitions of a TIOSTS introduce: a source state, a formula called a guard over data variables defining a constraint on variable interpretations for the transition firing, a communication action, an assignment of data variables to update their values and a target state. In addition to these features, transitions of a TIOSTS introduce two notions: first they declare a (possibly empty) set of time variables used to capture the current instant, second they introduce a guard over time. A guard over time is a time formula constraining the instant of the transition execution as defined in Definition 23.

Definition 34 (TIOSTS) *Let $\Sigma = (A, C)$ be a TIOSTS signature. A TIOSTS over Σ is a triple (Q, q_0, T), where Q is a set of* states, *$q_0 \in Q$ is the* initial state *and T is a set of* transitions *of the form $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ where $q, q' \in Q$, $\mathbb{T} \subseteq A_{I^*}$, $\phi_t \in \mathscr{T}_\Omega(A)$, $\phi_d \in Sen_\Omega^{qf}(A_{rw})$, $act \in Act(\Sigma)$ and ρ is a substitution of variables of A_{rw} in $T_\Omega(A_{rw})$.*

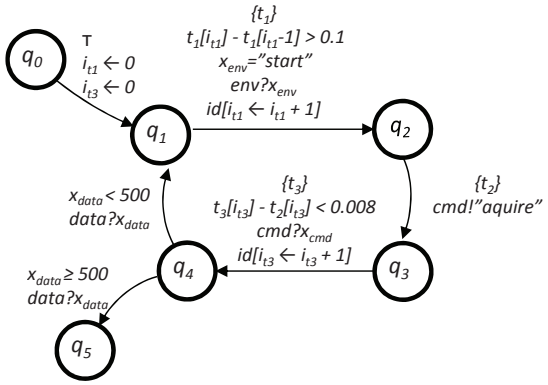
ϕ_t is a time formula constraining the instants at which the action *act* occurs. ϕ_d is a firing condition on data variables. ρ assigns new values to data variables when the transition is executed. Values assigned to variables occurring in $\mathbb{T}$ are updated implicitly by storing at the last defined index the instant of occurrence of *act* when the transition is executed. $\mathbb{T}$ is not restricted to a singleton because a TIOSTS may result from a composition of several TIOSTS, each of them defining instants with different time variables (See Definition 35).

Notation 8 *For any transition tr of the form $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ we use the notations $source(tr)$, $var_t(tr)$, $guard_t(tr)$, $guard_d(tr)$, $act(tr)$, $\rho(tr)$, and $target(tr)$ in order to refer, respectively, to q, $\mathbb{T}$, φ_t, ϕ_d, act, ρ, and q'. If $\rho(tr)$ does not affect any variable in A_{rw}, we note it id, which stands for the identity function over the set A_{rw}.*

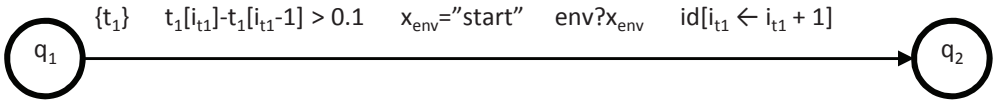
Example 25 *In the rest of this chapter, we use a toy example for illustration: the system consists of a controller and a sensor. The controller is repeatedly activated by a command "start", with a spaced time delay of at least* 0.1 *seconds. The controller sends an initiate command "acquire" to the sensor, and waits for the sensor to reply with an acknowledgment "ack". After the acknowledgment is received, the controller receives the measurement data from the sensor.*

Figure 5.1a illustrates the TIOSTS $\mathbb{G}^{ctrl}$ for the system controller over the signature Σ^{ctrl} that was defined in Example 23.

Let us focus on the transition from q_1 to q_2 (there is only one) of $\mathbb{G}$, Figure 5.1b zooms in on this transition. The transition reflects the system evolution from one source state (here state q_1) to another target state (state q_2) over time. The transition action ($env?x_{env}$) is fired. Note that the transition may be executed many times because of the iterative behavior of the system, and thus has many execution instants. We capture these instants in the time variable associated with the transition (t_1). Also these instants may be constrained by a time guard associated with the transition: in the example, the time guard is $t_1[i_{t_1}] - t_1[i_{t_1} - 1] > 0.1$. Also, a condition on



(a) $\mathbb{G}^{ctrl}$



(b) TIOSTS transition

Figure 5.1: Timed Input Output Transition Systems (TIOSTS)

data must be satisfied to cover the transition: the condition x_{env} = "start" must be true after the reception action env?x_{env} (of a value stored in x_{env}). The substitution associated with the transition is $id[i_{t_1} \leftarrow i_{t_1} + 1]$ or can be simply written $[i_{t_1} \leftarrow i_{t_1} + 1]$. It increments the time index i_{t_1} used to write the constraint on time as explained before, and leaves the other variables unchanged. As a whole, the transition denotes the activation of the system by the command "start" received from the environment on the channel env.

Consider the transition $q_2 \xrightarrow[cmd!"acquire"]{\{t_2\}} q_3$ of $\mathbb{G}^{ctrl}$. Here, the controller asks for a new data measure by sending the command "acquire" on the channel cmd.

Let us consider now the branching in the automaton. From state q_4, two behaviors are possible: either the system receives a piece of data which does not surpass the acceptable threshold (transition $q_4 \rightarrow q_1$, $x_{data} < 500$) and iterates with a new cycle; otherwise, the system exits the looping behavior (transition $q_4 \rightarrow q_5$, $x_{data} \geq 500$).

5.1.2 TIOSTS composition

Any TIOSTS $\mathbb{G}$ can be the result of a structuring of basic TIOSTS (not structured of composition) by means of a composition operator. That operator reflects communications between these basic TIOSTS. The composition makes both TIOSTS execute two communication actions performed on the same channel simultaneously. The synchronized execution consists in one TIOSTS outputting a value on a shared channel and the second TIOSTS inputting that value on the same channel. Any other action which does not have a match in that sense is evaluated asynchronously.

Definition 35 (TIOSTS composition) *Let $\Sigma_1 = (A^1, C^1)$ and $\Sigma_2 = (A^2, C^2)$ be two TIOSTS signatures such that $A^1_{rw} \cap A^2_{rw} = \emptyset$. Let us define $\Sigma_1 + \Sigma_2$ the signature (A, C) such that: $A_{rw} = A^1_{rw} \cup A^2_{rw}$, $A_r = (A^1_r \cup A^2_r) \setminus (A^1_{rw} \cup A^2_{rw})$ and $C = C^1 \cup C^2$.*

Let $\mathbb{G}_1 = (Q_1, q^1_0, T_1)$ and $\mathbb{G}_2 = (Q_2, q^2_0, T_2)$ be two TIOSTS respectively over Σ_1 and Σ_2. The composition of $\mathbb{G}_1$ and $\mathbb{G}_2$ *denoted $\mathbb{G}_1||\mathbb{G}_2$ is the TIOSTS (Q, q_0, T) over $\Sigma_1 + \Sigma_2$ where*

$Q = Q_1 \times Q_2$, $q_0 = (q_0^1, q_0^2)$ *and* T *is defined as follows:*

- **asynchronous execution** *for any* $(q_1, q_2) \in Q_1 \times Q_2$ *and* $tr_1 = (q_1, \mathbb{T}, \phi_t, \phi_d, act, \rho, q_1') \in T_1$ *where act is not of the form* $c?x$ *or* $c!t$ *with* $c \in C_1 \cap C_2$,
  we have $tr = ((q_1, q_2), \mathbb{T}, \phi_t, \phi_d, act, \rho, (q_1', q_2)) \in T$. *The role of* $\mathbb{G}_1$ *and* $\mathbb{G}_2$ *can be inverted.*
- **synchronous execution** *for any* $tr_1 = (q_1, \mathbb{T}_1, \phi_t^1, \phi_d^1, c?x, \rho_1, q_1') \in T_1$
  and $tr_2 = (q_2, \mathbb{T}_2, \phi_t^2, \phi_d^2, c!t, \rho_2, q_2') \in T_2$, *let us define* $\rho_1||\rho_2$ *the substitution from* $A_{rw}^1 \cup A_{rw}^2$ *to* $T_\Omega(A_{rw}^1 \cup A_{rw}^2)$ *as: for all*[1] $y \in A_{rw}^1 \cup A_{rw}^2$, $\rho_1||\rho_2(y) = [x \leftarrow t] \circ \rho_1(y)$ *if* $y \in A_{rw}^1$ *and* $\rho_1||\rho_2(y) = \rho_2(y)$ *if* $y \in A_{rw}$. *Then we have* $tr = ((q_1, q_2), \mathbb{T}_1 \cup \mathbb{T}_2, \phi_t^1 \wedge \phi_t^2, [x \leftarrow t](\phi_d^1) \wedge \phi_d^2, c!t, \rho_1||\rho_2, (q_1', q_2')) \in T$. *The role of* $\mathbb{G}_1$ *and* $\mathbb{G}_2$ *can be inverted.*

Example 26 *Recall that our system consists of a controller and a sensor. We built the TIOSTS corresponding to the whole system exactly from two TIOSTS: the one of the controller already defined in Figure 5.1a and the TIOSTS of the sensor given in Figure 5.2a.*

We consider also the signature $\Sigma^{sens} = (A^{sens}, C^{sens})$ *defined for the TIOSTS of the sensor as follows:*

- $A^{sens} = A_{rw}^{sens} \amalg A_r^{sens}$, *where* $A_{rw}^{sens} = \{x'_{cmd}, x'_{data}, i'_{t'_2}\}$ *and* $A_r^{sens} = \{t'_1, t'_2\}$
- $C^{sens} = \{cmd, data\}$.

We note $\mathbb{G}^{sys}$ *the TIOSTS of the system, depicted in Figure 5.2b. It is the result of the composition of* $\mathbb{G}^{ctrl}$ *and* $\mathbb{G}^{sens}$. *To see this, firstly note that* $\mathbb{G}^{sys}$ *is defined over the signature* $\Sigma^{sys} = (A^{sys}, C^{sys})$, *where* $A^{sys} = A^{ctrl} \cup A^{sens}$ *and* $C^{sys} = C^{ctrl} \cup C^{sens}$. *So, we have:*

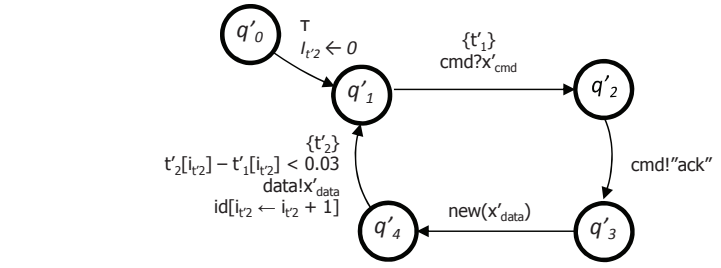
- $A^{sys} = A_{rw}^{sys} \amalg A_r^{sys}$ *such that* $A_{rw}^{sys} = \{x_{env}, x_{cmd}, x_{data}, i_{t_1}, i_{t_3}, x'_{cmd}, x'_{data}, i'_{t'_2}\}$ *and* $A_r^{sys} = \{t_1, t_2, t_3, t'_1, t'_2\}$,
- $C^{sys} = \{env, cmd, data\}$.

We next show examples of transitions in the product and how they were computed.

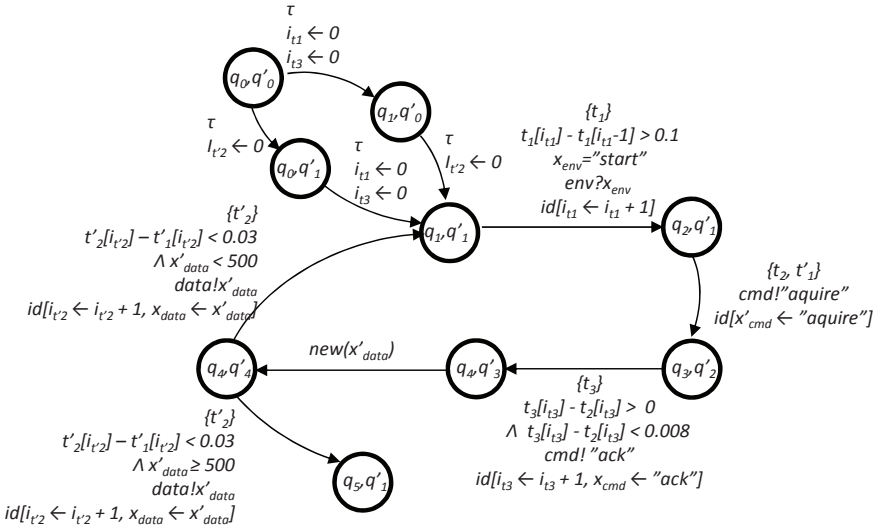
The transition $(q_1, q_1') \xrightarrow[\substack{env?x_{env} \\ i_{t_1} \leftarrow i_{t_1}+1}]{\{t_1\}\ 0.1<t_1[i_{t_1}]-t_1[i_{t_1}-1], x_{env}="start"} (q_2, q_1')$ *in Figure 5.2b was obtained from the asynchronous execution of the transition of* $\mathbb{G}^{ctrl}$, *illustrated in Figure 5.1b since the channel env is not shared between the two composed TIOSTS (see the first item of Definition 35). Thus the controller TIOSTS evolves to the state* q_2 *while the sensor TIOSTS stalls in state* q_1'.

Now consider the successor transition $(q_2, q_1') \xrightarrow[\substack{cmd!"acquire" \\ x'_{cmd} \leftarrow "acquire"}]{\{t_2, t'_1\}} (q_3, q_2')$. *It results from the synchronous execution of the transitions, which takes the controller from state* q_2 *to* q_3 *and the sensor from state* q_1' *to* q_2'. *They exchange the value "acquire" on the channel cmd (see the second item of Definition 35) which is stored in the variable* x'_{cmd} *(of the sensor): it is easy to see that* $[x'_{cmd} \leftarrow "acquire"] \circ id = [x'_{cmd} \leftarrow "acquire"]$. *Note that the set of time variables associated with the resulting transition contains two variables* $\{t_2, t'_1\}$, *each one comes from a different TIOSTS involved in the product.*

[1] $[x \leftarrow t]$ is the substitution associating t to x and leaving all other variables unchanged. We also note $[x \leftarrow t]$ its extension to formulae.



(a) $\mathbb{G}^{sens}$



(b) $\mathbb{G}^{sys} = \mathbb{G}^{ctrl}||\mathbb{G}^{sens}$

Figure 5.2: TIOSTS Composition

Finally note that all the τ transitions are executed asynchronously (see transitions between (q_0, q'_0) and (q_1, q'_1) corresponding the initialization of the time indexes).

In the sequel, we are interested in being able to identify transitions of $\mathbb{G}_1$ and $\mathbb{G}_2$ used to build a given transition in $\mathbb{G}_1||\mathbb{G}_2$. For example, the transition $((q_1, q_2), \mathbb{T}, \phi_t, \phi_d, act, \rho, (q'_1, q_2))$ of the item **asynchronous execution** in Definition 35 is build from the transition $(q_1, \mathbb{T}, \phi_t, \phi_d, act, \rho, q'_1)$ of $\mathbb{G}_1$. In the same way, the transition $((q_1, q_2), \mathbb{T}_1 \cup \mathbb{T}_2, \phi_t^1 \wedge \phi_t^2, [x \leftarrow t](\phi_d^1) \wedge \phi_d^2, c!t, \rho_1||\rho_2, (q'_1, q'_2))$ of item **synchronous execution** in Definition 35 is built over transition $(q_1, \mathbb{T}_1, \phi_t^1, \phi_d^1, c?x, \rho_1, q'_1)$ of $\mathbb{G}_1$ and the transition $(q_2, \mathbb{T}_2, \phi_t^2, \phi_d^2, c!t, \rho_2, q'_2)$ of $\mathbb{G}_2$.

In order to identify those transitions of basic TIOSTS used to build a transition of a composition, we use a syntactic naming mechanism. In the sequel, we suppose that all basic TIOSTS (i.e. not resulting of a composition) are associated with a naming function for transitions and we show how to build a name for any transition of any composition of basic TIOSTS. We begin by defining the set of TIOSTS that can be obtained by composing basic TIOSTS.

Definition 36 (Systems) *Let $\mathbb{C}ol$ be a set of TIOSTS. The set of $Sys(\mathbb{C}ol)$ of* systems *over $\mathbb{C}ol$ is defined inductively as follows:*

- *for any $\mathbb{G} \in \mathbb{C}ol$, $\mathbb{G} \in Sys(\mathbb{C}ol)$*
- *for any $\mathbb{G}_1$ and $\mathbb{G}_2 \in Sys(\mathbb{C}ol)$ such that $\mathbb{G}_1||\mathbb{G}_2$ is defined, we have $\mathbb{G}_1||\mathbb{G}_2 \in Sys(\mathbb{C}ol)$*

Now a set of TIOSTS being given (we call such a set *a collection*, that is why we note it $\mathbb{C}$ in Definition 36), we define the notion of *transition naming system* associated with it.

In the sequel, we consider that a set TN of transition names is given.

Definition 37 (Naming) *Let $\mathbb{C}ol$ be a set of TIOSTS. A transition naming system over $\mathbb{C}ol$ is a set $NS = \bigcup_{\mathbb{G}\in\mathbb{C}ol}\{(\mathbb{G}, n)\}$ such that for all $\mathbb{G} \in \mathbb{C}ol$ and for all $n, n' \in TN$, if $(\mathbb{G}, n) \in NS$ and $(\mathbb{G}, n') \in NS$ then $n = n'$. For any $(\mathbb{G}, n) \in NS$, $n : Trans(\mathbb{G}) \to 2^{TN}$ is injective such that for any $tr \in Trans(\mathbb{G})$, $n(tr)$ is a singleton. For any $(\mathbb{G}_1, n_1), (\mathbb{G}_2, n_2) \in NS$, for any $tr_1 \in Trans(\mathbb{G}_1)$ and any $tr_2 \in Trans(\mathbb{G}_2)$, we have if $\mathbb{G}_1 \neq \mathbb{G}_2$ then $n(tr_1) \neq n(tr_2)$.*

In the following, for any $(\mathbb{G}, n) \in NS$ we note $name_{\mathbb{G}}$ the function n. $name_{\mathbb{G}}$ returns a singleton. Definition 38 extends transition naming to systems by returning sets of basic transition names.

Definition 38 (Naming extension to systems) *Let NS be a transition naming system. The* extension of NS to $Sys(\mathbb{C}ol)$ *is the set $\breve{NS} = \bigcup_{S\in Sys(\mathbb{C}ol)}\{(S, n)\}$ such that $(S, n) \in \breve{NS}$ if and only if:*

- *if $S \in \mathbb{C}ol$ then $n = name_S$*
- *if S is of the form $S_1||S_2$ with $(S_1, n_1), (S_2, n_2) \in \breve{NS}$ then for any $tr \in Trans(S_1||S_2)$ we have:*
  - *if tr is obtained by applying the item* **asynchronous execution** *in Definition 35 with a transition $tr_1 \in Trans(S_1)$ (respectively $tr_2 \in Trans(S_2)$) then $n(tr) = n_1(tr_1)$ (respectively $n(tr) = n_2(tr_2)$),*
  - *if tr is obtained by applying the item* **synchronous execution** *in Definition 35 with a transition $tr_1 \in Trans(S_1)$ and a transition $tr_2 \in Trans(S_2)$ then $n(tr) = n_1(tr_1) \cup n_2(tr_2)$.*

We apply the convention that for any $(S, n) \in \breve{NS}$, we note $name_S$ for the function n as we do for basic TIOSTS. For any transition tr of a system S, $name_S(tr)$ returns names of basic TIOSTS transitions that have been used by applying successively items of Definition 35 to build tr.

Example 27 *Consider the transition $tr : (q_2, q'_1) \xrightarrow[\substack{cmd!"acquire"\\ x'_{cmd}\leftarrow"acquire"}]{\{t_2, t'_1\}} (q_3, q'_2)$ (see Figure 5.2b). tr was built by synchronizing the two transitions $tr_1 : q_2 \xrightarrow[cmd!"acquire"]{\{t_2\}} q_3$ and $tr_2 : q'_1 \xrightarrow[cmd?x'_{cmd}]{\{t'_1\}} q'_2$. Let $name_{\mathbb{G}^{ctrl}}(tr_1)$ and $name_{\mathbb{G}^{sens}}(tr_2)$ be respectively $\{n_1\}$ and $\{n_2\}$, we have $name_{\mathbb{G}^{ctrl}||\mathbb{G}^{sens}}(tr)$ is $\{n_1, n_2\}$.*

5.2 TIOSTS semantics

We propose to define TIOSTS semantics using TIOLTS (Timed Input Output Labeled Transition System).

TIOLTS associated with a TIOSTS

As glimpsed in the introduction of Section 3.3, semantics (i.e. behavior) of a TIOSTS is defined as a set of traces. In order to build this set of traces for a TIOSTS, we define a TIOLTS

associated with the TIOSTS and then define the semantics. To reach that goal, we first define the so-called *runs* of TIOSTS transitions. Intuitively, a run of a transition is simply the mathematical representation of a possible execution of that transition.

Definition 39 (Runs of Transitions) *Let $\mathbb{G} = (Q, q_0, T)$ be an TIOSTS over $\Sigma = (A, C)$. The set of snapshots of $\mathbb{G}$ is the set $Snp_M(\mathbb{G}) = Q \times M_I \times M^A$. For any $tr \in T$ of the form $(q, \mathbb{T}, \varphi_t, \varphi_d, act, \rho, q')$, the* set of runs of *$tr$ is the set $Run(tr) \subseteq Snp_M(\mathbb{G}) \times Act_M(C) \times Snp_M(\mathbb{G})$ such that:*

$((q, \mathcal{T}, \nu), act_M, (q', \mathcal{T}', \nu')) \in Run(tr)$ if and only if $\mathcal{T} \leq \mathcal{T}'$ and there exists $\nu^i : A \to M$ satisfying:

- *for all $x \in \mathbb{T}$, we have $\nu^i(x) = pushArr_{M_I}(\nu(x), \mathcal{T}')$,*
- *if act is of the form $c!t$ (respectively τ) then for all $x \in A \setminus \mathbb{T}$ we have $\nu^i(x) = \nu(x)$,*
- *if act is of the form $c?x$ (respectively $new(x)$) then for all $y \in A \setminus (\mathbb{T} \cup \{x\})$ we have $\nu^i(y) = \nu(y)$,*

such that $act_M = \nu^i(act)$, for all $x \in A_{rw}$ we have $\nu'(x) = \nu^i(\rho(x))$, for all $x \in A_{I^*}$ we have $\nu'(x) = \nu^i(x)$, $M \models_{\nu_i} WF(\varphi_t)$ and $M \models_{\nu_i} \varphi_d$.

Notation 9 *For any run r of the form $((source(tr), \mathcal{T}, \nu), act_M, (target(tr), \mathcal{T}', \nu')) \in Run(tr)$, the transition tr is called the* ground transition *of r and is denoted $g(r)$, the duration $\mathcal{T}' - \mathcal{T}$ is called* duration *of tr and is denoted $\delta(r)$. $source(r)$, $act(r)$ and $target(r)$ stand respectively for $(source(tr), \mathcal{T}, \nu)$, act_M and $(target(tr), \mathcal{T}', \nu')$. Finally for any snapshot $snp = (q, \mathcal{T}, \nu)$, $state(snp)$ stands for q, $\mathcal{T}(snp)$ stands for $\mathcal{T}$ and $\nu(snp)$ stands for ν.*

Let us comment Definition 39. $Snp_M(\mathbb{G})$ is the mathematical denotation of a current numeric state of the TIOSTS. Such numeric state, or snapshot, characterizes: a given state of $\mathbb{G}$, that is supposed to be reached after some executions; an instant of M_I which represent the instant at which the state is reached; and an interpretation of variables of A denoting current values of variables. A run of a transition is simply a triple that introduces : a snapshot denoting the numeric state before executing the transition; a numeric communication action associated with the transition execution; and finally a snapshot denoting the numeric state after the transition execution. Following notations of Definition 39, the instant $\mathcal{T}'$ after the execution of tr is supposed to be posterior to the instant $\mathcal{T}$ before the execution of tr ($\mathcal{T} \leq \mathcal{T}'$). It is also the instant at which act_M occurs. Now we have to take into account that tr can be executed if and only if both ϕ_t and ϕ_d are satisfied. Let us note two facts : first ϕ_t should be true at the instant $\mathcal{T}'$ when act_M occurs; second we suppose that ϕ_d is true after having taken into account the new value received whenever $act(tr)$ is a reception (or a new definition of variable in the case of an action of the form $new(x)$). Those two facts are not taken into account in ν. Therefore, we build an intermediate interpretation ν_i to take them into account. Let us comment in order the three items of Definition 39 :

- The variables of $\mathbb{T}$ are the time variables that are supposed to store the occurrence instant of act_M. Since they are all arrays, we use the function $pushArr_{M_I}$ to store them (denoted by $\nu^i(x) = pushArr_{M_I}(\nu(x), \mathcal{T}')$).
- The second item states that whenever tr does not introduce a reception (or a redefinition) then for all variables not in $\mathbb{T}$ we have ν_i is defined as ν.

- third item states that if tr introduces an input of the form $c?x$ (or a redefinition of x of the form $new(x)$) then for all variables not in $\mathbb{T}$ and different from x, we have ν_i is defined as ν. There is no constraints on $\nu_i(x)$ since it represents a value received which is not controlled by $\mathbb{G}$ (or a random redefinition of x).

Now we consider the satisfaction of ϕ_t. Recall as discussed in Section 4.2, that ϕ_t introduces terms of the form $t[i]$ referring to the value in the array t stored at place i. Moreover recall that the concrete value assigned to i may some times refer to an irrelevant place (typically -1 is an irrelevant place) and to be consistent with an accepting sequence diagram execution. We impose that transition can be fired if $\nu_i(i)$ represents such value (and of course if $M \models_{\nu_i} \phi_d$ too). To reach that goal in Definition 39, we do not impose that $M \models_{\nu_i} \phi_t$ which could be false although $\nu_i(i)$ refers to an irrelevant place. However, we impose $M \models_{\nu_i} WF(\phi_t)$ where $WF(\phi_t)$ is defined in Definition 25 of Section 4.2. Recall that $WF(\phi_t)$ is a formula which is true whenever either ϕ_t is true or same index associated to time variables refers to irrelevant places. Finally, ν' takes into account assignment ρ for all variables in A_{rw} ($\nu'(x) = \nu^i(\rho(x))$). ν' is ν_i for all variables in A_{I^*} and there are no constraints on ν' for variables in $A_r \setminus A_{I^*}$ because their associated value evaluation is not controlled by the TIOSTS $\mathbb{G}$.

Example 28 *Let us consider again the TIOSTS $\mathbb{G}^{ctrl}$ depicted in Figure 5.1a and defined over the signature $\Sigma^{ctrl} = (A^{ctrl}, C^{ctrl})$ in Example 23. Recall Σ^{ctrl} is defined as follows:*

- $A^{ctrl} = A^{ctrl}_{rw} \amalg A^{ctrl}_{r}$,
  where $A^{ctrl}_{r} = A^{ctrl}_{I^*} = \{t_1, t_2, t_3\}$ *($A^{ctrl}_{I} = \emptyset$) and* $A^{ctrl}_{rw} = \{x_{env}, x_{cmd}, x_{data}, i_{t_1}, i_{t_3}\}$.
- $C^{ctrl} = \{env, cmd, data\}$

Specifically consider the transition tr as an illustration of Definition 39:

$$q_1 \xrightarrow[\substack{env?x_{env} \\ i_{t_1} \leftarrow i_{t_1}+1}]{\{t_1\}\ 0.1<t_1[i_{t_1}]-t_1[i_{t_1}-1], x_{env}=\text{"}start\text{"}} q_2$$

We discuss a possible run r of the transition tr:

$$(q_1, 0, \nu_0) \xrightarrow{env?\text{"}start\text{"}} (q_2, 0.5, \nu_1),$$

where

interpretation of variables
$\nu_0(t_1) = \epsilon$, $\nu_0(t_2) = \epsilon$, $\nu_0(t_3) = \epsilon$ $\nu_0(i_{t_1}) = 0$, $\nu_0(i_{t_3}) = 0$ $\nu_0(x_{env}) =$ "*none*", $\nu_0(x_{cmd}) =$ "*none*", $\nu_0(x_{data}) = 999$
$\nu_1(t_1) = pushArr_{M_I}(\epsilon, 0.5) = 0.5$, $\nu_1(t_2) = \epsilon$, $\nu_1(t_3) = \epsilon$ $\nu_1(i_{t_1}) = 1$, $\nu_1(i_{t_3}) = 0$ $\nu_1(x_{env}) =$ "*start*", $\nu_1(x_{cmd}) =$ "*none*", $\nu_1(x_{data}) = 999$

Initially in the snapshot source(r) that is $(q_0, 0, \nu_0)$, the reached state of $\mathbb{G}^{ctrl}$ is q_0 and the time instant is 0. The time variable t_1 associated with tr is the empty array ($\nu_0(t_1) = \epsilon$ where the empty word ϵ is the semantic counterpart of emptyArr, refer to Section 4.2), hence we consider a run reflecting a first execution of tr.

In order for tr to be fired both guards $0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]$, $x_{env} =$ "start" must be satisfied. The latter is satisfied since after the run, according to the third item of Definition 39, x_{env} has an arbitrary value (being in a reception action $env?x_{env}$, and not redefined because the only redefinition here is $i_{t_1} \leftarrow i_{t_1} + 1$) however the data guard require it to have the value "start" ($x_{env} =$ "start") hence the only acceptable assignment is that value ($\nu_1(x_{env}) =$ "start").

At this level of the execution where $\nu_0(i_t) = 0$ *and* $\nu_0(t_1) = \epsilon$*, the timing guard to be satisfied is rather* $WF(0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1])$ *that is:*

$$\underbrace{0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]}_{0.1<\epsilon[0]-\epsilon[-1]} \vee \underbrace{i_{t_1} < 0 \vee i > len(t_1)}_{0<0\vee 0>0} \vee, \underbrace{i_{t_1} - 1 < 0 \vee i - 1 > len(t_1)}_{-1<0\vee -1>0}.$$

The first atom of the disjunction $0.1 < \epsilon[0] - \epsilon[-1]$ *may be true or false (*$\epsilon[-1], \epsilon[0]$ *return random instant values). All other atoms are false but* $-1 < 0$*. This makes the hole formula evaluated to true anyway. Consequently, the at the first execution of the transition this formula is satisfied. This is consistent with our interpretation of such timing formula constraining two successive occurrence instants: the formula is ignored at the first occurrence of the transition and is relevant starting from the second occurrence.*

We now define the TIOLTS associated with a TIOSTS which is mainly constructed by building TIOLTS transitions from all runs of all transitions of the TIOSTS and by adding other one that will be discussed after Definition 40.

Definition 40 (TIOLTS associated with an TIOSTS) *Let* $\mathbb{G}$ *be an TIOSTS* (Q, q_0, T) *over* $\Sigma = (A, C)$*.*
The TIOLTS associated with $\mathbb{G}$*, denoted* $LTS_{\mathbb{G}} = (Snp_M(\mathbb{G}) \cup \{init, q_\delta\}, init, T')$ *over* $Act_M(C)$*, is such that:*

- $init, q_\delta$ *are two distinct (arbitrary) states satisfying* $init, q_\delta \notin Snp_M(\mathbb{G})$
- T' *is the smallest subset of* $(Snp_M(\mathbb{G}) \cup \{init, q_\delta\}) \times (Act_M(C) \cup M_I) \times (Snp_M(\mathbb{G}) \cup \{q_\delta\})$ *such that:*
  - **Initialization transitions** *for any* $\nu \in M^A$ *such that for all* $x \in A_{I^*}$ *we have* $\nu(x) = \epsilon$*,* $(init, \tau, (q_0, 0, \nu))$ *is in* T'*,*
  - **Transitions of runs** *for all* $tr \in T$*, for any* $r \in Run(tr)$ *of the form* $((q, \mathcal{T}, \nu), act_M, (q', \mathcal{T}', \nu'))$*, we have* $((q, \mathcal{T}, \nu), \delta(r), (q, \mathcal{T}', \nu))$ *and* $((q, \mathcal{T}', \nu), act_M, (q', \mathcal{T}', \nu'))$ *are in* T'*,*
  - **Quiescence** *Let* $snp \in Snp_M(\mathbb{G}) \cup \{init\}$ *be a snapshot such that for all transitions of the form* $(snp, act_M, snp') \in T'$ *with* $snp' \in Snp_M(\mathbb{G})$*,* act_M *is of the form* $c?v$*, we have for any* $d \in M_I$*,* (snp, d, q_δ) *is in* T'*.*

The state *init* is the initial state of $LTS_{\mathbb{G}}$. It is not built as a snapshot over q_0 because we have to consider any arbitrary assignment of variables (regardless of time variables which has to be assigned by ϵ denoting the empty array). Those possible assignments are taken into account in the **Initialization transitions** item. Now transitions of TIOLTS introduces either communication actions or durations. In item **Transitions of runs**, we show how to decompose any run of any transition into two TIOLTS transitions: the first one denotes a delay before the observation of the action, and the second one introduce the action itself.

The third item **Quiescence** refers to the so-called *quiescence situations* as initially introduced by Jan Tretmans [82]. For some state *snp* of $LTS_{\mathbb{G}}$ (or for *init*), it may happen that no transition *tr* such that $source(tr) = snp$ that may be built considering the two previous items (if any) introduces an input action. It means that no (if any) reactions of the system are specified from the state. We interpret that situation by considering that the system does not react and represent this fact adding additional transitions in T'.

We can now simply define the semantics of a TIOSTS as the set of timed traces of its associated TIOLTS.

Definition 41 (Traces of a TIOSTS) *With notations of Definition 40, the* semantics of $\mathbb{G}$ *denoted* $Sem(\mathbb{G})$ *is the set* $TTraces(LTS_{\mathbb{G}})$.

5.3 Symbolic Execution

In the previous Section 5.2, we have seen how to define the semantics of a TIOSTS in terms of traces. In this section we show how to compute this semantics in order to represent it in intention thanks to symbolic execution techniques.

Symbolic execution was initially defined for programs [52] and extended to IOSTS [36, **?**]. Symbolic execution of TIOSTS, as for the case of IOSTS and programs, simply consists in executing the TIOSTS, not for concrete input values, but for symbolic ones, and to reason on those symbolic values to characterize the set of all possible executions (*i.e.* traces) of the TIOSTS. The main difference with IOSTS is that we have to define the symbolic treatment of time variables. We begin by introducing the notion of a *symbolic state* which is a structure used to store pieces of information concerning an execution. In order to represent symbolic values, we suppose that a set of variables $F = \bigcup_{s \in S} F_s$, disjoint of any set of variables introduced in TIOSTS signatures, is given.

Definition 42 (symbolic state) *For a TIOSTS* $\mathbb{G} = (Q, q_0, T)$ *over* (A, C), *a* symbolic state *over* F *is a quadruple* $\eta = (q, \pi_t, \pi_d, \mathcal{T}, \sigma)$ *where* $q \in Q$, $\pi_t \in \mathscr{T}_\Omega(F)$, $\pi_d \in Sen_\Omega(F)$, $\mathcal{T} \in T_\Omega(F_{I^*})$ *and* σ *is a function of variables of* A *in* $T_\Omega(F)$ *preserving types. We note* $\mathcal{S}$ *the set of all the symbolic extended states over* F.

q denotes the state reached after the execution leading to η, π_t is a constraint on symbolic execution instant values called *time path condition* and π_d is a constraint on symbolic data values called *data path condition*. Both constraints must be satisfied for the execution to reach η, $\mathcal{T}$ denotes the current instant and σ denotes the current terms (built over symbolic values) assigned to variables of A.

Notation 10 *In the sequel,* Σ_F *stands for* (F, C). *Moreover for any symbolic state* $\eta = (q, \pi_t, \pi_d, \mathcal{T}, \sigma)$, $q(\eta)$, $\pi_t(\eta)$,$\pi_d(\eta)$, $\mathcal{T}(\eta)$ *and* $\sigma(\eta)$ *stand respectively for* q, π_t, π_d, $\mathcal{T}$ *and* σ. *For any* $\sigma : A \to T_\Omega(F)$ *we also note* $\sigma : T_\Omega(A) \to T_\Omega(F)$ *and* $\sigma : Sen_\Omega(A) \to Sen_\Omega(F)$ *its canonical extensions respectively to terms and formulae. We also note* $\sigma : Act(\Sigma) \to Act(\Sigma_F)$ *its extension to communication actions defined as* $\sigma(c?x) = c?\sigma(x)$, $\sigma(c!t) = c!\sigma(t)$, $\sigma(new(x)) = \tau$ *and* $\sigma(\tau) = \tau$.

When generating the symbolic execution tree, it may happen that some symbolic states are not reachable. That is, the time and data path conditions of the symbolic state are not satisfiable. Thus, we define the set of satisfiable symbolic states.

Definition 43 (Satisfiable symbolic states) *Let* $\mathcal{S}$ *be the set of all symbolic extended states over* F. $\mathcal{S}_{sat}$ *is the set of all symbolic extended states of the form* $(q, \pi_t, \pi_d, \mathcal{T}, \sigma)$ *for which there exists an interpretation* ν *in* M^F *such that:* $\nu \models \phi_t$ *and* $\nu \models \phi_d$.

Similarly to the way we defined TIOLTS associated with a TIOSTS by starting to define runs of a transition, symbolic execution of TIOSTS is based on the symbolic execution of a transition.

Definition 44 (symbolic execution of a transition) *With notations of Definition 42, for any symbolic state* η *and* $tr = (q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q') \in T$ *with* $q = q(\eta)$, *a* symbolic execution *of*

tr from η is a triple $st = (\eta, act_F, \eta') \in \mathcal{S} \times Act(\Sigma_F) \times \mathcal{S}$ such that $q(\eta') = q'$, $\mathcal{T}(\eta')$ is of the form $\mathcal{T}(\eta) + \delta$ where $\delta \in F_I$ is a new fresh variable, there exists $\sigma^i : A \to T_\Omega(F)$ satisfying:

- *for all $x \in \mathbb{T}$, we have $\sigma^i(x) = pushArr(\sigma(\eta)(x), \mathcal{T}(\eta'))$,*
- *if act is of the form c!t (respectively τ) then for all $x \in A \setminus \mathbb{T}$ we have $\sigma^i(x) = \sigma(\eta)(x)$,*
- *if act is of the form c?x (respectively new(x)) then $\sigma^i(x)$ is a new fresh variable and for all $x \in A \setminus (\mathbb{T} \cup \{x\})$ we have $\sigma^i(x) = \sigma(\eta)(x)$,*

such that $act_F = \sigma^i(act)$, for all $x \in A_{rw}$ we have $\sigma(\eta')(x) = \sigma^i(\rho(x))$, for all $x \in A_{I^*}$ we have $\sigma(\eta')(x) = \sigma^i(x)$, $\pi_t(\eta') = \pi_t(\eta) \wedge \sigma^i(WF(\phi_t))$ and $\pi_d(\eta') = \pi_d(\eta) \wedge \sigma^i(\phi_d)$.

Notation 11 *st is called a* symbolic execution of *tr* from *η*. *δ is called the* duration of *st and is denoted $\delta(st)$*. *tr is called the* ground transition *of st and is denoted $g(st)$*. *$source(st)$, $act(st)$ and $target(st)$ stand respectively for η, act_F and η'.*

We note $Fresh(st) = \{\delta(st)\}$ if act is an output or τ and $Fresh(st) = \{\delta(st), \sigma^i(x)\}$ if act is of the form c?x for some $c \in C$ or of the form $new(x)$.

Instants at which actions occur are denoted by sum of symbolic durations introduced in the course of the symbolic execution ($\Delta_i = \sum_{j=0}^{i} \delta_j$). Similarly, $\Delta_{k \to i}$ means $\sum_{j=k}^{i} \delta_j$.

Note the strong similarity between Definition 44 and Definition 39 of runs. Clearly Definition 44 symbolic intentional representation of sets of runs of transitions.

$\mathcal{T}(\eta')$ is the instant at which the actions act_F occurs. The intermediate substitution σ^i is $\sigma(\eta)$ except that it updates values of time variables occurring in $\mathbb{T}$ by adding the new symbolic instant $\mathcal{T}(\eta')$ in arrays assigned to them. σ^i also redefines values of variables occurring in actions of the form $c?x$ or $new(x)$ to reflect that the value of x has changed due respectively to a value reception or a random updating. act_F is simply the symbolic interpretation of *act* and $\sigma(\eta')$ is obtained by taking into account the substitution ρ from σ^i. $\pi_t(\eta')$ and $\pi_d(\eta')$ are formulas respectively on time and data that must be satisfied so that the transition can be executed.

Example 29 *Consider again the transition tr as an illustration of Definition 44 :*

$$q_1 \xrightarrow[\substack{env?x_{env} \\ i_{t_1} \leftarrow i_{t_1}+1}]{\{t_1\}\ 0.1<t_1[i_{t_1}]-t_1[i_{t_1}-1],x_{env}=\text{"}start\text{"}} q_2$$

We have discussed a concrete run of the transition tr in Example 28, in a similar way we give a possible symbolic execution of tr :

$$(q_1, true, true, 0, \sigma) \xrightarrow{env?x_{env}\#1} (q_2, \pi_t^0, \pi_d^0, \delta_0, \sigma_0),$$

	substitution of variables
where	$\sigma(t_1) = emptyArr$, $\sigma(t_2) = emptyArr$, $\sigma(t_3) = emptyArr$ $\sigma(i_{t_1}) = 0$, $\sigma(i_{t_3}) = 0$ $\sigma(x_{env}) = x_{env}\#0$, $\sigma(x_{cmd}) = x_{cmd}\#0$, $\sigma(x_{data}) = x_{data}\#0$
	$\sigma_0(t_1) = pushArr(emptyArr, \delta_0) = \delta_0$, $\sigma_0(t_2) = emptyArr$, $\sigma_0(t_3) = emptyArr$ $\sigma_0(i_{t_1}) = 1$, $\sigma_0(i_{t_3}) = 0$ $\sigma_0(x_{env}) = x_{env}\#1$, $\sigma_0(x_{cmd}) = x_{cmd}\#0$, $\sigma_0(x_{data}) = x_{data}\#0$

time path condition
$\underbrace{0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]}_{0.1<emptyArr[0]-emptyArr[-1]} \vee \underbrace{i_{t_1} < 0 \vee i_{t_1} > len(t_1)}_{0<0\vee 0>0} \vee, \underbrace{i_{t_1} - 1 < 0 \vee i_{t_1} - 1 > len(t_1)}_{-1<0\vee -1>0}$ $\pi_t^0 = true$
data path condition
$\pi_d^0 = x_{env}\#1 = "start"$

Same reasoning as in Example 28, we have $0.1 < emptyArr[0] - emptyArr[-1]$ *may be true or false* ($emptyArr[0], emptyArr[-1]$ *return irrelevant fresh instant values). However* π_t *is true as a hole because* $-1 < 0$ *is true. This was a symbolic execution of* tr *corresponding to the first occurrence of the reception action* $env?x_{env}$ *of* tr *in the system* ($\sigma(t_1) = emptyArr$ *then* $\sigma_0(t_1) = \delta_0$*).*

Example 30 *Continuing the example 29, consider the following symbolic execution of* tr*, it corresponds to a second occurrence of* $env?x_{env}$ *(in between, some other preceding transitions have been symbolically executed). The goal is to illustrate how the symbolic execution handle the timing guard* $0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]$ *when the the array of instants* t_1 *associated with the action* $env?x_{env}$ *contains relevant values.*

$$(q_1, \pi_t^4, \pi_d^4, \Delta_4, \sigma_4) \xrightarrow{env?x_{env}\#2} (q_2, \pi_t^5, \pi_d^5, \Delta_5, \sigma_5),$$

where

substitution of variables
$\sigma_4(t_1) = \delta_0,$ $\sigma_4(i_{t_1}) = 1,$ $\sigma_4(x_{env}) = x_{env}\#1, \ldots$
$\sigma_5(t_1) = pushArr(\delta_0, \Delta_5) = \delta_0.\Delta_5,$ $\sigma_5(i_{t_1}) = 2,$ $\sigma_5(x_{env}) = x_{env}\#2, \ldots$

time path condition
$\underbrace{0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]}_{\underbrace{0.1 < \delta_0.\Delta_5[1] - \delta_0.\Delta_5[0]}_{0.1<\Delta_{1\to 6}}} \vee \underbrace{i_{t_1} < 0 \vee i_{t_1} > len(t_1)}_{1<0\vee 1>2} \vee \underbrace{i_{t_1} - 1 < 0 \vee i_{t_1} - 1 > len(t_1)}_{0<0\vee 0>2}$ $\pi_t^5 = \pi_t^4 \wedge 0.1 < \Delta_{1\to 5}$
data path condition
$\pi_d^5 = \pi_d^4 \wedge x_{env}\#2 = "start"$

Since the first occurrence of action $env?x_{env}$ *(corresponding to the symbolic action* $env?x_{env}\#1$*) at time instant* δ_0*, time elapsed of* $\Delta_{1\to 5} = \delta_1 + \ldots + \delta_5$ *when* $env?x_{env}$ *occurs again (that is at* $\Delta_5 = \delta_0 + \ldots + \delta_5$ *corresponding to the symbolic action* $env?x_{env}\#2$*). The timing guard* $0.1 < t_1[i_{t_1}] - t_1[i_{t_1} - 1]$ *equals* $0.1 < \Delta_{1\to 5}$ *for* i_{t_1} *and* t_1 *are mapped respectively to* 1 *and* $\delta_0.\Delta_5$*.*

The symbolic execution tree associated with the TIOSTS is then defined simply by executing exactly once all executable transitions from all symbolic states. We now introduce the symbolic tree of a TIOSTS.

Definition 45 (symbolic tree of a TIOSTS) *With notations of Definition 44, a symbolic execution of* $\mathbb{G}$ *is a couple* $T(\mathbb{G}) = (Init, T)$ *where:*

- *Init is a symbolic state of the form* $(q_0, true, true, 0, \sigma)$ *where for all* $x \in A_{I^*}$ *we have*

$\sigma(x) = emptyArr$ *and for all* $x, y \in A \setminus A_{I^*}$ *we have* $x \neq y \Rightarrow \sigma(x) \neq \sigma(y)$ *and* $\sigma(x) \in F$,

- T *is a set of symbolic transitions such that for any* $\eta \in \mathcal{S}$ *and for any* $tr \in T$ *there exists exactly one symbolic execution of* tr *from* η. *Moreover for any two* $st_1, st_2 \in T$ *we have* $Fresh(st_1) \cap Fresh(st_2) = \emptyset$.

Notice that $T(\mathbb{G})$ has a tree-like structure whose all paths denote in an abstract way all possible executions of $\mathbb{G}$. At the beginning of the execution, there is no constraint on time and on data as it is signified by the two occurrences of *true* respectively for time path condition and data path condition in the symbolic state $Init$.

Always concerning the initial states, time variables are initialized to the empty array and all other variables are initialized with fresh variables of F. In order to make no suppositions on their initial values, we impose the assignment to be injective.

Symbolic execution of a TIOSTS is simply the restriction of $T(\mathbb{G})$ to satisfiable symbolic extended states.

Definition 46 (symbolic tree of a TIOSTS) *The symbolic execution of* $\mathbb{G}$, *denoted* $SE(\mathbb{G})$ *is the couple* $(Init, ST)$ *where* ST *is the set of all* (η, act, η') *in* T *such that* η' *in* $\mathcal{S}_{sat}$.

Example 31 *Figure 5.3 depicts the symbolic execution for the TIOSTS of Figure 5.1a. For the readability sake, apart from the initial state Init of the symbolic tree which is given in details, only changes in affectations are shown inside the remaining symbolic states. The symbolic execution is shown until symbolic states* η_5 *and* η'_4. *In* η_5, *a new cyclic behavior of* $\mathbb{G}^{ctrl}$ *is re-visited for the second time. In* η'_4, *the execution stops. Note that in the branching state* η_3, *there is only one decision that had to be made depending on the value of the variable* x_{data} *(represents the data measure received from the sensor). If its value (*$x_{data}\#1$*) is below the threshold (*$x_{data}\#1 < 500$*), then the left branch is taken. The right branch represents the state of the system where* $x_{data}\#1 \geq 500$ *and no transition can fire anymore.*

Now as the reader can see in Definition 40, the TIOLTS associated with a TIOSTS introduces transitions reflecting quiescence and time passing. Those transitions have no counterpart in a symbolic tree. Next definition shows how to complete a symbolic tree with transitions reflecting quiescence and time passing.

Definition 47 (Symbolic execution with quiescence) *With notations of Definition 46, the* quiescence and time passing *enrichment of* $SE(\mathbb{G})$ *denoted* $SE(\mathbb{G})_\delta$ *is the couple* $(Init, ST \cup ST_\delta)$ *where* ST_δ *is defined as follows :*

For all $\eta \in \mathcal{S}$, *let us note* $React(\eta)$ *the set of all transitions of* ST *such that* $str \in React(\eta)$ *if and only if* $source(st) = \eta$ *and* $act(st)$ *is* τ *or an output.*

- **data based quiescence** *Let us note* $\pi_d^\delta(\eta)$ *the formula restricted to true if* $React(\eta) = \emptyset$ *and equal to* $\wedge_{str \in React(\eta)} \neg \pi_d(target(str))$. *Let us note* η_d^δ *the symbolic state* $(q_\delta, \pi_t(\eta), \pi_d(\eta) \wedge \pi_d^\delta(\eta), \mathcal{T}(\eta) + \delta, \sigma(\eta))$ *where* δ *is a new fresh variable in* F_I. *We have then* $(\eta, \tau, \eta_d^\delta) \in ST_\delta$.
- **time based quiescence** *Let us note* $\pi_t^\delta(\eta)$ *the formula restricted to true if* $React(\eta) = \emptyset$ *and equal to* $\wedge_{str \in React(\eta)} \forall \delta(str).(\neg \pi_t(target(str)))$. *Let us note* η_t^δ *the symbolic state* $(q_\delta, \pi_t(\eta) \wedge \pi_t^\delta(\eta), \pi_d(\eta), \mathcal{T}(\eta) + \delta, \sigma(\eta))$ *where* δ *is a new fresh variable in* F_I. *We have then* $(\eta, \tau, \eta_t^\delta) \in ST_\delta$.

Example 32 *Figure 5.4 depicts the application of Definition 47 on some states of the symbolic execution tree in Figure 5.3.*

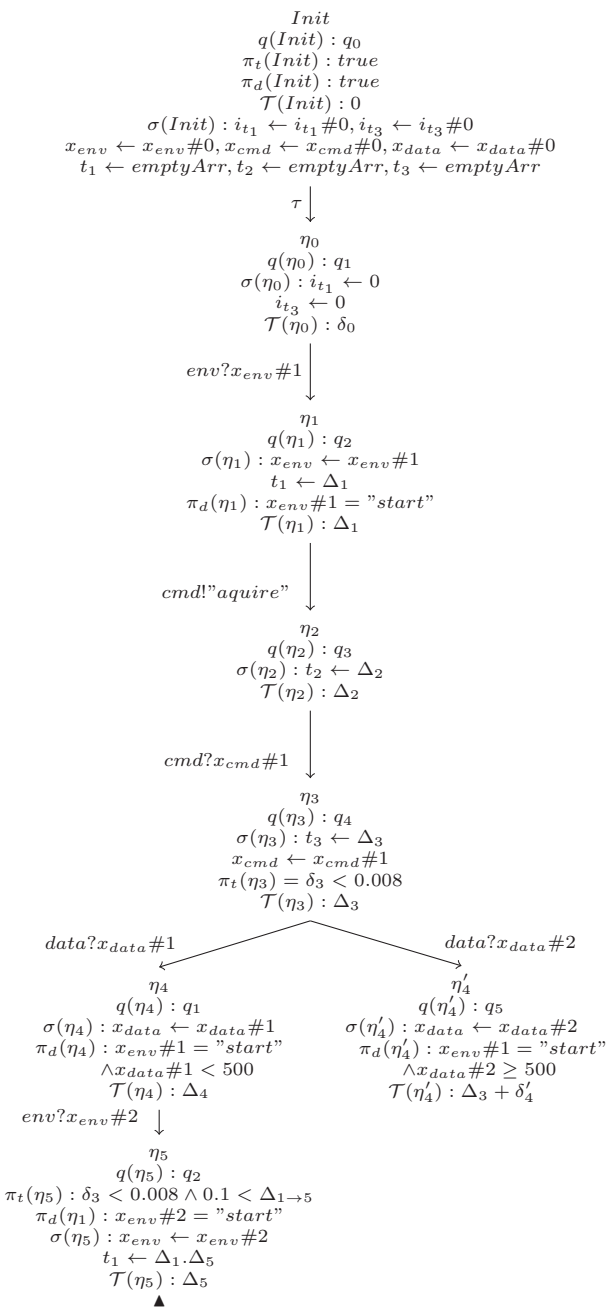


Figure 5.3: Symbolic tree

$SE(\mathbb{G})_\delta$ characterizes in an intentional way the set of all traces of the TIOLTS associated with $\mathbb{G}$. In the remaining of this section we show how to compute such traces. To reach that goal we begin by characterizing paths of $SE(\mathbb{G})_\delta$.

Definition 48 (Paths of $SE(\mathbb{G})_\delta$) *The set of* paths *of $SE(\mathbb{G})_\delta$ denoted $Path(SE(\mathbb{G})_\delta)$, contains all the finite sequences $st_1 \cdots st_n$ of transitions of $ST \cup ST_\delta$, such that:*

- $source(st_1) = Init$
- *for every* i, $1 \leq i \leq n$, $target(st_i) = source(st_{i+1})$

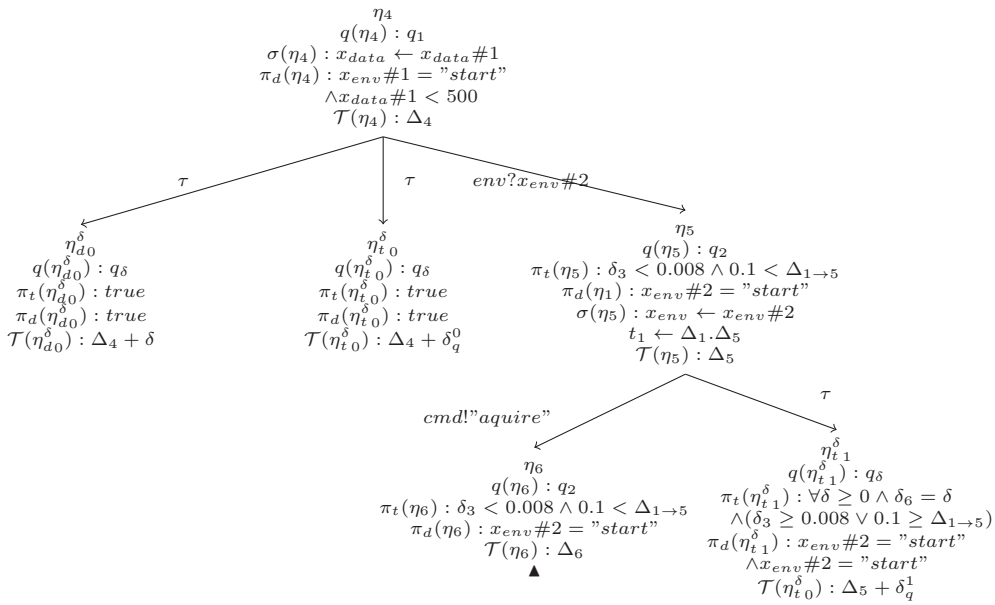


Figure 5.4: Symbolic tree enrichment

In the sequel, for any finite path p, $target(p)$ is $Init$ if p is empty and is the target state of its last transition otherwise.

Example 33 *Let us consider the symbolic execution of Figure 5.4. A finite path of that tree is the sequence of symbolic transitions going from the root state init until the symbolic state η_5, that is, the sequence:*

$$(init, \tau, \eta_0).(\eta_0, env?x_{env}\#1, \eta_1).(\eta_1, cmd!"aquire", \eta_2).(\eta_2, cmd?x_{cmd}\#1, \eta_3).(\eta_3, data?x_{data}\#1, \eta_4.$$

$$(\eta_4, env?x_{env}\#2, \eta_5)$$

In the sequel, we pay attention to particular symbolic execution trees (with quiescence) in which there do not exists arbitrary long sequences of transitions introducing τ actions. We call them "livelock free" symbolic execution trees.

Definition 49 (Livelock free symbolic tree) *$SE(\mathbb{G})_\delta$ is livelock free if and only if exists N in $\mathbb{N}$ such that for any path of $Path(SE(\mathbb{G})_\delta)$ of the form $st_1 \dots st_m$ for any i and j between 1 and m with $i \leq j$ satisfying: for all k in $\mathbb{N}$, $(k \geq i \wedge k \leq j) \Rightarrow act(st_k) = \tau$, we have $N \geq j - i$.*

If $\mathbb{G}$ is such that it does not contain any path with such sequences of transitions with τ actions of length greater than N, then $SE(\mathbb{G})_\delta$ is livelock free. However this restriction is very strong and the livelock freedom of $SE(\mathbb{G})_\delta$ may be satisfied while $\mathbb{G}$ does not have this property. Indeed the bound on the number of consecutive transitions with τ may come from constraints on time or data. When $SE(\mathbb{G})_\delta$ is livelock free it is possible to associate its paths their *τ-reduction* versions. The $\tau-$ reduced version of a path is a path characterizing the same sequence of actions than p with the same constraints but in which all occurrences of τ disappear. It is depicted as following:

Definition 50 (τ-reduction of a path) *Let p be a path in $Path(SE(\mathbb{G})_\delta)$ such that if we note p as $p'.st$ where $st \in ST \cup ST_\delta$ then either $act(st)$ is an input or an output or $state(target(st)) = q_\delta$. The τ-reduction of p denoted $\tau(p)$ is inductively defined as follows:*

- *if p is ϵ then $\tau(p)$ is ϵ,*
- *if p is of the form $p'.st$, we have:*
  - *if $act(st)$ is τ and $state(target(st)) \neq q_\delta$ then $\tau(p) = \tau(p')$,*
  - *if $act(st)$ is not τ or if $state(target(st)) = q_\delta$ then $\tau(p) = \tau(p').(target(\tau(p')), act(st), target(st))$.*

Example 34 *Let p_1 be $(Init, \tau, \eta_1).(\eta_1, c!u, \eta_2)$, the τ-reduction of p_1 is $\tau(p_1) = (Init, c!u, \eta_2)$. Consider now the path $p_2 = (Init, c!u, \eta_1).(\eta_1, \tau, \eta_2)$ where $state(\eta_2)$ is q_δ. In this case we have $\tau(p_2) = p_2$.*

The τ-reduction of p characterizes the same executions than p, only keeping the sequence of observable actions occurring in p. In the sequel, as we did in Definition 44, for any st occurring in $\tau(p)$, $\delta(st)$ stands for $\delta(target(st)) - \delta(source(st))$. Similarly to the case of transitions of a symbolic execution where $\delta(st)$ is simply a variable of F_I, in the case of a transition of a τ-reduced path, we may have $\delta(st)$ is a sum of variables of F_I.

τ-reduction of a path is a kind of normalization. We have to define another normalization process, this time on timed traces. It simply consists in summing all the consecutive durations of the trace (similar to [81]).

Definition 51 (Normalization of a trace) *Let σ be a timed trace which is either ϵ or of the form $\sigma'.act$ where act is an output or input and σ' begins by a duration. The* normalization of σ *denoted $norm(\sigma)$ is defined as follows:*

- *if σ is ϵ, we have $norm(\sigma)$ is 0,*
- *if σ is a sequence $d_1 \dots d_n$ of durations in M_I, we have $norm(\sigma)$ is $\Sigma_{i=1}^{n} d_i$,*
- *if σ is of the form $\sigma'.act$ where act is an input or an output, we have $norm(\sigma)$ is $norm(\sigma').act$,*
- *if σ is of the form $\sigma'.d_1 \dots d_n$ where σ' is of the form $\sigma''.act$ and act is an input or output, we have $norm(\sigma)$ is $norm(\sigma').\Sigma_{i=1}^{n} d_i$.*

Example 35 *Let σ be the trace $(0.1).(0.1).(0.1).c!200$. The normalization of σ is the trace $(0.3).c!200$.*

We are now in position to define the traces of a path.

Definition 52 (Traces of a symbolic path) *Let σ be a timed trace and p be a path in $Path(SE(\mathbb{G})_\delta)$. We note $Indent(\sigma, p)$ the formula defined as follows:*

- *if $norm(\sigma)$ is ϵ and $\tau(p)$ is ϵ, $Ident(\sigma, p)$ is True,*
- *if $norm(\sigma)$ is ϵ and $\tau(p)$ is not ϵ, $Ident(\sigma, p)$ is False,*
- *if $norm(\sigma)$ is not ϵ and $\tau(p)$ is ϵ, $Ident(\sigma, p)$ is False,*
- *if $norm(\sigma)$ is of the form $\sigma'.d$ where d is in M_I, let us note $\tau(p)$ as $st_1 \dots st_m$. We have $Indent(\sigma, p)$ is $Indent(\sigma', st_1 \dots st_{m-1}) \wedge d = \delta(st_m)$,*
- *otherwise let us note $norm(\sigma)$ as $\sigma'.d.act$ where d is duration (in M_I) and act is an input or an output. Let us note $\tau(p)$ as $p'.st$:*

- *if act is of the form c!v (respectively c?v) and act(st) is of the form c!u (respectively c?u), we have Ident(σ, p) is Ident(σ′, p′) ∧ v = u ∧ d = δ(st),*
- *if act is of the form c!v (respectively c?v) and act(st) is not of the form c!u (respectively c?u), we have Ident(σ, p) is False.*

We say that σ belongs to *p if and only if Ident(σ, p) ∧ φ_t(target(p)) ∧ φ_d(target(p)) is satisfiable, that is there exists an interpretation ν in M^F such that ν ⊨ Ident(σ, p) ∧ φ_t(target(p)) ∧ φ_d(target(p)). TTraces(p) is the set of all timed trace that belong to p.*

Example 36 *Let σ be the trace* (0.1).(0.1).(0.1).c!200 *and p be the path* (Init, τ, η1).(η1, c!u, η2). *Besides we have that φ_t(η2) = δ1 < 0.5 and φ_d(η2) = u < 700. We have already seen that norm(σ) = (0.3).c!200 and τ(p) = (Init, c!u, η2). Given that δ((Init, c!u, η2)) = δ0 + δ1. By applying Definition 52, we obtain Ident(σ, p) = True ∧ 200 = u ∧ 0.3 = δ0 + δ1. The formula to be satisfied so that σ belongs to p is:* 200 = u ∧ 0.3 = δ0 + δ1 ∧ δ1 < 0.5 ∧ u < 700. *This formula is obviously satisfiable. We deduce that σ belongs to p.*

Finally, we state in the following definition when a trace belongs to a symbolic execution.

Definition 53 (Traces of a symbolic execution) *Let σ be a timed trace. We say that σ* belongs to *SE(G_δ) if and only if exists a path p in Path(SE(G)_δ) such that σ belongs to p. TTraces(SE(G_δ)) is the set of all timed traces that belong to SE(G_δ).*

5.4 Related work

Recently, some authors have suggested seemingly different approaches to represent symbolically both time and data (approaches in [87], [5], and [31]). Time is handled by means of clocks whose values constrain occurrences of actions in the timed automata style. In all these works, time instants at which actions occur are not explicitly referred to as it is the case in timing annotations on sequence diagrams. We show in this section that the timing constraints expressing relations between time instants can be encoded using clocks. However, our TIOSTS formalism allows a more straightforward encoding of these kind of constraints (as they are formulated in sequence diagrams) thanks to special variables capturing time instants of occurrences. In the following, we present the approaches [87, 5, 31] while putting more emphasis on [87]. This one serves as a proof of concept showing some clock-based patterns which can encode the kind of timing constraints discussed before.

TA were extended to support symbolic treatment of data based on first order logic into Symbolic Timed Automata (STA) [87]. We give some transitions in Figure 5.5 of the STA of the *Beverage Vending Machine*. The system accepts money, allows to choose a beverage, serves it within a parameterizable delay depending on the nature of the beverage and returns the change. Transitions in STA define input or output communication like actions with parameters (e.g. the input $?money < x >$, where x is a data parameter representing the money received from the user; the input $?choice < y, t >$ where y is the choice of beverage and t is the delay to serve the beverage; the output $!serve < y, x >$, where y represents the served beverage and x is the change returned back to the user). The transitions define two kinds of assignments. Classically, there are variable assignments (e.g. $money \rightarrow x$, the variable *money* assigned to x and thus stores the money received by the user when $?money < x >$ is performed; $time \rightarrow t$ means that the delay t given by the user through $?choice < y, t >$ to serve the beverage is stored in the variable *time*). There are clock assignments consisting exactly in clock resets. An example of a clock reset is $c := 0$ where c is a clock variable. In fact a clock is a special variable storing delays such that its value evolves spontaneously as time elapses since it is reset.

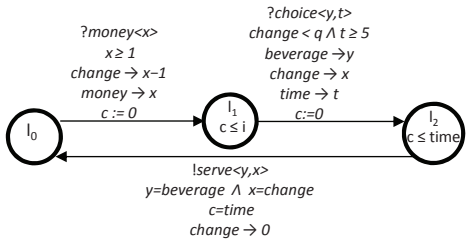


Figure 5.5: STA transitions as in [87]

Transitions are labeled by a data guard and a clock guard. The data guard $change < q$ for instance signifies that change has to be available to execute the transition (it has to be lower than the cash q in the machine). The time guard $c = time$, where c is a clock and $time$ is variable used to control time ($time$ was set to the delay t after $?choice < y, t >$), makes the transition execute when the time elapsed is exactly $time$. States, i.e. locations, may also be labeled by guards called invariants which restrict the way time may elapse in a location. An example of such invariants is $c \leq time$ in location l_2 which together with the guard $c = time$ of the only outgoing transition from l_2 ensures that location l_2 is left after exactly $time$ time units.

On the whole, a part from symbolic representation of data, STA handle time with clocks conforming to TA syntax. This is the case of the TIOSTS in [5] (except state invariants which are not considered).

Note that the same name "TIOSTS" is given to the automaton formalisms in [5], [31] and ours. We make sure to mention the reference of the approach when the context is confusing.

In fact TIOSTS [5] extend a variant of a symbolic transition system IOSTS [80] with time as in TA style. Figure 5.6 shows examples of transitions of a TIOSTS as defined in [5] that models a withdrawal transaction in an ATM system.

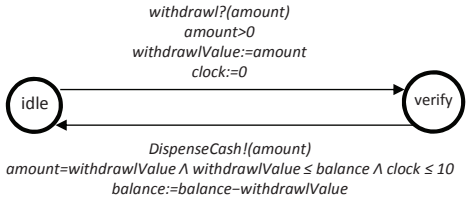


Figure 5.6: TIOSTS transitions as in [5]

An input or output communication action may be associated with a transition (e.g. $DispenseCash!(amount)$ dispenses the value in $amount$). Clocks may be reset to zero ($clock := 0$, where $clock$ is a clock). A single guard constrains the execution of a transition which ranges over time and data (e.g. the guard $amount = withdrawlValue \wedge withdrawlValue \leq balance \wedge clock \leq 10$ means that the value of $withdrawalValue$ is less than the $balance$ and the time elapsed represented by $clock$ is less than 10 time units).

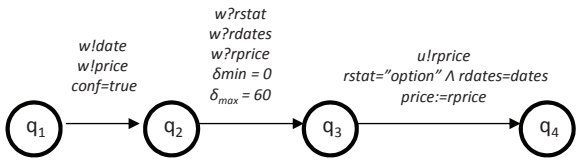


Figure 5.7: TIOSTS transitions as in [31]

In the same spirit of our work, IOSTS [36] were extended with time into TIOSTS [31]. Examples of transitions of such TIOSTS which model an *orchestration* of Hotel Reservation service are

illustrated in Figure 5.7. A single transition of such a TIOSTS may define a sequence of communication actions (e.g. $w!date; w!price$ denoting the outputs *date* and *price* communicated to the web service). Time is handled by means of implicit clocks. In fact, a transition has a minimal and a maximal delay to be fired (respectively δ_{min} and δ_{max}) as if the transition is associated with one clock reset after each measure (e.g. the answer from the Hotel Web service $w?rstat; w?rdates; w?rprice$ must arrive before $\delta_{max} = 60$ time units). These TIOSTS can be seen as restrictions of STA with one clock per transition. The transition format here reflects the common usage of timers in orchestrator descriptions (typically when using WS-BPEL).

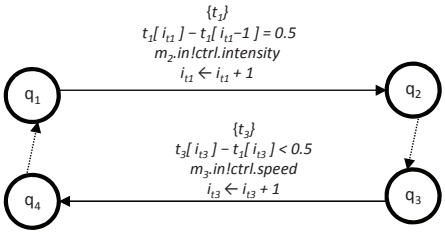


Figure 5.8: TIOSTS transitions as in [12] (ours)

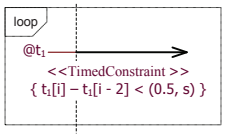
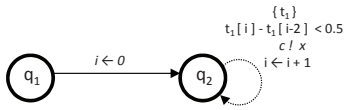
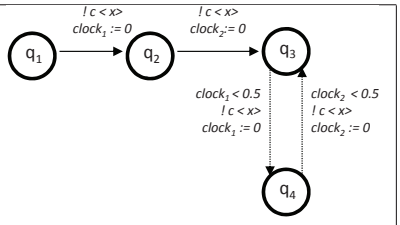
Finally we introduce our version of TIOSTS. The Figure 5.8 depicts TIOSTS transitions corresponding to some behavior of a Rain-sensing wiper control system that have already specified as a sequence diagram in Figure 4.1. The idea is to capture explicitly instants in time of occurrences of actions rather than delays between them. For that purpose, transitions are labeled by variables which are arrays capturing time instants as discussed in Section 4.2 (e.g. t_1, t_3 are examples of such a variables). This allows one to deal naturally with a particular kind of MARTE timing constraints that may annotate a sequence diagram stating that: "two successive stimuli are spaced of 0.5 time slots", typically in a periodic sampling or "an output occurs 0.5 time slots after the beginning of the period" (Example of such constraints are respectively $t_1[i] - t_1[i-1] = 0.5$ where the period starts at $t_1[i]$ when the system gets a new rain intensity and $t_3[i] - t_1[i] < 0.5$ where the wiper speed is produced at $t_3[i]$. See sequence diagram in Figure 4.1).

It is possible to encode such guards constraining two occurrence instants of the same action using clocks. An example is given for the guard $t_1[i] - t_1[i-2] < 0.5$ with the STA formalism in Table 5.1 (see first row). Another example specifying the guard $t_2[i] - t_1[i] < 0.1$ which relates occurrence instants of two different actions is also illustrated in the table (see second row).

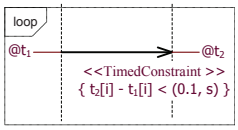
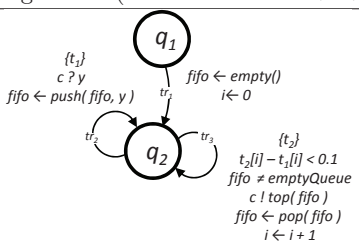
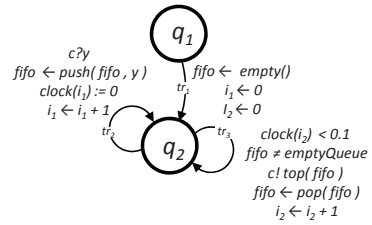
Recall that the guard $t_1[i] - t_1[i-2] < 0.5$ states that there is a delay of at most of $0.5s$ between the i^{th} and $i + 2^{th}$ occurrences of the message sending, which is annotated by t_1 in the sequence diagram (see first row, first column). Similarly to the example in Figure 5.8, this constraint is trivially captured by our TIOSTS formalism. Note that the first and the second occurrences may occur at any time. Starting from the third occurrence, the constraint must hold. We have specified this constraint also using STA (see first row, third column). This required two clocks $clock_1$ and $clock_1$. $clock_1$ measures the time elapsed since the first (inductively since the i^{th}) occurrence and $clock_2$ measures the time elapsed since the second (inductively since the $i + 1^{th}$) occurrence. In this way, at the third (inductively $i + 2^{th}$) occurrence, $clock_1$ contains the delay to constrain ($clock_1 < 0.5$) and is then reset (transition $q_3 \rightarrow q_4$). Meanwhile, $clock_2$ is recording time such that at the forth (inductively $i + 3^{th}$) occurrence, the time elapsed respects as well the deadline ($clock_2 < 0.5$) and then (again) $clock_2$ is reset (transition $q_4 \rightarrow q_3$).

The example of the second row is a typical example of the translation into a TIOSTS of an asynchronous message in our framework. A fifo queue receives the values cyclically (transition tr_2). Concurrently, values stored in the fifo are retransmitted (tr_3). A deadline of $0.1s$ is specified for the transmission delay of each received piece of data. The difficulty is that many values (not known beforehand, depending on the sender operating rate) may be received and enqueued before

Table 5.1: Specification of MARTE timing constraints relating instants : Comparison of the expressiveness of TIOSTS (ours) and STA [87]

Timing constraint	TIOSTS (ours)	STA
		

Timing constraint relating occurrence instants of same action :
there is a delay of at most of $0.5s$ between the i^{th} and $i + 2^{th}$
of the same sending action (noted $c!x$ and $!c < x >$ resp. in TIOSTS and STA).

		 $clock : \mathbb{N} \to \mathcal{C}$ is a mapping, where $\mathcal{C}$ is a set of clocks

Timing constraint relating occurrence instants of two different actions :
there is a (transmission) delay of at most of $0.1s$ between the reception
action (noted $c?y$ or $?c < y >$) and the sending action (noted $c!top(fifo)$ or $!c < top(fifo) >$).

a given value is consumed from the queue. So, we reset a new clock $clock[i_1]$ each time a given value is received (where i_1 is a natural number identifying the i_1^{th} reception occurrence and $clock$ is a mapping from natural numbers to a set of clocks $\mathcal{C}$, see tr_2). Since the i_2^{th} stored value is also the i_2^{th} transmitted value, when the emission occurs, the guard $clock[i_2] < 0.1$ must be satisfied (where i_2 identifies emissions, see tr_3).

In full generality, using STA formalism requires as many clocks as the number of occurrence instants to constrain, to choose suitable moments to reset the clocks cyclically whenever it is needed, and a non trivial transformation of the constraint. That latter may result in loss of essential traceability informations. In this context, the work [61] suggests to equip a variant of MSC [51] (scenario-based specification charts similar to sequence diagrams) with TA-like clock variables, clock constraints and clock reset actions. This resulting scenarios are therefore easily translated into TA. Unlike our approach, this work does not handle data in transit symbolically as first-order structures, they are rather enumerated. Besides, to the best of our knowledge, our approach is unique as we analyze (MARTE) constraints containing relations between time instants in scenarios. Our TIOSTS format is clearly better tailored to capture this kind of constraints as they are formulated in sequence diagrams. It is a natural extension of TIOSTS ([31]) and can be viewed simply as syntactic sugar for the STA ([87]) or TIOSTS ([5]) patterns needed to encode this kind of constraints.

Chapter 6

Operational semantics of UML MARTE Sequence Diagram

Contents

In this chapter we give the operational semantics of a subset of sequence diagrams with timing annotations as they were presented in natural language earlier in Chapter 2. The semantics is obtained by translating timed sequence diagrams into TIOSTS. In the following, we present the translation mechanism to obtain a TIOSTS from the textual definition of a sequence diagram (as described in Section 4.3). We translate each lifeline and message into a TIOSTS. Each lifeline TIOSTS is then completed with transitions which allow lifelines to be aware when a region of the sequence diagram is entered. Some regions does not concern the lifeline, however the the lifeline is notified when they are entered anyway. Those notifications are simply ignored in the completion transitions, hence the importance of knowing which regions concern each lifeline. Finally, a new TIOSTS is obtained by composing the automata of the lifelines and messages altogether. The resulting TIOSTS gives the semantics of the sequence diagram.

For the rest of this chapter, we suppose a sequence diagram signature $\Sigma_{sd} = (P \cup \{e\}, Var \cup \{i\}, Msg, Obs, Reg)$ is given. We write the translation mechanism from the textual definition of a sequence diagram to a TIOSTS as rules. The general form of a rule is the following:

$$\text{RULE NAME}\ \frac{expr = \ldots[o]\ldots[expr']\ldots \qquad [\mathbb{G}_{expr'}\ over\ \Sigma_{expr'}] \qquad [reg]}{\mathbb{G}_{expr}\ over\ \Sigma_{expr} \qquad [reg \cup \{o\}]}$$

Above the horizontal line of the deduction rule, we have first the textual expression ($expr$) of any pattern of the sequence diagram (a lifeline, a sub lifeline, a message, etc.) or the sequence diagram itself. Since the translation is inductive on the form of the sequence diagram definition, the translation of a sequence diagram pattern ($expr$) may be based on the translation of one of its sub patterns (see $expr'$ occurring in $expr$). Therefore the upper part of the a rule may introduce TIOSTS ($\mathbb{G}_{expr'}$) resulting from the translation of such a sub pattern. In addition, any region occurring in a sub pattern, more precisely in a sub lifeline pattern, is accumulated during the translation in the set reg: This is a kind of syntactic analysis of each lifeline definition in order to deduce which regions concern the lifeline among all the regions of the sequence diagram ($reg \subseteq Reg$). reg is used in the completion of transitions of the lifeline TIOSTS.

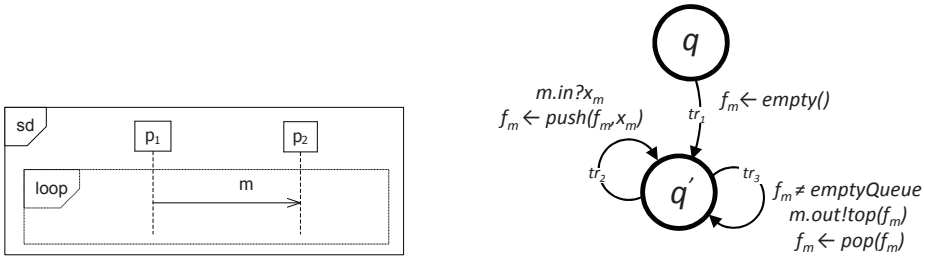
Below the horizontal line, we have first the new TIOSTS ($\mathbb{G}_{expr}$) constructed by the rule and its signature (Σ_{expr}). Then we have, in the case of the translation of lifeline expression, the new set of accumulated regions (it is increased with the new region o along with regions of reg obtained by antecedent rules applications).

As introduced in Definition 31, a sequence diagram is defined as couple of sets: a set of messages and a set of lifelines containing exactly one lifeline per port. We show a step-by-step translation going from messages then lifelines translation in order to obtain finally the full translation of the sequence diagram expression.

6.1 Translation of messages

A message in a sequence diagram represents the data exchanged between lifelines. It is depicted as an arrow from the sending lifeline to the receiving one. As mentioned in chapter 2, we consider asynchronous messages. That is to say that the sender of a message is not blocked until the message is received. Therefore, the translation of a message has to reflect this signification by decoupling emissions and receptions. Intuitively, we use a FIFO queue to hold data conveyed by a message until the target lifeline is ready to receive it.

There are two possible textual representations of a message msg: The form $(\_, true, m, \_)$ denotes a simple (not timed) message, and the form (t, ϕ_t, m, t') denotes a timed message (refer to Definition 28). The translation rules of the two forms are given respectively in Figures 6.1 and 6.2. Both rules are similar in the way they handle data in transit. In addition, the second rule, the one for the timed message, takes into consideration the timing features in the translation.



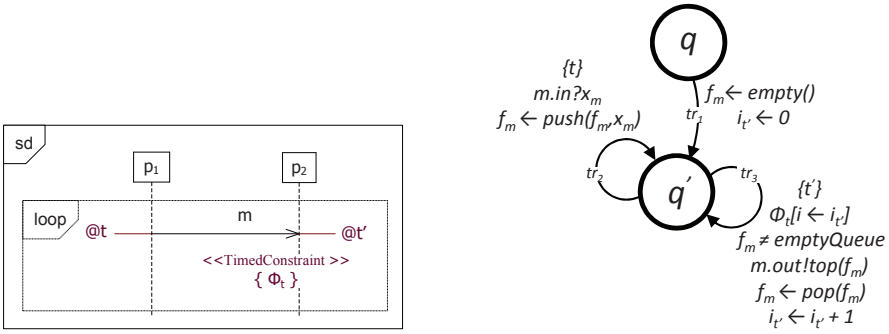
$$\text{MESSAGE}\ \frac{msg = (\_, true, m, \_)}{\begin{array}{l}\mathbb{G}_{msg} = (\{q, q'\}, q, \{tr_1, tr_2, tr_3\}) \\ over\ \Sigma_{msg} = (\{f_m, x_m\}, \{m.in, m.out\})\end{array}}$$

$$where \begin{cases} Read(\Sigma_{msg}) = \emptyset \\ Write(\Sigma_{msg}) = \{f_m, x_m\} \\ tr_1 = (q, \emptyset, true, true, \tau, id[f_m \leftarrow empty()], q') \\ tr_2 = (q', \emptyset, true, true, m.in?x_m, id[f_m \leftarrow push(f_m, x_m)], q') \\ tr_3 = (q', \emptyset, true, f_m \neq emptyQueue, m.out!top(f_m), id[f_m \leftarrow pop(f_m)], q') \end{cases}$$

Figure 6.1: Translation of a simple message

In the left upper parts of Figures 6.1 and 6.2, msg is represented graphically in a sequence diagram as being exchanged between two ports: p_1 and p_2. Thus $m \in Msg_{(p_1,p_2)}$. The rule in Figure 6.2 shows how to translate a timed message annotated with two time variables $t, t' \in Obs$, respectively capturing successive emission and reception instants of m. The transmission of m is constrained by the time guard $\phi_t \in \mathscr{T}_\Omega(V)$. E.g. ϕ_t may be of the form $t'[i] - t[i] < 0.1$ stating that the transmission of m is takes at most 0.1s.

[1]For any formula ϕ, $\phi[z \leftarrow y]$ is the formula where all occurrences of z are replaced by y



$$\text{TIMED MESSAGE}\ \frac{msg = (t, \phi_t, m, t')}{\begin{array}{l}\mathbb{G}_{msg} = (\{q, q'\}, q, \{tr_1, tr_2, tr_3\}) \\ over\ \Sigma_{msg} = (\{f_m, x_m, t, t', i_{t'}\} \cup Var(\phi_t) \setminus \{i\}, \{m.in, m.out\})\end{array}}$$

$$where\ ^{1}\begin{cases} Read(\Sigma_{msg}) = \{t, t'\} \cup Var(\phi_t) \setminus \{i\} \\ Write(\Sigma_{msg}) = \{f_m, x_m, i_{t'}\} \\ tr_1 = (q, \emptyset, true, true, \tau, id[f_m \leftarrow empty(), i_{t'} \leftarrow 1], q') \\ tr_2 = (q', \{t\}, true, true, m.in?x_m, id[f_m \leftarrow push(f_m, x_m)], q') \\ tr_3 = (q', \{t'\}, \phi_t[i \leftarrow i_{t'}], f_m \neq emptyQueue, m.out!top(f_m), id[f_m \leftarrow pop(f_m), i_{t'} \leftarrow i_{t'} + 1], q') \end{cases}$$

Figure 6.2: Translation of a timed message

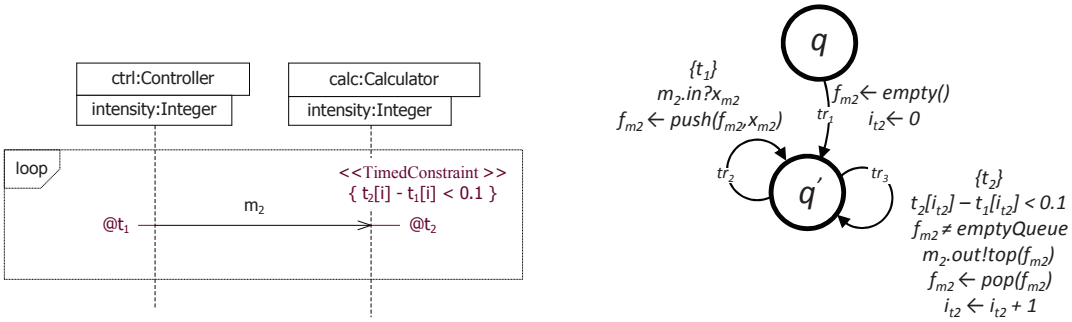
We map msg to a TIOSTS $\mathbb{G}_{msg}$ (represented graphically in Figures 6.1 and 6.2 in the right side, when a constraint is not shown, it means that it is $true$) over the signature Σ_{msg}. The TIOSTS $\mathbb{G}_{msg}$ communicates over channels of the form $m.in$ and $m.out$ respectively for reception and emission of values to be transmitted from a port lifeline to another. As we will see later in the chapter, the channel $m.in$ is shared with the emitter lifeline and the channel $m.out$ is shared with the receiver lifeline. Intuitively, for any message whose name is m, i.e. as it appears in the third field of the message definition, the convention $m.in$ is used to represent a channel to receive values of the sender lifeline while the convention $m.out$ is used to represent a channel to emit values to the receiver one. Now, the two data variables f_m and x_m are auxiliary variables introduced to handle the *transported* values by m: f_m is an unbounded FIFO queue used to store these values successively received on a variable x_m through the channel $m.in$. Given $m \in Msg_{(u,v)}$ such that $Msg_{(u,v)} \neq \emptyset$, note that the type of the data stored in f_m and x_m is of the same type as u and v (refer to Definition 27). Both variables x_m and f_m are read/write variables ($x_m, f_m \in Write(\Sigma_{msg})$).

When the message is timed (as in Figure 6.2), the message signature Σ_{msg} contains additionally the time variables t, t' (of type I^*) respectively associated with the receptions and the emissions. Times variables t and t' are read-only variables of Σ_{msg}. Recall that time variables are not controlled by the TIOSTS and their values are implicitly updated and thus are read-only variables. Besides t and t', some other time variables may occur in ϕ_t, those variables belong to Σ_{msg} as read-only variables ($Var(\phi_t) \subseteq Read(\Sigma_{msg})$, where $Var(\phi_t)$ returns all variables occurring in ϕ_t expression). Recall that instants in ϕ_t are of the form $t[i], t'[i]$, where i a distinct variable of the sequence diagram signature Σ_{sd} (refer to Definition 27) denotes a given i^th instant of an execution in a generic manner (it may also be used elsewhere in the sequence diagram for that purpose). For instance, $Var(t'[i] - t[i] < 0.1)$ returns the set of variables $\{t, t', i\}$. As we explain later in this section, we translate this use of i by introducing the time index $i_{t'}$ (of type $Integer$) in order to capture the last instant of the occurrence of the execution, specifically here the last emission instant: $i_{t'}$ refers to the last relevant location in t' and is incremented accordingly. i is replaced as the context requires by $i_{t'}$ since it is not used. For example, $t'[i] - t[i] < 0.1$ constraining the emission instants becomes $t'[i_{t'}] - t[i_{t'}] < 0.1$. $i_{t'}$ is a read/write variable of

Σ_{msg} ($i_{t'} \in Write(\Sigma_{msg})$) and we do not keep i in Σ_{msg}.

For both cases of either a simple or timed message, the built TIOSTS $\mathbb{G}_{msg}$ has two states q, q' and three transitions tr_1, tr_2, tr_3: The transition tr_1 from q to q' is a variables-initialization transition; the self−looping transition tr_2 on q' contains a value reception action; and the self−looping transition tr_3 on q' contains a value emission action. This way, the reception is decoupled from the emission. It remains to explain the storage mechanism in order to fully characterize asynchronous messages. Consider the transition tr_1. Initially f_m is empty. The queue f_m stores the values received (on a variable x_m) through the channel $m.in$ (see the communication action $m.in?x_m$ of the transition tr_2). Each time a value reaches its target port, it is interpreted as an emission on channel $m.out$ (transition tr_3). Values reach destination in the same order than they arrive. For this reason, at each step only one message can be emitted and it corresponds to $top(f_m)$.

We discuss now the additional features to consider in the case of timed messages. In this case, the time variables t, t' are respectively associated with the reception transition tr_2 and the emission transition tr_3 in Figure 6.2. The major difficulty is that if $t[i]$ and $t'[i]$ occurs in ϕ_t then $t[i]$ corresponds to the date the value arrived through $m.in$ while $t'[i]$ corresponds to the time the value is emitted through $m.out$. To identify those respective dates we introduce a time index $i_{t'}$ associated with time variable t', initially assigned by 0 and which is incremented in order to correspond to the size of t'. In the TIOSTS the variable t' (respectively t) is updated by adding a new instant in the transition corresponding to the emission (respectively reception). That is done implicitly by stating that t and t' belong to the sets of time variables associated with transitions corresponding respectively to the reception and the emission case. $t'[i_{t'}]$ thus denotes the instant of emission of the $i^{th}_{t'}$ value. Since the $i^{th}_{t'}$ emitted value is also the $i^{th}_{t'}$ received value, $t'[i_{t'}]$ is the date of reception of the value. The transition guarding the emission is thus ϕ_t where i is replaced by $i_{t'}$.



$$\text{MESSAGE}\ \frac{msg_2 = (t_1, t_2[i] - t_1[i] < 0.1, m_2, t_2)}{\mathbb{G}_{msg_2} = (\{q, q'\}, q, \{tr_1, tr_2, tr_3\}) \quad over\ \Sigma_{msg_2} = (\{f_{m_2}, x_{m_2}, t_2, t_1, i_{t_2}\}, \{m_2.in, m_2.out\})}$$

Figure 6.3: Translation of a message of Rain-sensing Wiper Control system (RWC)

Example 37 *Figure 6.3 depicts the translation of the message m_2 of the Rain-sensing Wiper Control system (RWC) as specified in Section 4.1. As discussed in the section, m_2 coveys cyclically (being in a loop operator region in the sequence diagram of RWC, refer to Figure 4.1) the rain intensity from the controller to the calculator within at most $0.1s$. The TIOSTS $\mathbb{G}_{msg_2}$ was obtained by applying the rule* TIMED MESSAGE *on the message definition msg_2 as $(t_1, t_2[i] - t_1[i] < 0.1, m_2, t_2)$ where $m_2 \in Msg_{(ctrl.intensity, calc.intensity)}$.*

Note that, the intensity values are of type Integer (type of ports $ctrl.intensity, calc.intensity$), so is x_{m_2} and f_{m_2} is a queue of Integers. The transition tr_1 initializes f_{m_2} and i_{t_2}. The

message TIOSTS encodes indeed a cyclic behavior where the two looping transitions tr_2, tr_3 models respectively the intensity values received through the channel $m_2.in$ on x_{m_2} and emissions of these values in the same order they arrive through $m_2.out$. This last transition is guarded by $t_2[i_{t_2}] - t_1[i_{t_2}] < 0.1$ obtained by substituting the i by i_{t_2} in $t_2[i] - t_1[i] < 0.1$ as in the message m_2 definition and this because in the context of the TIOSTS $\mathbb{G}_{msg_2}$ the index i_{t_2} captures the current emission instant stored in the last location in t_2 (i_{t_2} being incremented at each emission).

6.2 Translation of lifelines

In this section, we provide the rules to obtain the translation of a lifeline to a TIOSTS on the basis of possible lifeline expressions as described in Definition 30. Therefore, the translation is inductively defined on the form of the lifeline.

In the sequel, let p be a port in P and lf be a lifeline of p (in $Lf(p, \Sigma)$).

6.2.1 Empty lifeline

Let lf be $\varepsilon \in Lf(p, \Sigma)$. We map the empty lifeline to the TIOSTS $\mathbb{G}_\epsilon$. Here is the translation rule:

$$\text{EMPTY}\ \frac{\epsilon}{\begin{array}{l}\mathbb{G}_\epsilon = (\{q\}, q, \emptyset) \\ over\ \Sigma_\epsilon = (\{p, sched_p\}, \{start\}) \qquad \emptyset\end{array}}$$

where $Write(\Sigma_\epsilon) = \{p\}$

The set of variables is the set $\{p, sched_p\}$ where: p is the port associated with the lifeline ϵ ($\varepsilon \in Lf(p, \Sigma)$) and considered as a distinct variable used to store values arriving on (or emitted from) the lifeline as discussed in Section 2.1.2. p is built as a read/write variable in Σ_ϵ; $sched_p$ is an unbounded FIFO variable storing occurrences of region names each time a remote lifeline, not the one of p, execution goes into a region. It is a mechanism of scheduling locally, at the level of the lifeline, executions of operators defined as global in the sequence diagram. This will be detailed later in the sub section treating of combining operators translation. The set of channels of Σ_ϵ is the singleton $\{start\}$ where *start* is a channel used also in the scheduling mechanism though which exchanged regions names transit between lifelines (It is a shared channel between them).

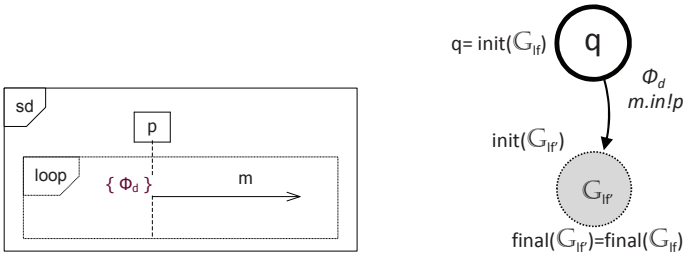
The symbol q denotes a new fresh state. The state q is the initial state of $\mathbb{G}_\epsilon$, denoted $init(\mathbb{G}_\epsilon)$. We also call it the *final state* of $\mathbb{G}_\epsilon$, denoted $final(\mathbb{G}_\epsilon)$.

6.2.2 Simple sequencing

In the special case of lf is of the form $(\mathbf{seq}, atom, lf')$ where $atom \in Atom(p, \Sigma)$ and $lf' \in Lf(p, \Sigma)$, let us note $\mathbb{G}_{lf'} = (Q, q_0, T)$ the translation of lf'. The translation of lf is of the form $\mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\})$ where q is a new fresh state symbol and tr is a transition depending of the *atom* form and whose target state is q_0. Note that $final(\mathbb{G}_{lf})$ is $final(\mathbb{G}_{lf'})$. In the following subsections we consider all possible forms of atoms as defined in Definition 29.

6.2.2.1 Emission/reception atom

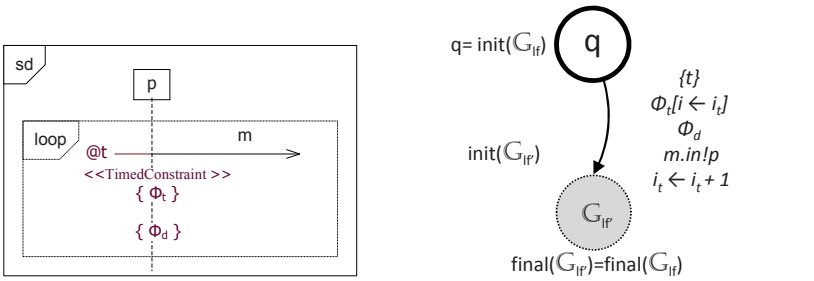
Consider the case where *atom* is of the form $(\_, true, \phi_d, m)$ or (t, ϕ_t, ϕ_d, m) respectively for simple/timed atoms where m is a message name (see Definition 29). Here two cases are possible: either m is of source p ($\exists v \in P \cup \{e\}$ such that $m \in Msg_{(p,v)}$) that denotes an emission of a value and the corresponding translation rules SEND and TIMED SEND are given in Figures 6.4–6.5; Or m is a message name of target p ($\exists u \in P \cup \{e\}$ such that $m \in Msg_{(u,p)}$) that denotes a reception of a value and the corresponding translation rules named RECEIVE and TIMED RECEIVE are given in Figures 6.6–6.7.



$$\text{SEND}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (\_, true, \phi_d, m), lf') \qquad m \in Msg_{(p,v)} \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup Var(\phi_d), C \cup \{m.in\}) \qquad reg \end{array}}$$

$$where\ ^{2} \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup Var(\phi_d) \setminus Write(\Sigma_{lf'}) \\ Write(\Sigma_{lf}) = Write(\Sigma'_{lf}) \\ tr = (q, \emptyset, true, \phi_d, m.in!p, id, q_0) \end{cases}$$

Figure 6.4: Translation of simple sending atom of a lifeline

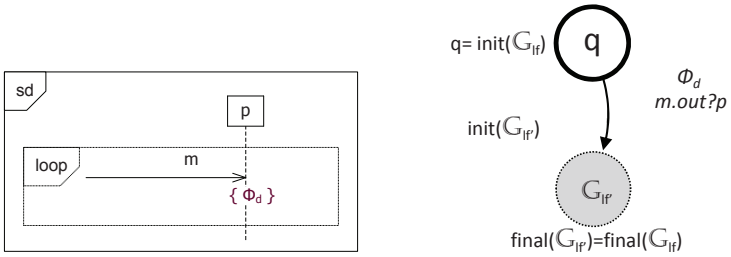


$$\text{TIMED SEND}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (t, \phi_t, \phi_d, m), lf') \qquad m \in Msg_{(p,v)} \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{t, i_t\} \cup Var(\phi_t) \cup Var(\phi_d) \setminus \{i\}, C \cup \{m.in\}) \qquad reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup \{t\} \cup Var(\phi_t) \cup Var(\phi_d) \setminus (Write(\Sigma_{lf'}) \cup \{i\}) \\ Write(\Sigma_{lf}) = Write(\Sigma'_{lf}) \cup \{i_t\} \\ tr = (q, \{t\}, \phi_t[i \leftarrow i_t], \phi_d, m.in!p, id[i_t \leftarrow i_t + 1], q_0) \end{cases}$$

Figure 6.5: Translation of timed sending atom of a lifeline

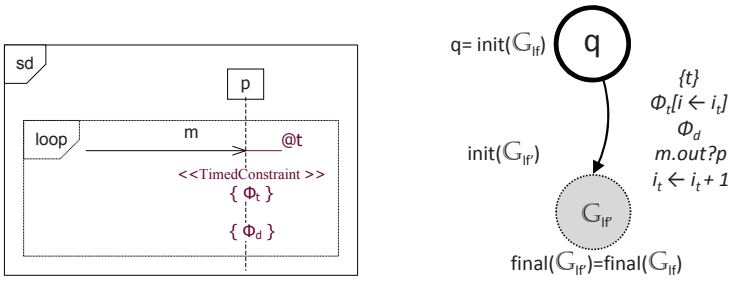
[2]Let $\phi \in Sen_\Omega(V)$. The *set of variables of* ϕ, denoted $Var(\phi)$ is the subset of V defined as follows: If ϕ is *true* or *false* then $Var(\phi) = \emptyset$; If ϕ is of the form $t_1 = t_2$ then $Var(\phi) = Var(t_1) \cup Var(t_2)$; If ϕ is of the form $\phi_1 \vee \phi_2$ or $\phi_1 \wedge \phi_2$ then $Var(\phi) = Var(\phi_1) \cup Var(\phi_2)$; If ϕ is of the form $\neg\psi$ then $Var(\phi) = Var(\psi)$.



$$\text{RECEIVE}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (\_, true, \phi_d, m), lf') \qquad m \in Msg_{(u,p)} \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup Var(\phi_d) \setminus \{i\}, C \cup \{m.out\}) \qquad reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup Var(\phi_d) \setminus Write(\Sigma_{lf'}) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf'}) \\ tr = (q, \emptyset, true, \phi_d, m.out?p, id, q_0) \end{cases}$$

Figure 6.6: Translation of simple reception atom of a lifeline



$$\text{TIMED RECEIVE}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (t, \phi_t, \phi_d, m), lf') \qquad m \in Msg_{(u,p)} \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{t, i_t\} \cup Var(\phi_t) \cup Var(\phi_d) \setminus \{i\}, C \cup \{m.out\}) \qquad reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup \{t\} \cup Var(\phi_t) \cup Var(\phi_d) \setminus (Write(\Sigma_{lf'}) \cup \{i\}) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf'}) \cup \{i_t\} \\ tr = (q, \{t\}, \phi_t[i \leftarrow i_t], \phi_d, m.out?p, id[i_t \leftarrow i_t + 1], q_0) \end{cases}$$

Figure 6.7: Translation of timed reception atom of a lifeline

Let us discuss the emission case. Recall that the emission of the value conveyed by the message m (depicted as an outgoing arrow from the lifeline of p in the sequence diagrams in Figures 6.4–6.5, see the upper left part) may happen only if ϕ_d and additionally, when the atom is timed, ϕ_t are satisfied. p is considered as a one buffer variable that contains the value to be sent. If this variable is not defined/assigned after an initialization for example, the same initial value is sent each time the system outputs.

We start by explaining the constituents of the built TIOSTS signature Σ_{lf}. Obviously based on the inductive form of the lifeline $lf = (\mathbf{seq}, atom, lf')$, Σ_{lf} contains all variables and channels of $\Sigma_{lf'}$. It contains as well the variables $Var(\phi_d)$ occurring in ϕ_d expression. In the same way as in the case where the message is timed, time variables associated with the timed atom of an emission are added to Σ_{lf}: which are the time variable associated with the emission t together with time variables of $Var(\phi_t)$. The partition read/write and read only of variables is similar to messages translation for variables t, i_t (the time index associated with t) and the timing variables

of $Var(\phi_t)$. Clearly all variables of $Write(\Sigma_{lf'})$ are also read/write of Σ_{lf}. We consider further the data variables occurring in ϕ_d ($Var(\phi_d)$). Some of them may have been already declared as read/write variables in $\Sigma_{lf'}$ (that is in $Write(\Sigma_{lf'})$) and so only the remaining variables, i.e. in $Var(\phi_d) \setminus Write(\Sigma_{lf'})$, are considered as read-only variables in Σ_{lf}.

A key point in the translation of an emission atom is the introduction of the channel $m.in$ in Σ_{lf}. Emitted values by the lifeline actually transit through $m.in$ which is a shared channel between $\mathbb{G}_{lf}$ and the TIOSTS of m in order to operate a synchronization between the two TIOSTS allowing the message to transmit these values. This is possible thanks to the naming convention $m.in$ used in either translation rules: on one hand, the rules MESSAGE and TIMED MESSAGE and on the other hand the rules SEND and TIMED SEND.

Now let us look at the augmented transition structure. As glimpsed at the beginning of the section, tr is built over a fresh state q as its source state and its target state is q_0 which is the initial state of $\mathbb{G}_{lf'}$. The execution of tr is guarded by ϕ_d as it is formulated in the sequence diagram. tr performs on the channel $m.in$ the communication action $m.in!p$ to output the value contained in p. Consider the case when the atom is timed: tr is associated with the time variable t; it is guarded by ϕ_t where all occurrences of i are replaced by the index variable i_t (same reasoning as in the timed message case); and the substitution of tr increments i_t and leaves all other variables unchanged.

Consider now the case of a reception atom ($atom = (t, \phi_t, \phi_d, m)$ or $(\_, true, \phi_d, m)$ with m is a message name of target p). The corresponding translation rules RECEIVE and TIMED RECEIVE are shown in Figures 6.7– 6.6. The signature Σ_{lf} differ from the one built by the previous emission rules only in the channel identity shared with TIOSTS of the message m. The channel is named conventionally rather $m.out$ as in the messages rules. The resulting transition tr here is very similar to the one constructed by the emission rules except that its communication action $m.out?p$ denotes an input on the channel $m.out$. The received value is stored in the variable p. As a result, the TIOSTS of the message m is supposed to synchronize with the TIOSTS of the emitter lifeline TIOSTS on $m.in$ in order to get data values. In a symmetrical manner, it should synchronize with the receiver lifeline TIOSTS on $m.out$ in order to transmit those values.

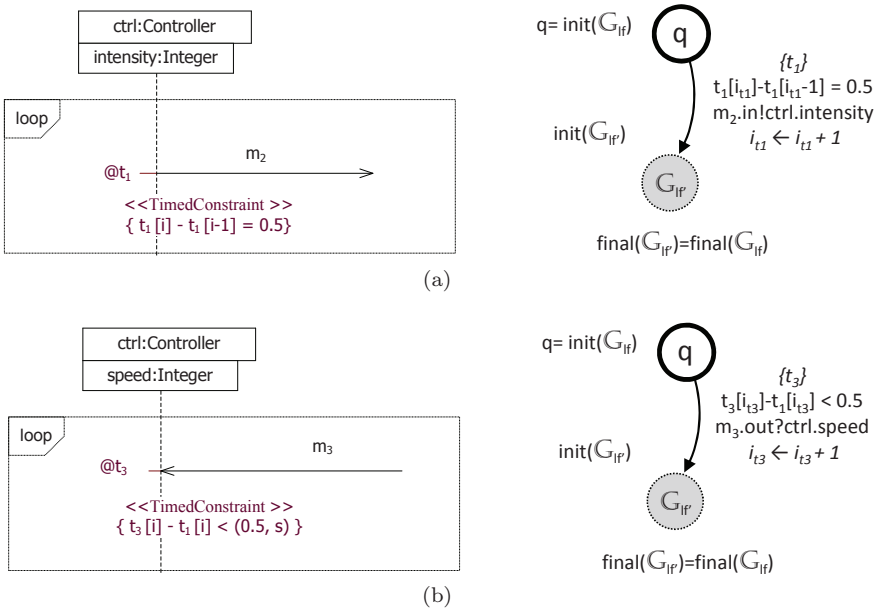


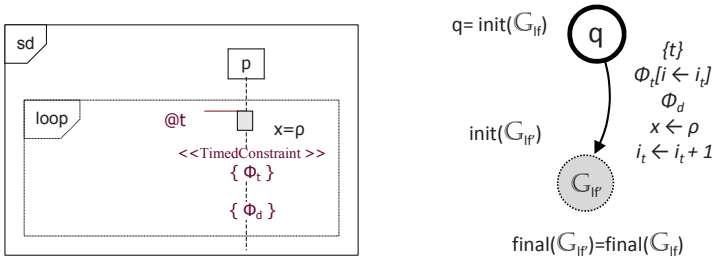
Figure 6.8: Examples of translation of respectively a sending and reception atoms

Example 38 *Consider again the sequence diagram of RWC system in Figure 4.1. The Figure 6.8a*

illustrates the translation of the sub lifeline of the port ctrl.intensity starting at the atom whose associated instant is t_1 (lf' is the empty lifeline in this case). The rule RECEIVE *is applied such that the atom atom is equal to $(t_1, t_1[i] - t_1[i-1] = 0.5, true, m_2)$ where $m_2 \in Msg_{(ctrl.intensity, calc.intensity)}$. The generated transition constrains the sending of the intensity (see the output action $m_2.in?ctrl.intensity$) to be performed every $0.5s$: the transition guard $t_1, t_1[i_{t_1}] - t_1[i_{t_1} - 1] = 0.5$ is obtained replacing i by i_{t_1}. i_{t_1} is a fresh index incremented exclusively by the transition ($i_{t_1} \leftarrow i_{t_1} + 1$) oin order to capture the current time instant of the emission ($t_1[i_{t_1}]$, t_1 quantifies all time elapsed between successive emissions). The Figure 6.8b illustrates another example of an atom translation: the translation of the sub lifeline of ctrl.speed starting at the atom $(t_3, t_3[i] - t_1[i] < 0.5, true, m_3)$ whose associated instant is t_3 (lf' starts with a alt operator in Figure 4.1).*

6.2.2.2 Assignement atom

Let *atom* be of the form $(t, \phi_t, \phi_d, x = \varrho)$ or $(\_, true, \phi_d, x = \varrho)$. The corresponding translation rules named respectively TIMED ASSIGN and ASSIGN are defined in Figure 6.9.



$$\text{TIMED ASSIGN}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (t, \phi_t, \phi_d, x = \varrho), lf') \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{x, t, i_t\} \cup Var(\phi_t) \cup Var(\phi_d) \cup Var(\varrho) \setminus \{i\}, C \cup \{m.out\}) \qquad reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup \{t\} \cup Var(\phi_t) \cup Var(\phi_d) \cup Var(\varrho) \setminus (Write(\Sigma_{lf'}) \cup \{x, i\}) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf'}) \cup \{x, i_t\} \\ tr = (q, \{t\}, \phi_t[i \leftarrow i_t], \phi_d, \tau, id[i_t \leftarrow i_t + 1, x \leftarrow \varrho], q_0) \end{cases}$$

$$\text{ASSIGN}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (\_, true, \phi_d, x = \varrho), lf') \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{x\} \cup Var(\phi_d) \cup Var(\varrho), C \cup \{m.out\}) \qquad reg \end{array}}$$

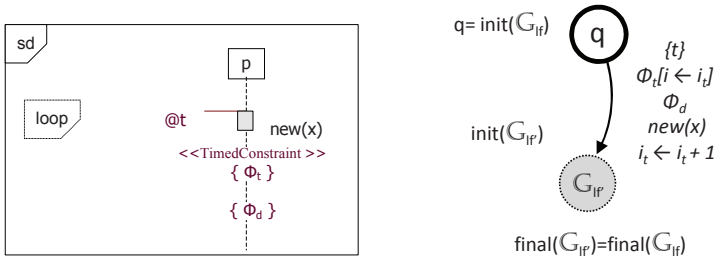
$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup Var(\phi_d) \cup Var(\varrho) \setminus (Write(\Sigma_{lf'}) \cup \{x\}) \\ Write(\Sigma_{lf}) = \{x\} \cup Write(\Sigma_{lf'}) \\ tr = (q, \emptyset, true, \phi_d, \tau, id[x \leftarrow \varrho], q_0) \end{cases}$$

Figure 6.9: Translation of an assignment atom of a lifeline

Note that the partition read/write and read-only variables in Σ_{lf} takes into account that some data variables occurring in ϱ expression ($Var(\varrho)$) may be already read/write variables in $\Sigma_{lf'}$ and of course that the variable x which is defined by the atom is necessarily a read/write variable. Obviously, in the constructed transition tr, there is no communication action (then τ is introduced). The substitution of the transition tr assigns the term ϱ to x, all other variables are not modified.

6.2.2.3 Underspecification atom

Let *atom* be of the form $(t, \phi_t, \phi_d, new(x))$ or $(\_, true, \phi_d, new(x))$. The corresponding translation rules named respectively TIMED UNDERSPEC and UNDERSPEC are defined in Figure 6.10.



$$\text{TIMED UNDERSPEC}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (t, \phi_t, \phi_d, new(x)), lf') \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{x, t, i_t\} \cup Var(\phi_t) \cup Var(\phi_d) \cup Var(\varrho) \setminus \{i\}, C \cup \{m.out\}) \\ reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup \{t\} \cup Var(\phi_t) \cup Var(\phi_d) \setminus (Write(\Sigma_{lf'}) \cup \{x, i\}) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf'}) \cup \{x, i_t\} \\ tr = (q, \{t\}, \phi_t[i \leftarrow i_t], \phi_d, new(x), id[i_t \leftarrow i_t + 1], q_0) \end{cases}$$

$$\text{UNDERSPEC}\ \frac{\begin{array}{l} lf = (\mathbf{seq}, (\_, true, \phi_d, new(x)), lf') \\ \mathbb{G}_{lf'} = (Q, q_0, T)\ over\ \Sigma_{lf'} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q \cup \{q\}, q, T \cup \{tr\}) \\ over\ \Sigma_{lf} = (A \cup \{x\} \cup Var(\phi_d) \cup Var(\varrho), C \cup \{m.out\}) \qquad reg \end{array}}$$

$$where \begin{cases} Read(\Sigma_{lf}) = Read(\Sigma'_{lf}) \cup Var(\phi_d) \setminus (Write(\Sigma'_{lf}) \cup \{x\}) \\ Write(\Sigma_{lf}) = \{x\} \cup Write(\Sigma_{lf'}) \\ tr = (q, \emptyset, true, \phi_d, new(x), id, q_0) \end{cases}$$

Figure 6.10: Translation of a lifeline underspecification atom

Now we construct the transition tr for the underspecification atom that is recognizable by the symbol new. Recall that $new(x)$ assigns a random value to the variable x (and it is still possible that this variable is assigned with its previous value). Therefore as in the case of an assignment atom, x is considered as a read/write variable in Σ_{lf}. The action of tr is naturally $new(x)$ which was intended for that use in our TIOSTS formalism.

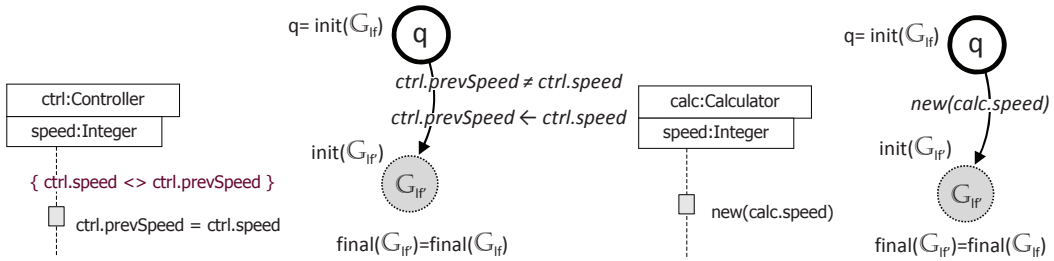


Figure 6.11: Examples of translation of respectively an assignment and underspecification atoms

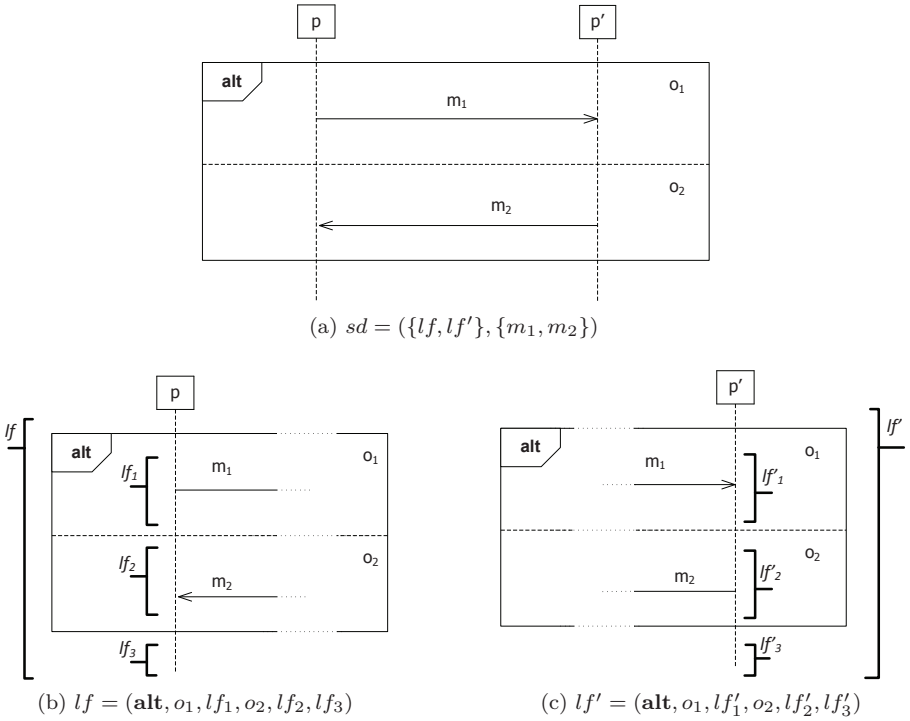
Example 39 *Figure 6.11 gives some examples of translation of atoms taken from the sequence diagram of RWC system in Figure 4.1.*

6.2.3 Combination operator

Recall that we are discussing the translation rules of all possible forms of a lifeline as defined in Definition 30: starting with an empty lifeline; continuing with all possible forms which contain atoms; and now in this section finishing with the remaining forms, that is, those which contain combining operators.

6.2.3.1 alt operator

From a local point of view, the one of the lifeline, the alt operator defines a non deterministic choice between (two) lifeline sub patterns. The translation of the *alt* operator is more subtle than the translation of the other operators. The difficulty arises because it may cover lifelines of other ports as well, each of which operates at its own rate while the translation must be able to guarantee consistent global choices.



(a) $sd = (\{lf, lf'\}, \{m_1, m_2\})$

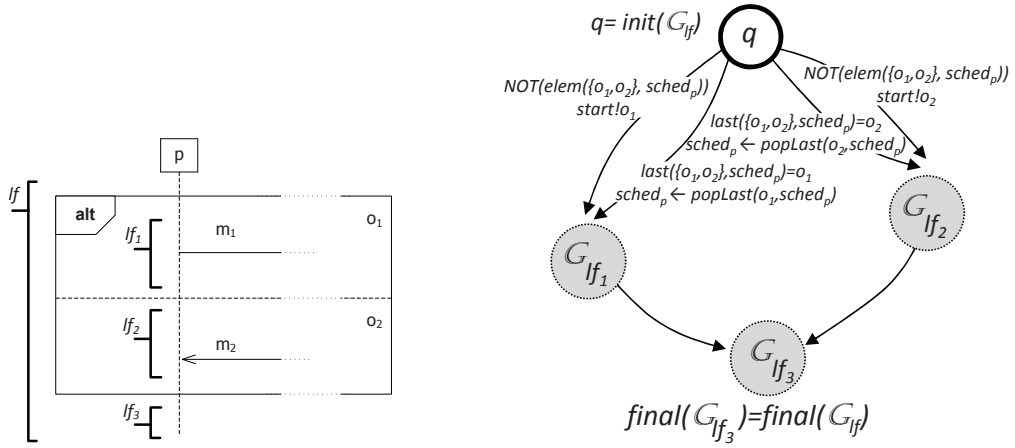
(b) $lf = (\mathbf{alt}, o_1, lf_1, o_2, lf_2, lf_3)$

(c) $lf' = (\mathbf{alt}, o_1, lf'_1, o_2, lf'_2, lf'_3)$

Figure 6.12

Let lf be of the form $(\mathbf{alt}, o_1, lf_1, o_2, lf_2, lf_3)$ where $lf_1, lf_2, lf_3 \in Lf(p, \Sigma)$. In order to discuss the translation of lf, we need a global view of some interactions structured by an alt and involving besides the port p (recall that $lf \in Lf(p, \Sigma)$), another port. A generic example is illustrated in the sequence diagram of Figure 6.12a where is depicted further the lifeline lf' of another port p' ($lf' \in Lf(p', \Sigma)$). lf' is of the form $(\mathbf{alt}, o_1, lf'_1, o_2, lf'_2, lf'_3)$ where $lf'_1, lf'_2, lf'_3 \in Lf_{p'}(\Sigma)$. Figures 6.12b–6.12c show how the sequence diagram is decomposed by the syntax that we have introduced in Chapter 4 in order to encode the alt operator at the level of the lifelines lf and lf' respective expressions. Note the role of the region names in preserving informations about each lifeline sub behavior location in the diagram. For instance, we know that the sub lifeline lf_1 (respectively lf_2) of p is located in the region o_1 (respectively lf_2) (symmetric information is available for lf'_1 and lf'_2). Recall that the semantics of the alt operator requires that exactly one region will be executed either o_1 or o_2. Consequently, the translation rule of the alt operator has to make sure that if lf_1 of p is executed then, on p' side, lf'_1 is executed and vice versa (this must be true for lf_2 and lf'_2 as well).

The translation rule ALT for the *alt* operator is given in Figure 6.13. Remember that the lifeline lf of the port p is our "reference" lifeline, the one for which the rule generates a TIOSTS $\mathbb{G}_{lf}$.



(a)

$$\text{ALT}\ \frac{\begin{array}{l} lf = (\mathbf{alt}, o_1, lf_1, o_2, lf_2, lf_3) \\ \mathbb{G}_{lf_1} = (Q_1, q_0^1, T_1)\ over\ \Sigma_{lf_1} = (A_1, C_1) \\ \mathbb{G}_{lf_2} = (Q_2, q_0^2, T_2)\ over\ \Sigma_{lf_2} = (A_2, C_2) \\ \mathbb{G}_{lf_3} = (Q, q_0, T)\ over\ \Sigma_{lf_3} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q_1 \cup Q_2 \cup Q \cup \{q\}, q, T_1 \cup T_2 \cup T \cup \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6\}) \\ over\ \Sigma_{lf} = (A_1 \cup A_2 \cup A, C_1 \cup C_2 \cup C) \qquad reg \cup \{o_1, o_2\} \end{array}}$$

where

$$\begin{cases} Read(\Sigma_{lf}) = Read(\Sigma_{lf_1}) \cup Read(\Sigma_{lf_2}) \cup Read(\Sigma_{lf_3}) \setminus (Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2}) \cup Write(\Sigma_{lf_3})) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2}) \cup Write(\Sigma_{lf_3}) \\ tr_1 = (q, \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_p)), start!o_1, id, q_0^1) \\ tr_2 = (q, \emptyset, true, last(\{o_1, o_2\}, sched_p) = o_1, \tau, id[sched_p \leftarrow popLast(o_1, sched_p)], q_0^1) \\ tr_3 = (q, \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_p)), start!o_2, id, q_0^2) \\ tr_4 = (q, \emptyset, true, last(\{o_1, o_2\}, sched_p) = o_2, \tau, id[sched_p \leftarrow popLast(o_2, sched_p)], q_0^2) \\ tr_5 = (final(\mathbb{G}_{lf_1}), \emptyset, true, true, \tau, id, q_0) \\ tr_6 = (final(\mathbb{G}_{lf_2}), \emptyset, true, true, \tau, id, q_0) \end{cases}$$

Figure 6.13: Translation of the alt operator

In the rule ALT, we note $\mathbb{G}_{lf_1}$, $\mathbb{G}_{lf_2}$ and $\mathbb{G}_{lf_3}$ the TIOSTS respectively associated with lf_1, lf_2 and lf_3. The TIOSTS $\mathbb{G}_{lf}$ associated with lf is built upon a new fresh initial state (denoted q). It contains all transitions of $\mathbb{G}_{lf_1}$, $\mathbb{G}_{lf_2}$ and $\mathbb{G}_{lf_3}$. Since the *alt* operator permits to define choices of executions between different sub lifelines (lf_1 and lf_2), it introduces transitions to reflect those choices (the four transitions tr_1, tr_2, tr_3, tr_4 of source q also illustrated in Figure 6.13a).

Let us discuss the two transitions tr_1, tr_2 of source q situated on the top right part of Figure 6.13a (the transitions tr_3, tr_4 are the ones on the top left of the figure and represent a symmetric case). Obviously when two lifelines (here lf and lf') share a common region, the fact that an execution associated with one of the lifelines went through that region has an impact on the execution of the other: if $\mathbb{G}_{lf}$ and $\mathbb{G}_{lf'}$ share regions o_1 and o_2 and some execution of $\mathbb{G}_{lf}$ went successively into region o_1 and o_2, then the corresponding execution of $\mathbb{G}_{lf'}$ should also go into o_1 and o_2. Therefore the choice to be made to follow an execution of lf_1 or lf_2 is conditioned by a decision made by some other lifelines sharing o_1 or o_2 with lf. $sched_p$ is an unbounded FIFO variable storing occurrences of region names o_i ($i \in \{1, 2\}$) each time such a remote lifeline execution go into o_i. If neither o_1 nor o_2 occurs in $sched_p$ then the execution may go in region o_2 (transition tr_1 illustrated in the upper right part of Figure 6.13a) and the value o_2 is sent through the channel *start*, which is shared by all lifeline translations, in order to inform the other

concerned remote lifelines of that execution choice. In the transition tr_2 of source q just below in Figure 6.13a, o_2 occurs in $sched_p$ and if o_1 also occurs in $sched_p$ then the last occurrence of o_2 is after the last occurrence of o_1 (thus an execution through o_2 was required *before* executions through o_1 since $sched_p$ has a FIFO structure). In that case the execution has to go in o_2 and the last occurrence of o_2 in $sched_p$ is removed.

Note that additional operations were defined on the FIFO queues in order to encode the complex behavior explained before: recall that S is the set of types and F is the set of operations. The operation $elem \in F$ with profile $set(s).queue(s) \rightarrow bool$ tells whether at least an element of type $s \in S$ from the input set is in the queue; The operation *last* with profile $set(s).queue(s) \rightarrow s$ returns the element of the input set which was the first to be enqueued in the queue (without removal), if none of them exists in the queue then the operation fails; The operation *popLast* with profile $s.queue(s) \rightarrow queue(s)$ removes the first occurrence,i.e. enqueued first, of the input element from the queue if it does exist.

Lastly, the two transitions tr_5, tr_6, that are in the lower part of the figure, whose targets are the initial state of $\mathbb{G}_{lf_3}$ are non guarded transitions with τ actions and that do not modify variables assignments; they are added to connect executions of $\mathbb{G}_{lf_1}$ and $\mathbb{G}_{lf_2}$ to those of $\mathbb{G}_{lf_3}$.

Example 40 *In Figure 6.14a, the lifeline of port p' shares regions o_1, o_2 (resp. o_3, o_4) with the one of port p (resp. p''). A possible scenario of execution is illustrated in Figure 6.14b: we show how the content of the FIFO queue $sched_{p'}$ evolves. We have that the lifelines of p'', p went successively in regions o_3, o_2 before any execution is started on the port p' side. At this level, the queue $sched_{p'}$ contains in order of arrival the regions o_3 then o_2. However, the lifeline of p' has to execute the behavior in region o_2 before the one in o_3 (as specified in Figure 6.14a). That is why, the region token o_2 is consumed first (The effect of popLast operation call is the removal of o_2 from the queue).*

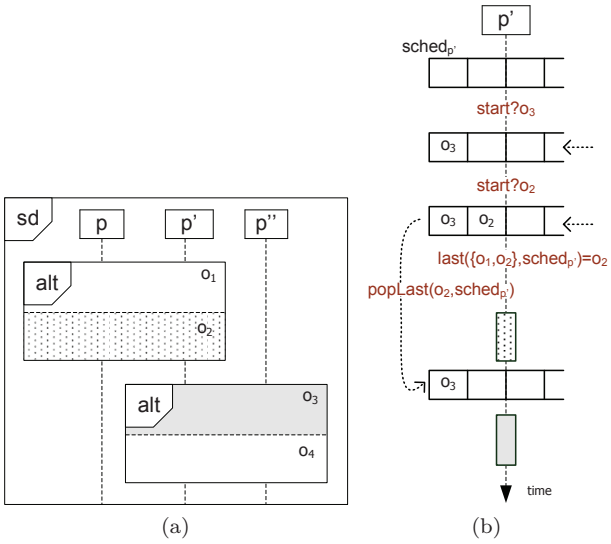
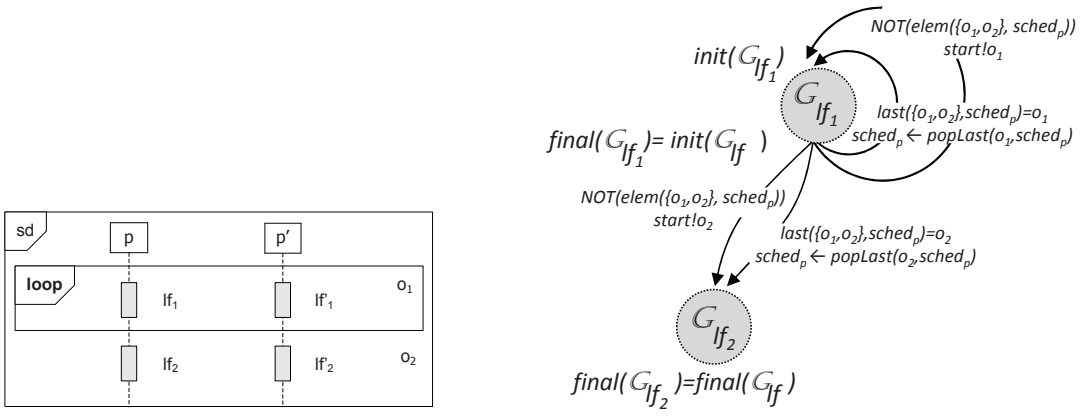


Figure 6.14: Synchronization mechanism of operator regions

Example 41 *The TIOSTS in Figure 6.13a illustrates the translation of the sub lifeline of ctrl.speed (p = ctrl.speed) starting at the most external alt operator, see the sequence diagram in Figure 4.1.*

6.2.3.2 loop operator

Let lf be of the form $(\mathbf{loop}, o, lf_1, lf_2)$. The translation is given in Figure 6.15 by the rule LOOP.



(a)

$$\text{LOOP}\ \frac{\begin{array}{l} lf = (\mathbf{loop}, o, lf_1, lf_2) \\ \mathbb{G}_{lf_1} = (Q_1, q_0^1, T_1)\ over\ \Sigma_{lf_1} = (A_1, C_1) \\ \mathbb{G}_{lf_2} = (Q, q_0, T)\ over\ \Sigma_{lf_2} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q_1 \cup Q, final(\mathbb{G}_{lf_1}), T_1 \cup T \cup \{tr_1, tr_2, tr_3, tr_4\}) \\ over\ \Sigma_{lf} = (A_1 \cup A, C_1 \cup C) \qquad reg \cup \{o_1, o_2\} \end{array}}$$

where

$$\begin{cases} Read(\Sigma_{lf}) = Read(\Sigma_{lf_1}) \cup Read(\Sigma_{lf_2}) \setminus (Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2})) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2}) \\ tr_1 = (final(\mathbb{G}_{lf_1}), \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_p)), start!o_1, id, q_0^1) \\ tr_2 = (final(\mathbb{G}_{lf_1}), \emptyset, true, last(\{o_1, o_2\}, sched_p) = o_1, \tau, id[sched_p \leftarrow popLast(o_1, sched_p)], q_0^1) \\ tr_3 = (final(\mathbb{G}_{lf_1}), \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_p)), start!o_2, id, q_0) \\ tr_4 = (final(\mathbb{G}_{lf_1}), \emptyset, true, last(\{o_1, o_2\}, sched_p) = o_2, \tau, id[sched_p \leftarrow popLast(o_2, sched_p)], q_0) \end{cases}$$

Figure 6.15: Translation of the loop operator

Translation rule for the *loop* operator is based on the same kind of synchronization mechanisms than the one used for *alt*.

Translation of the *loop* operator adds transitions tr_1, tr_2 to reflect cyclic executions (the two transitions on the upper right part of Figure 6.15a). The cyclic behavior consists in repetitive executions of $\mathbb{G}_{lf_1}$ the translation of the sub lifeline lf_1 which is in the region o_1 delimited by the *loop* operator frame in the diagram. The number of iteration is not predefined and may be finite or infinite. The challenge when the number of iteration is finite and not known in advance, is to force the execution associated with any lifeline to iterate that same number of iterations on $\mathbb{G}_{lf_1}$ then continue with the execution of lf_2 without waiting for the others to leave o_1. The synchronization mechanism makes each lifeline execution that initiates a new iteration, informs the others by sending the region token o_1 on the channel *start* (transition tr_1). The execution necessarily enters the region o_1 if the corresponding token is available in the queue $sched_p$ (transition tr_2).

Two transitions tr_3, tr_4 are introduced to force the execution to leave the region o_1 of the cyclic behavior (the two transitions on the lower left part of Figure 6.15a). In fact, a lifeline execution may choose non deterministically to leave the region o_1 and execute $\mathbb{G}_{lf_2}$ so it informs the others by sending on channel *start* an artifact region o_2, a fresh region symbol introduced for that purpose (transition tr_3). Of course, the lifeline execution may have been informed before to leave that region by another lifeline execution (captured by the transition tr_4).

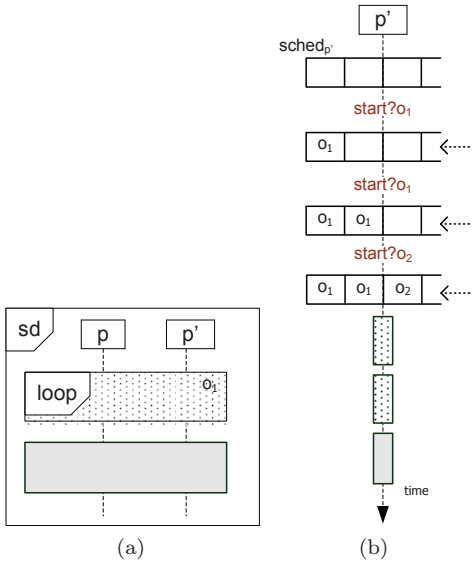


Figure 6.16: Synchronization mechanism of operator regions

Example 42 *In Figure 6.16a, the lifeline of port p' shares regions o_1 of the loop with the one of port p. A possible scenario of execution is illustrated in Figure 6.16b. The lifeline of p went twice in the region o_1 then left that region before any execution is started on the port p' side. The queue $sched_{p'}$ contains in order of arrival two occurrences of the region o_1 then o_2, o_2 is an artifact region used as token to notify the others when leaving the cyclic behavior. Thus, the lifeline of p' has to execute only twice the behavior in region o_1 before leaving the loop and continue with the following behavior.*

6.2.3.3 strict operator

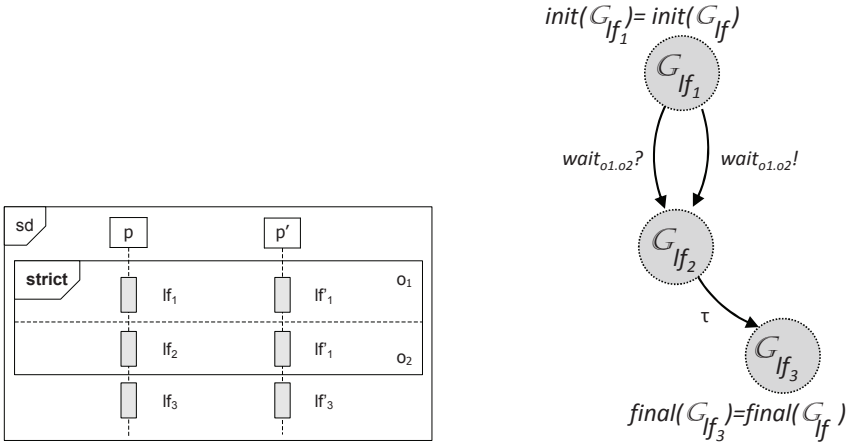
Let lf be of the form (**strict**, $o_1, lf_1, o_2, lf_2, lf_3$). Consider the translation rule STRICT for the strict operator in Figure 6.17.

The rule STRICT is illustrated in Figure 6.17.

The key idea here is to force the execution of the first sub lifeline translation $\mathbb{G}_{lf_1}$ to wait for others before resuming with the execution of the second sub lifeline translation $\mathbb{G}_{lf_2}$. Consequently, this rule has to make all executions of the first sub lifelines lf_1 of all ports in region o_1 interleave and then synchronize before continuing with the executions of the second sub lifelines lf_2 in the region o_2. Since the treatment of the *strict* operator is performed simultaneously, the lifeline does not exchange region values, so no need to increase the set reg. For this aim, we introduce the channel $wait_{o_1.o_2}$ on which all the lifelines executions have to synchronize. This name is unique. Two transitions tr_1, tr_2 are constructed to encode the synchronized execution: we have either the translation of lifeline $\mathbb{G}_{lf}$ is the one to output on the channel $wait_{o_1.o_2}$ (transition tr_1) or it is among some other lifelines translation $\mathbb{G}_{lf'}$ to input on that channel (transition tr_2). Finally, the transition tr_3 connects the executions of $\mathbb{G}_{lf_2}$ to those of $\mathbb{G}_{lf_3}$.

6.2.4 Completion operations

Once all lifelines lf of the sequence diagram to be translated have been associated with a TIOSTS $\mathbb{G}_{lf}$, two last operations are performed on it. First the set of transitions of $\mathbb{G}_{lf}$ is enriched by an



(a)

$$\text{STRICT}\ \frac{\begin{array}{l} lf = (\textbf{strict}, o_1, lf_1, o_2, lf_2, lf_3) \\ \mathbb{G}_{lf_1} = (Q_1, q_0^1, T_1)\ over\ \Sigma_{lf_1} = (A_1, C_1) \\ \mathbb{G}_{lf_2} = (Q_2, q_0^2, T_2)\ over\ \Sigma_{lf_2} = (A_2, C_2) \\ \mathbb{G}_{lf_3} = (Q, q_0, T)\ over\ \Sigma_{lf_3} = (A, C) \qquad reg \end{array}}{\begin{array}{l} \mathbb{G}_{lf} = (Q_1 \cup Q_2 \cup Q, q_0^1, T_1 \cup T_2 \cup T \cup \{tr_1, tr_2, tr_3\}) \\ over\ \Sigma_{lf} = (A_1 \cup A_2 \cup A, C_1 \cup C_2 \cup C \cup \{wait_{o_1.o_2}\}) \qquad reg \end{array}}$$

where

$$\begin{cases} Read(\Sigma_{lf}) = Read(\Sigma_{lf_1}) \cup Read(\Sigma_{lf_2}) \cup Read(\Sigma_{lf_3}) \setminus (Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2}) \cup Write(\Sigma_{lf_3})) \\ Write(\Sigma_{lf}) = Write(\Sigma_{lf_1}) \cup Write(\Sigma_{lf_2}) \cup Write(\Sigma_{lf_3}) \\ tr_1 = (final(\mathbb{G}_{lf_1}), \emptyset, true, true, wait_{o_1.o_2}!, id, q_0^2) \\ tr_2 = (final(\mathbb{G}_{lf_1}), \emptyset, true, true, wait_{o_1.o_2}?, id, q_0^2) \\ tr_3 = (final(\mathbb{G}_{lf_2}), \emptyset, true, true, \tau, id, q_0) \end{cases}$$

Figure 6.17: Translation of the loop operator

initialization transition (see Figure 6.18 for illustration) which assigns the value 0 to all the time indexes associated with the time variables of the lifeline (in Figure 6.18, $i_{t_i} \leftarrow 0$ for all $i \leq n$ where $\{t_1, \cdots, t_n\}$ is the set of all time variables of the lifeline). The FIFO $sched_p$ is initialized to the empty FIFO and a variable $myReg_p$ is assigned to the set of all regions occurring in the definition of lf. This information was obtained by the static analysis of lf definition where regions were accumulated in reg during successive applications of translation rules.

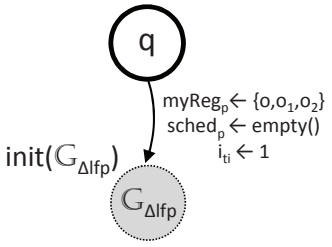


Figure 6.18: Initialization Completion.

The second operation consist in adding for all states of the TIOSTS a looping transition whose purpose is to store in $sched_p$ all the region crossing decision made by lifelines sharing regions with lf. As illustrated in Figure 6.19 each time a region name occurring in $myReg_p$ is received on the channel *start*, it is stored in $sched_p$.

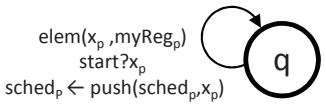


Figure 6.19: Input Completion.

We note $\overline{\mathbb{G}_{lf}}$ the TIOSTS resulting of the application of the two operations described above on $\mathbb{G}_{lf}$.

6.3 Full translation of a sequence diagram

Let sd be a sequence diagram $(\{lf_{p_1}, \ldots, lf_{p_n}\}, \{msg_1, \ldots, msg_l\})$. The translation of sd is a TIOSTS $\mathbb{G}_{sd}$ defined as $\overline{\mathbb{G}_{lf_{p_1}}} || \cdots || \overline{\mathbb{G}_{lf_{p_n}}} || \mathbb{G}_{msg_1} || \cdots || \mathbb{G}_{msg_l}$ (the chosen order of the composition has no impact since the operator $||$ of Definition 35 is commutative).

Example 43 *Consider the lifeline $lf_{ctrl.intensity}$ associated with the port ctrl.intensity in Figure 4.1:*

$$(\textbf{\textit{loop}}, o, lf^1_{ctrl.intensity}, lf^0_{ctrl.intensity})$$

where $lf^0_{ctrl.intensity} = \epsilon$
$lf^1_{ctrl.intensity} = (\textbf{\textit{alt}}, o_1, lf^2_{ctrl.intensity}, o_2, lf^3_{ctrl.intensity}, lf^4_{ctrl.intensity})$
$lf^3_{ctrl.intensity} = \epsilon$
$lf^4_{ctrl.intensity} = \epsilon$
$lf^2_{ctrl.intensity} = (\textbf{\textit{seq}}, (\_, true, true, m_1), lf^5_{ctrl.intensity})$
$lf^5_{ctrl.intensity} = (\textbf{\textit{seq}}, (t_1, t_1[i] - t_1[i-1] = 0.5, true, m_2), lf^6_{ctrl.intensity})$
$lf^6_{ctrl.intensity} = \epsilon$

We apply inductively the translation rules until we obtain at the end (check further for the step 9) the translation of the lifeline $lf_{ctrl.intensity}$. A sub lifeline can be translated only if any sub lifeline which occurs in its expression has been already translated. For example, in order to apply the rule SEND *in the 2^{nd} step on the sub lifeline $lf^5_{ctrl.intensity}$, the translation of the sub lifeline $lf^6_{ctrl.intensity}$ was computed in the 1^{st} step by the applying the rule* EMPTY.

1.

$$\text{EMPTY}\ \frac{lf^6_{ctrl.intensity} = \epsilon}{\begin{array}{l}\mathbb{G}_{lf^6_{ctrl.intensity}} = (\{q_0\}, q_0, \emptyset) \\ over\ \Sigma_{lf^6_{ctrl.intensity}} = (\emptyset, \emptyset) \qquad \emptyset\end{array}}$$

2.

$$\text{SEND}\ \frac{\begin{array}{l}lf^5_{ctrl.intensity} = (\textbf{\textit{seq}}, (t_1, t_1[i] - t_[i-1] = 0.5, true, m_2.in!ctrl.intensity), lf^6_{ctrl.intensity}) \\ \mathbb{G}_{lf^6_{ctrl.intensity}}\ over\ \Sigma_{lf^6_{ctrl.intensity}} \qquad \emptyset\end{array}}{\begin{array}{l}\mathbb{G}_{lf^5_{ctrl.intensity}} = (\{q_0, q_1\}, q_1, \{tr_0\}) \\ over\ \Sigma_{lf^5_{ctrl.intensity}} = (\{t_1, ctrl.intensity, i_{t_1}\}, \{m_2.in\}) \qquad \emptyset\end{array}}$$

where $tr_0 = (q_1, \{t_1\}, t_1[i_{t_1}] - t_[i_{t_1} - 1] = 0.5, true, m_2.in!ctrl.intensity, id[i_{t_1} \leftarrow i_{t_1} + 1], q_0)$

3.

$$\text{RECEIVE}\ \frac{\begin{array}{l} lf^2_{ctrl.intensity} = (\textbf{\textit{seq}}, (\_, true, true, m_1.out?ctrl.intensity), lf^5_{ctrl.intensity}) \\ \mathbb{G}_{lf^5_{ctrl.intensity}}\ over\ \Sigma_{lf^5_{ctrl.intensity}} \qquad \emptyset \end{array}}{\begin{array}{l} \mathbb{G}_{lf^2_{ctrl.intensity}} = (\{q_0, q_1, q_2\}, q_2, \{tr_0, tr_1\}) \\ over\ \Sigma_{lf^2_{ctrl.intensity}} = (\{t_1, ctrl.intensity, i_{t_1}\}, \{m_2.in, m_1.out\}) \qquad \emptyset \end{array}}$$

where $tr_1 = (q_2, \emptyset, true, true, m_1.out?ctrl.intensity, id, q_1)$

4.

$$\text{EMPTY}\ \frac{lf^3_{ctrl.intensity} = \epsilon}{\begin{array}{l} \mathbb{G}_{lf^3_{ctrl.intensity}} = (\{q_3\}, q_3, \emptyset) \\ over\ \Sigma_{lf^3_{ctrl.intensity}} = (\emptyset, \emptyset) \qquad \emptyset \end{array}}$$

5.

$$\text{EMPTY}\ \frac{lf^4_{ctrl.intensity} = \epsilon}{\begin{array}{l} \mathbb{G}_{lf^4_{ctrl.intensity}} = (\{q_4\}, q_4, \emptyset) \\ over\ \Sigma_{lf^4_{ctrl.intensity}} = (\emptyset, \emptyset) \qquad \emptyset \end{array}}$$

6.

$$\text{ALT}\ \frac{\begin{array}{l} lf^1_{ctrl.intensity} = (\textbf{\textit{alt}}, o_1, lf^2_{ctrl.intensity}, o_2, lf^3_{ctrl.intensity}, lf^4_{ctrl.intensity}) \\ \mathbb{G}_{lf^2_{ctrl.intensity}}\ over\ \Sigma_{lf^2_{ctrl.intensity}} \\ \mathbb{G}_{lf^3_{ctrl.intensity}}\ over\ \Sigma_{lf^3_{ctrl.intensity}} \\ \mathbb{G}_{lf^4_{ctrl.intensity}}\ over\ \Sigma_{lf^4_{ctrl.intensity}} \qquad \emptyset \end{array}}{\begin{array}{l} \mathbb{G}_{lf^1_{ctrl.intensity}} = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, q_5, \{tr_0, tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7\}) \\ over\ \Sigma_{lf^1_{ctrl.intensity}} = (\{t_1, ctrl.intensity, i_{t_1}, sched_{ctrl.intensity}\}, \{m_2.in, m_1.out, start\}) \\ \{o_1, o_2\} \end{array}}$$

$$where \begin{cases} tr_2 = (q_5, \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_{ctrl.intensity})), start!o_1, id, q_2) \\ tr_3 = (q_5, \emptyset, true, last(\{o_1, o_2\}, sched_{ctrl.intensity}) = o_1, \tau, \\ \qquad id[sched_{ctrl.intensity} \leftarrow popLast(o_1, sched_{ctrl.intensity})], q_2) \\ tr_4 = (q_5, \emptyset, true, NOT(elem(\{o_1, o_2\}, sched_{ctrl.intensity})), start!o_2, id, q_3) \\ tr_5 = (q_5, \emptyset, true, last(\{o_1, o_2\}, sched_{ctrl.intensity}) = o_2, \tau, \\ \qquad id[sched_{ctrl.intensity} \leftarrow popLast(o_2, sched_{ctrl.intensity})], q_3) \\ tr_6 = (q_0, \emptyset, true, true, \tau, id, q_4) \\ tr_7 = (q_3, \emptyset, true, true, \tau, id, q_4) \end{cases}$$

7.

$$\text{LOOP}\ \frac{\begin{array}{l} lf^0_{ctrl.intensity} = (\textbf{\textit{loop}}, o, lf^1_{ctrl.intensity}), lf^0_{ctrl.intensity} \\ \mathbb{G}_{lf^1_{ctrl.intensity}}\ over\ \Sigma_{lf^1_{ctrl.intensity}} \\ \mathbb{G}_{lf^0_{ctrl.intensity}}\ over\ \Sigma_{lf^0_{ctrl.intensity}} \qquad \{o_1, o_2\} \end{array}}{\begin{array}{l} \mathbb{G}_{lf_{ctrl.intensity}} = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, q_4, \\ \qquad \{tr_0, tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8, tr_9, tr_{10}, tr_{11}\}) \\ over\ \Sigma_{lf_{ctrl.intensity}} = (\{t_1, ctrl.intensity, i_{t_1}, sched_{ctrl.intensity}\}, \{m_2.in, m_1.out, start\}) \\ \{o_1, o_2, o, o'\} \end{array}}$$

where

$$\begin{cases} tr_8 = (q_4, \emptyset, true, NOT(elem(\{o, o'\}, sched_{ctrl.intensity})), start!o, id, q_5) \\ tr_9 = (q_4, \emptyset, true, last(\{o, o'\}, sched_{ctrl.intensity}) = o, \tau, \\ \qquad id[sched_{ctrl.intensity} \leftarrow popLast(o, sched_{ctrl.intensity})], q_5) \\ tr_{10} = (q_4, \emptyset, true, NOT(elem(\{o, o'\}, sched_{ctrl.intensity})), start!o', id, q_6) \\ tr_{11} = (q_4, \emptyset, true, last(\{o, o'\}, sched_{ctrl.intensity}) = o', \tau, \\ \qquad id[sched_{ctrl.intensity} \leftarrow popLast(o', sched_{ctrl.intensity})], q_6) \end{cases}$$

Consider the TIOSTS in Figure 6.20 without the dotted transitions. It is the translation $\mathbb{G}_{lf_{ctrl.intensity}}$ *of the lifeline* $lf_{ctrl.intensity}$ *obtained by applying inductively the rules as detailed previously. The translation does not include the application of the completion operators. Now consider the TIOSTS with the dotted transitions which are the results of the input completion and the initialization operators. Thus, this TIOSTS is* $\overline{\mathbb{G}_{lf_{ctrl.intensity}}}$ *exactly the full translation of the lifeline.*

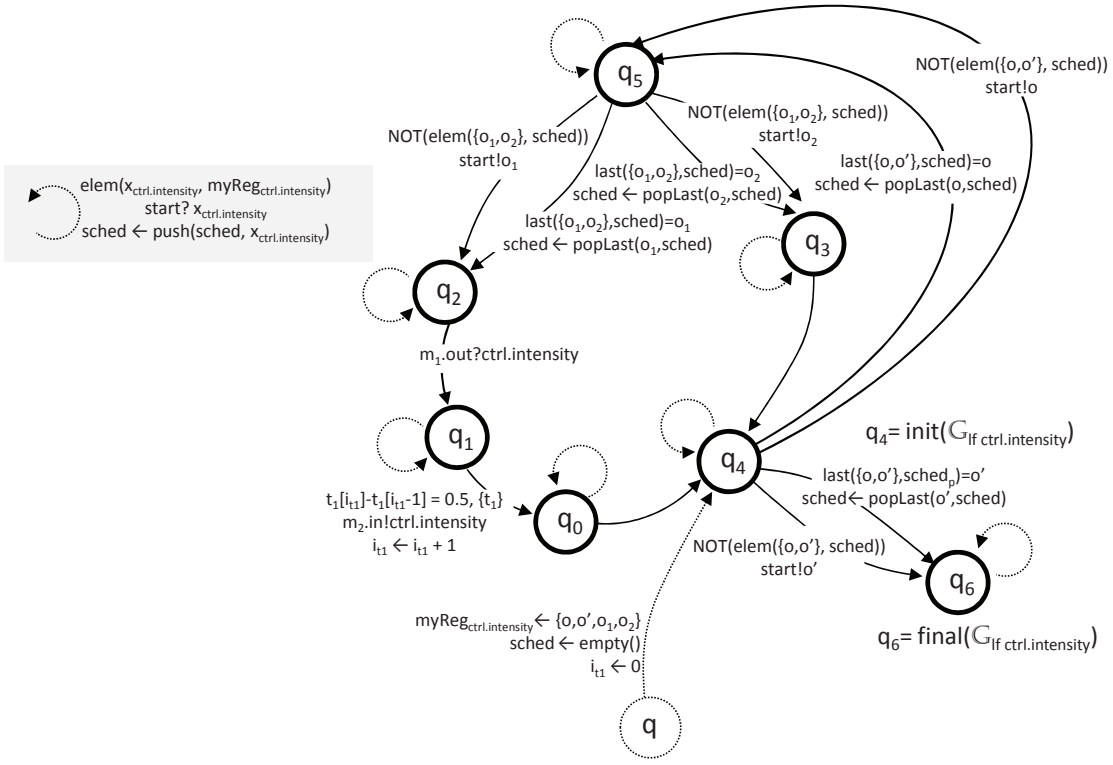


Figure 6.20: TIOSTS $\overline{\mathbb{G}_{lf_{ctrl.intensity}}}$ of the lifeline corresponding to the port *ctrl.intensity* in Figure 4.1

6.4 Symbolic execution of a sequence diagram

After the translation phase of a sequence diagram *sd*, we obtain a TIOSTS $\mathbb{G}_{sd}$. Based on Definition 46, the symbolic *symbolic execution of sd* is simply the symbolic execution $SE(\mathbb{G}_{sd})$ of its associated TIOSTS $\mathbb{G}_{sd}$ obtained by translation.

Note that $SE(\mathbb{G}_{sd})$ is a tree whose all paths denote in an abstract way all possible executions of $\mathbb{G}_{sd}$. By solving time and data path conditions, we obtain concrete actions and delays corresponding to the sequence of symbolic actions in paths and hence extract timed traces specified by *sd*.

In Figures 6.21– 6.22b, we illustrate the symbolic execution $SE(\mathbb{G}_{sd})$ where *sd* is the sequence diagram of the RWC system (see Figure 4.1). The symbolic tree has infinitely many executions

paths (because of the loop operator modeling reactive behaviors of RWC system), we just depict branches in Figure 6.21 which give enough intuition of how the execution works, at least at the beginning. Then we look in detail at a particular path of the symbolic tree depicted in Figure 6.22b.

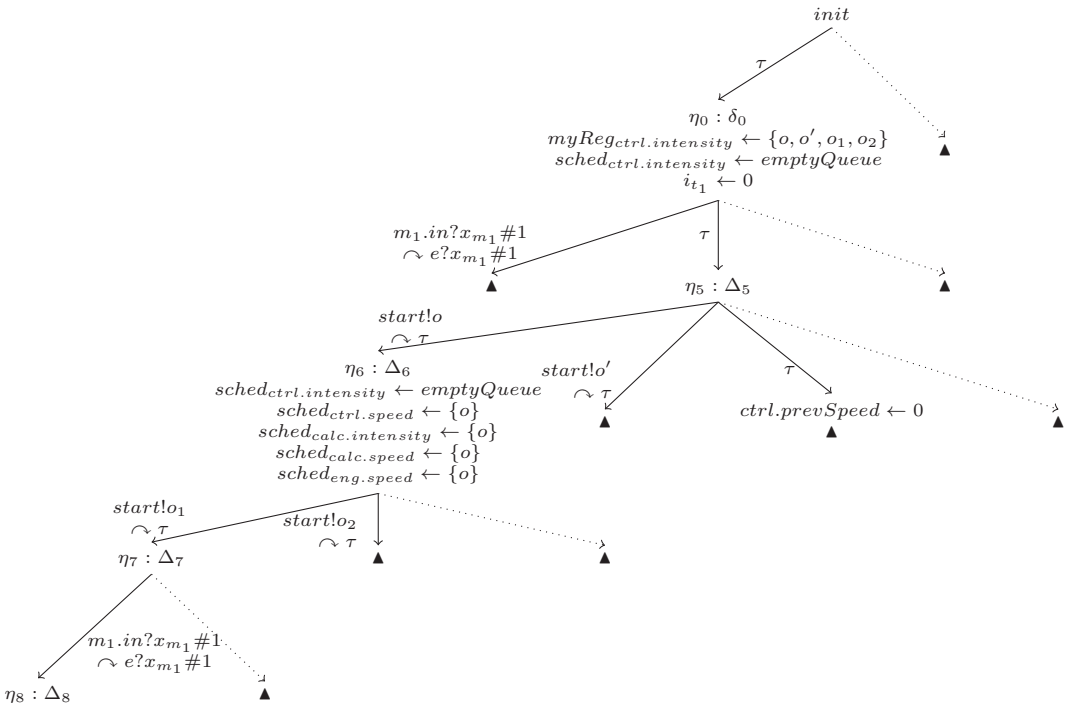


Figure 6.21: Symbolic execution of the Rain-sensor Wiper Controller system (partial view)

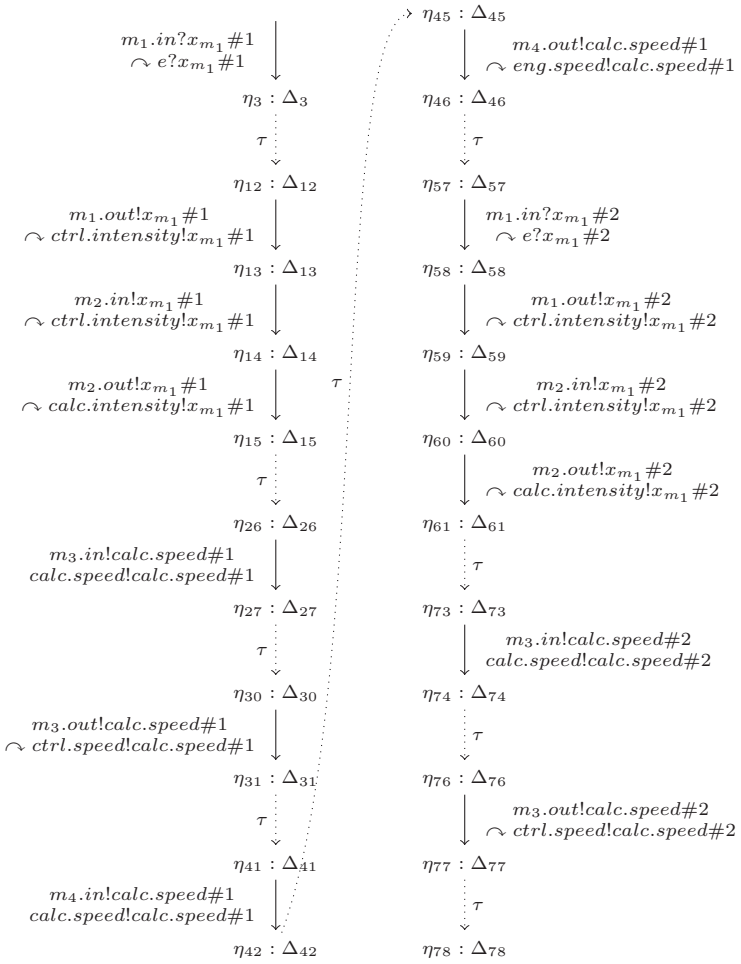
In Figure 6.22b, channels of the form *m.in* and *m.out* corresponding to a message name m are replaced by the source or the target port of m depending on the component point of view: e.g. The channel $m_3.in$ is replaced by port name *calc.speed* because the component *calc* emitted a speed value from that port through m_3 ($m_3 \in Msg_{(calc.speed,ctrl.speed)}$). The channel $m_3.out$ is replaced by port name *ctrl.speed* because the component *ctrl* received the speed value on that port through m_3. This improves readability by helping the reader to make links with Figure 4.1. Also all actions performed on channel *start* are replaced by τ because they are just artifacts to schedule the combining operators and do not have counterparts in the component interface (i.e. ports). These transformations are shown by curvy arrows.

Recall that instants at which actions occur are denoted by sum of symbolic durations introduced in the course of the symbolic execution ($\Delta_i = \sum_{j=0}^{i} \delta_j$). Similarly, $\Delta_{k \to i}$ means $\sum_{j=k}^{i} \delta_j$.

Now let us look at the structure of the tree. From the symbolic initial state *init*, the symbolic transition $init \to \eta_0$ corresponds to some execution in the TIOSTS product $\mathbb{G}_{sd}$ of the transition $q \to q_4$ of the TIOSTS $\mathbb{G}_{ctrl.intensity}$ (see the Figure 6.20): the substitution $\sigma(\eta_0)$ assigns to variables $myReg_{ctrl.intensity}, sched_{ctrl.intensity}, i_{t1}$ respectively the values $\{o, o', o_1, o_2\}, emptyQueue, 0$. Then, after executing similar transitions of the remaining TIOSTS in $\mathbb{G}_{sd}$ in the state from η_5, the loop region o may be entered by $lf_{ctrl.intensity}$ ($\eta_5 \xrightarrow{start!o} \eta_6$). In which case, the lifeline notifies the others by this choice (see the $\sigma(\eta_6)$, e.g. $sched_{ctrl.intensity}$ of the notifier is still assigned to the empty queue *emptyQueue* and $sched_{calc.intensity}$ is assigned by $\{o\}$, the chosen region). The other alternative behavior from η_5 is ignore the loop ($\eta_0 \xrightarrow{start!o'} .$).

Continuing with a particular path of the symbolic tree (depicted in Figure 6.22b), the reception of the intensity $calc.intensity!x_{m1}\#0$ is observed at time instant Δ_{15}. The first new speed value

emitted by *calc* after the first reception is not null ($0 \neq calc.speed\#1$) and the second value is equal to the last calculated speed ($calc.speed\#1 = calc.speed\#2$). From constraint $\Delta_{15\to60} = 0.5$, we deduce that the duration between the first reception of the rain intensity by the calculator ($calc.intensity!x_{m1}\#0$ at instant Δ_{15}) and the second reception ($calc.intensity!x_{m1}\#1$ at instant Δ_{61}) is at least of $0.5s$.



(a) Symbolic path

$$\pi(\eta_{78}) \begin{cases} \delta_{15} < 0.1 \\ \Delta_{15\to31} < 0.5 \\ 0 \neq calc.speed\#1 \\ \Delta_{15\to60} = 0.5 \\ \delta_{61} < 0.1 \\ \Delta_{61\to77} < 0.5 \\ calc.speed\#1 = calc.speed\#2 \end{cases}$$

(b) Path conditions

Figure 6.22

Chapter 7

Application to testing

Contents

In Chapter 6, we have shown how to associate semantics to sequence diagrams in the form of a set of timed traces of a TIOLTS. In this chapter, we study how to use sequence diagrams as references for testing. In Section 7.1, we present the *tioco* conformance relation [20, 54, 81] that we use as a basis. Section 7.2 presents compositional results relating correctness of systems and correctness of components composing them. Such results are a first attempt to define an incremental approach for testing in which a system would be tested pieces by pieces rather than as a whole. Finally, as all results above are defined in the TIOLTS framework, we have to relate them to sequence diagrams which is done in Section 7.3.

7.1 Testing framework

In this section, we introduce the conformance relation *tioco*, that grounds our testing framework. Subsection 7.1.1 is dedicated to the characterization of a system under test while subsection 7.1.2 presents the conformance relation.

7.1.1 System Under Test

In order to denote a conformance relation in a mathematical way, the first step is to mathematically represent the objects it relates. The conformance relation is supposed to define the correctness of implementations (or *systems under test*, SUT for short) with respect to specifications. In our approach specifications are given in the form of sequence diagrams that may be associated with a TIOLTS. Now we are interested in black box testing framework. In such a framework, SUT are only observable by means of traces that a tester builds while interacting with the SUT. Therefore, a SUT can be conventionally represented as a TIOLTS that we do not know but for which we can discover associated traces by interacting with it.

Definition 54 (System Under Test) *Let C be a set of channels. An* System Under Test over C *is a TIOLTS $\mathbb{A} = (Q, q_0, T)$ over C satisfying the following property:*

- **Input enableness***: for all q in Q, for all c in C_u and v in M there exists tr in T of the form $(q, c?v, q')$,*

- **Time elapsing**: *for all q in Q such that there are no transitions in T whose source state is q and whose action is τ or an output, there exists q' in Q and d in M_I such that (q, d, q') in T,*
- **Time decomposition**: *for any q_1 and q_2 in Q, for any d_1 and d_2 in M_I, if $(q_1, d_1 + d_2, q_2)$ in T then there exists q in Q such that (q_1, d_1, q) is in T and (q, d_2, q_2) in T,*
- **Time additivity**: *for any q_1, q_2 and q_3 in Q and for any d_1 and d_2 in M_I, if (q_1, d_1, q_2) and (q_2, d_2, q_3) are in T then $(q_1, d_1 + d_2, q_3)$ is in T,*
- **τ-closure**: *for any q_1, q_2 and q_3 in Q, and for any d in M_I, for any two transitions (q_1, a_1, q_2) and (q_2, a_2, q_3) in T such that (a_1, a_2) is (d, τ) or (τ, d), we have (q_1, d, q_3) in T.*

Input enableness is very classical in testing and only states that from the point of view of the tester, SUT can not refuse an input. **Time elapsing** ensures that the absence of reaction of SUT amounts to observing a waiting time during which no output occurs. **Time decomposition** states that any possible decomposition of durations is taken into account in the SUT. Conversely, we assume **time additivity** which states that any merging of subsequent delays into one delay is taken into account in SUT. Those two properties are important for ensuring that the product of two SUT reflect correctly the system resulting of their connection: in particular it permits to ensure the ability to apply the item **synchronous time passing** of Definition 22.

Example 44 (System under test) *Let us consider Figure 7.1. It illustrates a SUT over the set of communication channels $C = \{c_1, c_2\}$. We consider that M_I is isomorphic to natural numbers and we consider that inputs and outputs are natural numbers too.*

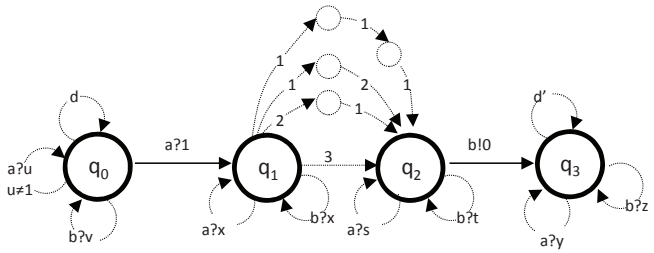


Figure 7.1: System under test SUT_1

In the depicted TIOLTS there are some elements which are denoted by symbols because otherwise there would be an infinity of transitions to represent. For the same reason we do not represent transitions with zero delay. The TIOLTS is input enabled. Consider for example the state q_0, the TIOLTS accepts all inputs: $a?1$, $b?v$ and $a?u$ where v is any natural number and u is any natural number different from 1 (since the input is already represented, see the transition $q_0 \xrightarrow{a?1} q_1$).

Now consider all the transitions making the system evolve from q_1 to q_2. They illustrate the properties of time decomposition/additivity by an arbitrary decomposition of the delay 3 as a sum of delays $1+1+1$, $1+2$ and $2+1$. Finally the time elapsed property is respected: in the states q_0 and q_3, there are two looping transitions respectively with delays d and d' allowing time to elapse since there is no reaction of the system, i.e. output or an unobservable action. Note that those transitions are illustrative to time decomposition/additivity properties.

7.1.2 Timed conformance relation

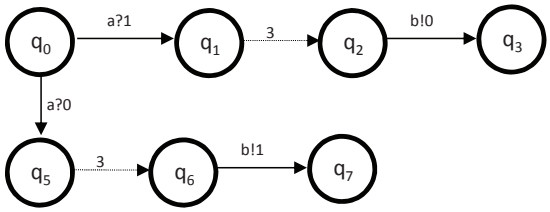
Let us now introduce the *tioco* relation defined in [20, 54, 81]. Intuitively a TIOLTS $\mathbb{A}_1$ conforms to a TIOLTS $\mathbb{A}_2$ for the *tioco* relation if and only if for any timed trace σ common to $\mathbb{A}_1$ and $\mathbb{A}_2$, any reaction (output or delay) of $\mathbb{A}_1$ after σ is also a possible reaction of $\mathbb{A}_2$ after σ.

Definition 55 (tioco) *Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be two TIOLTS over the same set of channels C.* $\mathbb{A}_1$ conforms to $\mathbb{A}_2$, *denoted $\mathbb{A}_1$ tioco $\mathbb{A}_2$, if and only if:*

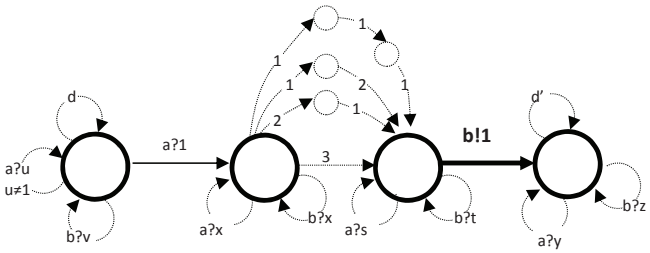
for all σ in $TTraces(\mathbb{A}_1) \cap TTraces(\mathbb{A}_2)$ and r in $O_M(C) \cup M_I$,

$$\sigma.r \in TTraces(\mathbb{A}_1) \Rightarrow \sigma.r \in TTraces(\mathbb{A}_2).$$

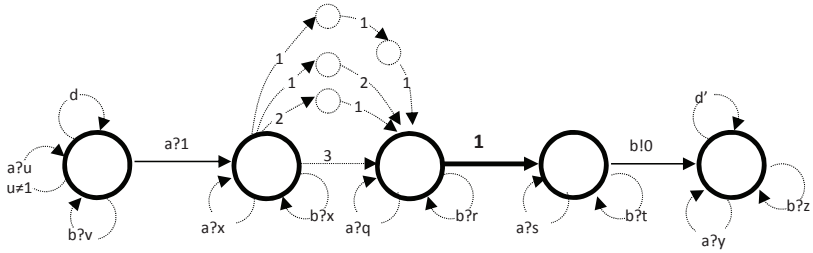
Example 45 (tioco) *Consider the TIOLTS $\mathbb{A}$ in Figure 7.2a as a reference specification.*



(a) Specification $\mathbb{A}$



(b) SUT_2 ~~tioco~~ $\mathbb{A}$: Not conform output



(c) SUT_3 ~~tioco~~ $\mathbb{A}$: Not conform delay

Figure 7.2: Not conform SUT

Based on the TIOLTS $\mathbb{A}$ of Figure 7.2a, we can notice that SUT_1 of Figure 7.1 ***tioco*** $\mathbb{A}$, *because after any specified sequence of actions and delay (trace), the outputs and delay emitted by SUT_1 are also specified. Even if the SUT does not realize all the behaviors of $\mathbb{A}$ (Specifically the one on the lower part of Figure 7.2a).*

Figure 7.2 depicts different system under test.

SUT_2 in Figure 7.2b does not conform to $\mathbb{A}$ (in the sense of tioco). This is because after the trace $tra = (a?1).3$, in state q_1 SUT_2 produces the output $(b!1)$ while $\mathbb{A}$ after tra produces $(b!0)$ which breaks tioco relation.

Finally, SUT_3 does not conform to $\mathbb{A}$, because after the specified trace $tra' = (a?1).3$, the non-specified delay 1 can be observed. It means that this realization of the system allows in all a delay of 4 time slots before outputting $b!0$. However the intended behavior is to have a delay of exactly 3 time slots.

7.2 Results

We want to state results allowing a tester to incrementally test a system during its design phase. Let us suppose that we know the specification of a system built by synchronizing two TIOLTS $\mathbb{A}_1$ and $\mathbb{A}_2$. Our system specification consists in $\mathbb{A}_1||\mathbb{A}_2$. By analyzing $\mathbb{A}_1||\mathbb{A}_2$ we can identify behaviors of $\mathbb{A}_1$ (respectively $\mathbb{A}_2$) that are involved in $\mathbb{A}_1||\mathbb{A}_2$. In order to explain what we mean here, let us consider an example. Let $\mathbb{A}_1$ (respectively $\mathbb{A}_2$) be the TIOLTS of Figure 7.3a (respectively of Figure 7.3b).

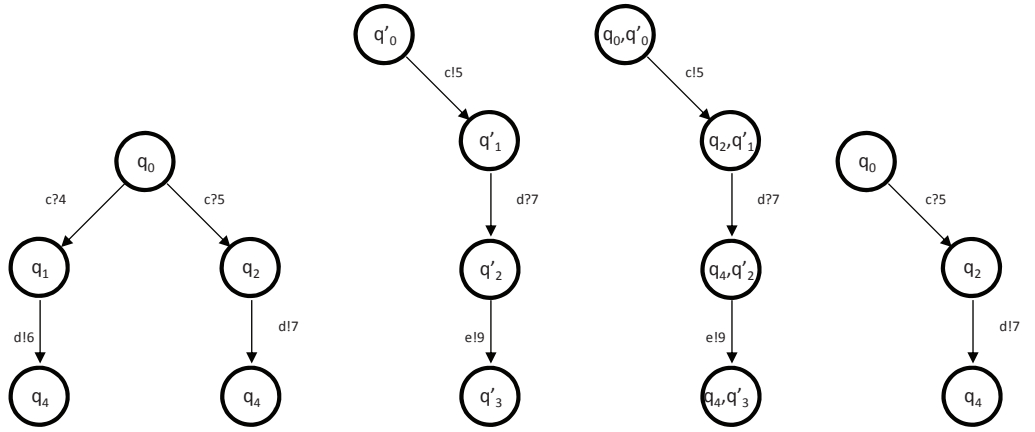


Figure 7.3: Unitary behaviors by projection

The product $\mathbb{A}_1||\mathbb{A}_2$ is depicted in Figure 7.3c.

Note that the transitions $(q_0, c?4, q_1)$ and $(q_1, d!6, q_3)$ of $\mathbb{A}_1$, can not be used to build transitions of $\mathbb{A}_1||\mathbb{A}_2$ in any item of Definition 22 because those two transitions introduce actions defined on channels shared with $\mathbb{A}_2$, and $\mathbb{A}_2$ does not contain any transitions that can be synchronized with them. The TIOLTS containing all transitions of $\mathbb{A}_1$ which can be used to build transitions of $\mathbb{A}_1||\mathbb{A}_2$ is depicted in Figure 7.3d.

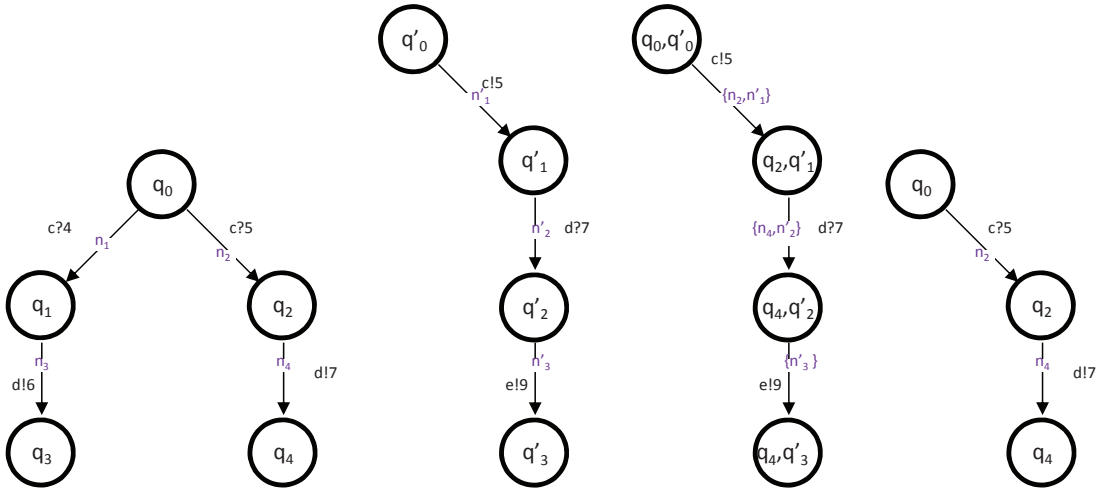
We say that the set of finite paths of the TIOLTS depicted in Figure 7.3d depicts all the behaviors of $\mathbb{A}_1$ that are involved in $(\mathbb{A}_1||\mathbb{A}_2)$. The TIOLTS depicted in Figure 7.3d is called the *projection of* $(\mathbb{A}_1||\mathbb{A}_2)$ *on* $\mathbb{A}_1$ and is denoted $(\mathbb{A}_1||\mathbb{A}_2)_{\mathbb{A}_1}$. In order to define such a projection for any two TIOLTS $\mathbb{A}_1$ and $\mathbb{A}_2$, we use a naming functions $name_{\mathbb{A}_1||\mathbb{A}_2} : Trans(\mathbb{A}_1||\mathbb{A}_2) \rightarrow 2^{TN}$ which is an adaptation of the naming function defined for TIOSTS in Definitions 36– 37 – 38 of Section 5.1). It suffices to replace the word "TIOSTS" by "TIOLTS" in those definitions to obtain the formal definition of $name(\mathbb{A})$."

Definition 56 (TIOLTS projection) *Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be two TIOLTS. For any $i \in \{1, 2\}$, the* projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_i$*, denoted $(\mathbb{A}_1||\mathbb{A}_2)_{\mathbb{A}_i}$ is the TIOLTS $(state(\mathbb{A}_i), init(\mathbb{A}_i), T)$ where T is the set such that for all $tr \in (\mathbb{A}_1||\mathbb{A}_2)$: if there exists $tr' \in Trans(\mathbb{A}_i)$ such that $name_{\mathbb{A}_i}(tr') \subseteq name_{\mathbb{A}_1||\mathbb{A}_2}(tr)$ then tr' is in T. tr' is called the* projection of tr on $\mathbb{A}_i$ *and is denoted $tr_{\mathbb{A}_i}$.*

Note that in Definition 56, if such a transition tr' exists it is necessarily unique by definition

of naming functions. In the sequel when there is no transition tr' in $Trans(\mathbb{A}_i)$ such that $name_{\mathbb{A}_i}(tr') \subseteq name_{\mathbb{A}_1||\mathbb{A}_2}(tr)$, we say that $tr_{\mathbb{A}_i}$ is undefined. If there is tr' in $Trans(\mathbb{A}_i)$ such that $name_{\mathbb{A}_i}(tr') \subseteq name_{\mathbb{A}_1||\mathbb{A}_2}(tr)$, we say that $tr_{\mathbb{A}_i}$ is defined.

Example 46 *Consider again the example in Figure 7.3. Actually $\mathbb{A}_1$ and $\mathbb{A}_2$ are basic TIOSTS associated respectively with the naming functions $name_{\mathbb{A}_1}$ and $name_{\mathbb{A}_2}$ (see respectively Figures 7.4a–7.4b, names are colored labels on transitions edges). These are some examples of a transition names : $name_{\mathbb{A}_1}((q_0, c?5, q_2)) = \{n_2\}$ and $name_{\mathbb{A}_2}((q'_0, c!5, q'_1)) = \{n'_1\}$. The product $\mathbb{A}_1||\mathbb{A}_2$ is associated with the naming function $name_{\mathbb{A}_1||\mathbb{A}_2}$ computed by applying the Definition 38 (see Figure 7.4c). An example of transition name in the product is $name_{\mathbb{A}_1||\mathbb{A}_2}(((q_0, q'_0), c!5, (q_2, q'_1))) = \{n_2, n'_1\}$ (obtained by synchronizing the two transitions mentioned previously). In this settings, the projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_1$ is given in Figure 7.4d : By applying Definition 56, we retain the two transitions of $\mathbb{A}_1$ named $\{n_2\}$ and $\{n_4\}$ because they correspond to some transitions in $\mathbb{A}_1||\mathbb{A}_2$. That is we have $\{n_2\} \subseteq \{n_2, n'_1\}$ and $\{n_4\} \subseteq \{n_4, n'_2\}$.*



(a) $name_{\mathbb{A}_1}$ (b) $name_{\mathbb{A}_2}$ (c) $name_{\mathbb{A}_1||\mathbb{A}_2}$ (d) $(\mathbb{A}_1||\mathbb{A}_2)_{\mathbb{A}_1}$: the projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_1$

Figure 7.4: Unitary behaviors by projection

We now extend the projection of TIOLTS to both paths and traces.

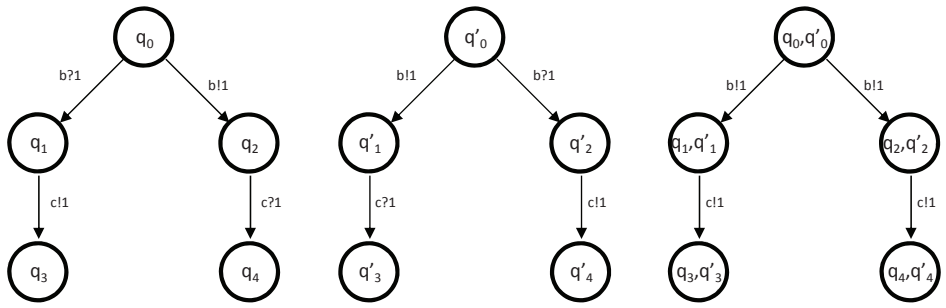
Definition 57 (Path projection) *With notations of Definition 56, for any path $p \in FP(\mathbb{A}_1||\mathbb{A}_2)$, the* projection of p on $\mathbb{A}_i$, *denoted $p_{\mathbb{A}_i}$ is the path of $FP(\mathbb{A}_i)$ inductively defined as follow:*

- *if p is the empty path ε then we have $p_{\mathbb{A}_i}$ is ε,*
- *if p is of the form $p'.tr$ where $p' \in FP(\mathbb{A}_1||\mathbb{A}_2)$ and $tr \in Trans(\mathbb{A}_1||\mathbb{A}_2)$ then:*
  - *if $tr_{\mathbb{A}_i}$ is defined we have $p_{\mathbb{A}_i} = p'_{\mathbb{A}_i}.tr_{\mathbb{A}_i}$,*
  - *if $tr_{\mathbb{A}_i}$ is not defined we have $p_{\mathbb{A}_i} = p'_{\mathbb{A}_i}$.*

The extension of the projections to timed traces is a bit more difficult to define than the one defined for path for two reasons. The first reason is that a timed trace in $\mathbb{A}_1||\mathbb{A}_2$ may be projected in several ways on $\mathbb{A}_1$ or on $\mathbb{A}_2$: an output action $c!v$ (where c is a channel shared between $\mathbb{A}_1$ and $\mathbb{A}_2$) introduced in a transition of $\mathbb{A}_1||\mathbb{A}_2$ may be mapped on an input or an output on $\mathbb{A}_1$ or on $\mathbb{A}_2$. Indeed, $c!v$ may result from a synchronization of $c?v$ in $\mathbb{A}_1$ and $c!v$ in $\mathbb{A}_2$ but also from

a synchronization of $c!v$ in $\mathbb{A}_1$ and $c?v$ in $\mathbb{A}_2$ (see Definition 22). Moreover those cases are not exclusive since a TIOLTS may introduce a transition with an input on c and another transition with an output on c. The next example illustrate this point.

Example 47 *The example in Figure 7.5 shows that the projection of a trace of a product is not necessarily a singleton.*



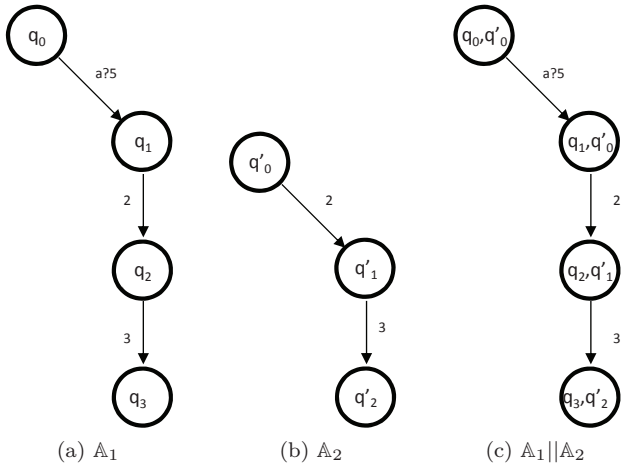
(a) $\mathbb{A}_1$ (b) $\mathbb{A}_2$ (c) $\mathbb{A}_1||\mathbb{A}_2$

Figure 7.5: One trace/two projections

The sequence $b!1.c!1$ *is a trace of* $TTraces(\mathbb{A}_1||\mathbb{A}_2)$*. This trace may originate from two possible paths in* $TTraces(\mathbb{A}_1||\mathbb{A}_2)$ *in each of which the roles of* $\mathbb{A}_1$ *and* $\mathbb{A}_2$ *is different. Let us focus on the role of* $\mathbb{A}_1$*. The first path (in left side of Figure 7.5) corresponds to* $\mathbb{A}_1$ *performing successively* $b?1$ *and* $c!1$*. The second one (in right side of Figure 7.5) corresponds to* $\mathbb{A}_1$ *performing successively* $b!1$ *and* $c?1$*. Therefore, the projection of* $b!1.c!1$ *on* $\mathbb{A}_1$ *has to return both possibilities, that is, both traces* $b?1.c!1$ *and* $b!1.c?1$ *of* $TTraces(\mathbb{A}_1)$*.*

The second difficulty is that durations occurring in a timed trace may come from decompositions or re-compositions of durations introduced in transitions (See Definition 21).

Example 48 *Consider the product defined in Figure 7.6.*



(a) $\mathbb{A}_1$ (b) $\mathbb{A}_2$ (c) $\mathbb{A}_1||\mathbb{A}_2$

Figure 7.6: Durations in the product

Let us look at the following finite path in $\mathbb{A}_1||\mathbb{A}_2$ *:* $p = (q_0, q'_0) \xrightarrow{a?5} (q_1, q'_0) \xrightarrow{2} (q_2, q'_1) \xrightarrow{3} (q_3, q'_2)$*. First the trace of* p *according to Definition 20 is* $trace(p) = a?5.2.3$*. These are some timed traces of* p *obtained by delays decomposition/re-composition from* $trace(p)$*, i.e. in* $ttraces(p)$ *:*

$a?5.\overbrace{1.1}^{2}.\overbrace{1.1.1}^{3}$ *(decomposition)*

$a?5.\overbrace{4}^{1.1.1.1}.1$ *(recomposition)*

Note that $a?5.4.1$ *is a trace of* p *since the subsequent delays* 2 *then* 3 *may be decomposed and recomposed to get* 4 *then* 1*. The sequence* $a?5.4$ *is also a trace of* $\mathbb{A}_1||\mathbb{A}_2$ *because* $a?5.4.1$ *is a trace of* p *and timed traces of TIOLTS are stable by prefix for delays (see Definition 21). Therefore the projection mechanism which explores all the paths* $\mathbb{A}_1||\mathbb{A}_2$ *has to accept* $a?5.4$ *as a trace of* p *even though the delay* 4 *does not occur explicitly in* p*. Assuming for example that* $q_0 \xrightarrow{a?5} q_1$ *does not originate from a transition in* $\mathbb{A}_1$*, typically the projection of* $a?5.4$ *on* $\mathbb{A}_1$ *returns simply the trace* 4 *and the projection on* $\mathbb{A}_2$ *returns* $a?5.4$*.*

Definition 58 characterizes the projection of a timed trace σ.

Definition 58 (Trace projection) *With notations of Definition 57, for any* $\sigma \in TTrace(\mathbb{A}_1||\mathbb{A}_2)$*, the set of projections of* σ *on* $\mathbb{A}_i$*, denoted* $Proj_{\mathbb{A}_i}(\sigma)$ *is the set* $\bigcup_{p \in FP(\mathbb{A}_1||\mathbb{A}_2)} Proj_{\mathbb{A}_i}(p, \sigma)$ *where* $Proj_{\mathbb{A}_i}(p, \sigma) \subseteq TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}})$ *is the empty set if* σ *is not a prefix of a timed trace in* $ttraces(p)$ *and otherwise is defined as follow:*

- *if* σ *is the empty trace* ε *we have* $Proj_{\mathbb{A}_i}(p, \sigma) = \{\varepsilon\}$*,*
- *if* p *is not the empty path let us note* p *as* $p'.tr$ *where* p' *is a finite path and* tr *is a transition:*
  - *if* $act(tr) \in I_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2)) \cup O_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2))$ *let us note* σ *as* $\sigma'.a$ *where* σ' *is a timed trace and* a *is an action:*
    - *if* $act(tr) \neq a$ *or* $\sigma' \notin ttraces(p')$ *we have* $Proj_{\mathbb{A}_i}(p, \sigma) = Proj_{\mathbb{A}_i}(p', \sigma)$*,*
    - *if* $\sigma' \in ttraces(p')$ *and* $a = act(tr)$*:*
      - *if* $tr_{\mathbb{A}_i}$ *is defined we have* $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p',\mathbb{A}_i}.act(tr_{\mathbb{A}_i}) \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p', \sigma')\}$*,*
      - *if* $tr_{\mathbb{A}_i}$ *is not defined we have* $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p',\mathbb{A}_i} \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p', \sigma')\}$*,*
  - *if* $act(tr) \in M_I$*:*
    - *if* σ *can be decomposed as* $\sigma'.a$ *where* σ' *is a timed trace and where* a *is an action such that* $a \notin M_I$ *then we have* $Proj_{\mathbb{A}_i}(p, \sigma) = Proj_{\mathbb{A}_i}(p', \sigma)$*,*
    - *if* σ *can be decomposed as* $\sigma'.d_0 \cdots d_N$ *where for all* $i \leq N$*,* $d_i \in M_I$*, and* σ' *is either the empty trace or a trace of the form* $\sigma''.b$ *where* σ'' *is a timed trace and* b *is an action such that* $b \notin M_I$*,*
      let us decompose p *as* $p''.tr_0 \cdots tr_M$ *where for all* $j \leq M$ tr_j *is a transition such that* $act(tr_j) \in M_I$*, and where* p'' *is either the empty path or a path of the form* $p'''.tr''$ *where* p''' *is a finite path and* tr'' *is a transition such that* $act(tr'') \notin M_I$*:*
      - *if* $\Sigma_{i=0}^{N} d_i > \Sigma_{j=0}^{M} act(tr_j)$ *or if* $\sigma' \notin ttraces(p'')$ *then we have* $Proj_{\mathbb{A}_i}(p, \sigma) = Proj_{\mathbb{A}_i}(p'', \sigma)$*,*
      - *if* $\Sigma_{i=0}^{N} d_i \leq \Sigma_{j=0}^{M} act(tr_j)$ *and* $\sigma' \in ttraces(p'')$ *then we have* $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p'',\mathbb{A}_i}.d_0 \cdots d_N \backslash \sigma'_{p'',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p'', \sigma')\}$*,*

Let us discuss Definition 58. As exemplified previously, for any $i \in \{1, 2\}$ the projection $Proj_{\mathbb{A}_i}(\sigma)$ of a trace σ of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_i$ is a set which is not necessarily reduced to a singleton. Many paths of $\mathbb{A}_1||\mathbb{A}_2$ may have σ as a trace, and hence, we need to visit all paths of $\mathbb{A}_1||\mathbb{A}_2$:

$\bigcup_{p \in FP(\mathbb{A}_1||\mathbb{A}_2)} Proj_{\mathbb{A}_i}(p, \sigma)$ where $Proj_{\mathbb{A}_i}(p, \sigma)$ is the trace obtained by projecting σ on a given path p.

We focus now on how the projection $Proj_{\mathbb{A}_i}(p, \sigma)$ is computed. Obviously when σ is not a prefix of $ttraces(p)$ then $Proj_{\mathbb{A}_i}(p, \sigma)$ is the empty set.

For instance, the following trace $a?5.b!5.2$ of some given product of TIOLTS is a prefix of a timed trace of the following path $p : q_0 \xrightarrow{a?5} q_1 \xrightarrow{b!5} q_2 \xrightarrow{4} q_3$. However the trace $\sigma = a?5.b!6$ is not, in which case $Proj_{\mathbb{A}_i}(p, \sigma) = \emptyset$.

The other trivial case is when σ is the empty trace, i.e. empty word ϵ, we have then $Proj_{\mathbb{A}_i}(p, \epsilon) = \{\epsilon\}$: that is the projection of the empty trace is the empty trace (see the first item of Definition 58, note that the trace ϵ is a prefix of any path trace).

Let us consider now the case of a non empty path p (see the second item of Definition 58) and $\sigma \neq \epsilon$ is a prefix of a trace of p, $Proj_{\mathbb{A}_i}(p, \sigma)$ defined by induction on the sequence of transitions in p. In fact, p may be decomposed as a path p' and a subsequent transition tr ($p = p'.tr$). Here two cases are possible either $act(tr)$ is an input/output action ($act(tr) \in I_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2)) \cup O_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2))$) or a delay ($act(tr) \in M_I$). We discuss in the following these two cases starting with $act(tr)$ being an input/output action.

The trace σ is decomposed into a trace σ' followed by an action a ($\sigma = \sigma'.a$). The comparison of a and $act(tr)$ decides of the projection :

- When they do not coincide ($act(tr) \neq a$), we deduce that p goes beyond σ. Simply σ is a trace obtained from a shorter path overlapping with p. This is an example of such situation :

  $\sigma = \underbrace{a?5}_{\sigma'} . \underbrace{b!5}_{a}$

  $p = \underbrace{q_0 \xrightarrow{a?5} q_1 \xrightarrow{b!5}}_{p'} \underbrace{q_2 \xrightarrow{\overbrace{c!6}^{act(tr)}} q_3}_{tr}$

  Note that $\sigma = a?5.b!5$ is a prefix of a trace of p. Regarding p, the shortest path carrying $a?5.b!5$ is not p but rather p' with q_2 as a target state. This is captured by the fact that $act(tr) = c!6$ is different from $a = b!5$. Now for p', $act(p') = b!5$ coincide with $a = b!5$ in which case the projection is defined as discussed in the the next item.

  However even when a and $act(tr)$ coincide ($act(tr) = a$), p may go beyond σ. Consider for instance this example :

  $\sigma = \underbrace{a?5}_{\sigma'} . \underbrace{b!5}_{a}$

  $p = \underbrace{q_0 \xrightarrow{a?5} q_1 \xrightarrow{b!5} q_2 \xrightarrow{c!6}}_{p'} \underbrace{q_3 \xrightarrow{\overbrace{b!5}^{act(tr)}} q_4}_{tr}$

  In the example, $a = b!5$ and $act(tr) = b!5$ are indeed equal. Obviously this is because $b!5$ occurs twice in p and not because p is the shortest path carrying σ. Such a situation is captured by the fact that $\sigma' \notin ttraces(p')$. In the example, we have $a?5.b!5 \notin ttraces(p')$ because p' itself goes beyond σ'.

  In both discussed cases, $Proj_{\mathbb{A}_i}(p, \sigma)$ is equal to the inductively computed projection $Proj_{\mathbb{A}_i}(p', \sigma)$. That is the transition tr is ignored and the visiting of $p = p'.tr$ proceeds with p'.

- When σ' and $act(tr)$ do not satisfy any of the previous cases, i.e. $\sigma' \in ttraces(p')$ and $a = act(tr)$, $Proj_{\mathbb{A}_i}(p, \sigma)$ is defined as follows :
  - "if $tr_{\mathbb{A}_i}$ is defined we have $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p',\mathbb{A}_i}.act(tr_{\mathbb{A}_i}) \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p', \sigma')\}$". Let us comment this item of Definition 58. Recall that $tr_{\mathbb{A}_i}$ is called the projection of tr on $\mathbb{A}_i$. $tr_{\mathbb{A}_i}$ is defined means that tr originates from a transition in $\mathbb{A}_i$. That is $tr_{\mathbb{A}_i} \in Trans(\mathbb{A}_i)$ such that $name_{\mathbb{A}_i}(tr) \subseteq name_{\mathbb{A}_1||\mathbb{A}_2}(tr)$ (see Definition 56). Therefore, the action of $act(tr_{\mathbb{A}_i})$ is retained in the projection of σ on $\mathbb{A}_i$ ($\sigma'_{p',\mathbb{A}_i}.act(tr_{\mathbb{A}_i})$), where the inductively defined projection $Proj_{\mathbb{A}_i}(p', \sigma')$). Here is an illustration :
    $\sigma = \underbrace{a?5}_{\sigma'} . \underbrace{b!5}_{a}$
    $p = \underbrace{(q_0, q'_0) \xrightarrow[\{n'_1\}]{a?5} (q_0, q'_1)}_{p'} \underbrace{\xrightarrow[\{n_1, n'_2\}]{b!5} (q_1, q'_2)}_{tr}$
    Note that n_1 is a transition name of $\mathbb{A}_1$ and n'_1, n'_2 are names of $\mathbb{A}_2$. We have then $Proj_{\mathbb{A}_1}(p, a?5.b!5) = \{\sigma'_{p',\mathbb{A}_1}.b?5 \backslash \sigma'_{p',\mathbb{A}_1} \in Proj_{\mathbb{A}_1}(p', a?5)\}$ given $tr_{\mathbb{A}_1}$ is $q_0 \xrightarrow[\{n_1\}]{b?5} q_1$.
  - "if $tr_{\mathbb{A}_i}$ is not defined we have $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p',\mathbb{A}_i}.act(tr_{\mathbb{A}_i}) \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p', \sigma')\}$". $tr_{\mathbb{A}_i}$ is not defined means that tr does not originate from a transition in $\mathbb{A}_i$. Hence $act(tr)$ a ignored in the projection. Looking at the previous example, we have $Proj_{\mathbb{A}_1}(p', a?5) = \{\sigma'_{\epsilon,\mathbb{A}_1} \backslash \sigma'_{\epsilon,\mathbb{A}_1} \in Proj_{\mathbb{A}_1}(\epsilon, \epsilon)\} = \{\epsilon\}$ because $tr_{\mathbb{A}_1}$ is not defined. Now, $tr_{\mathbb{A}_2}$ is defined and equal to $q'_0 \xrightarrow[\{n'_1\}]{a?5} q'_1$. So we have $Proj_{\mathbb{A}_2}(p', a?5) = \{a?5\}$.

Recall that we have identified two cases either $act(tr)$ is an input/output action or a delay. Recall that p is of the form $p'.tr$ where tr is a transition. We have discussed the first case. We are interested in the second case namely $act(tr) \in M_I$. We pay attention to all possible decompositions/re-composition of delays as defined in Definition 21.

Again we first look at the structure of σ. When σ is of the form $\sigma'.a$ where a is an input/output action and not a delay ($a \notin M_I$), it means that p which ends with a delay action goes beyond σ. Obviously the visiting of p by the projection continues with p' ($Proj_{\mathbb{A}_i}(p, \sigma) = Proj_{\mathbb{A}_i}(p', \sigma)$). Otherwise σ ends with a sequence of delays $d_0, \ldots, d_N$. These delays result from a decompositions/re-composition of delays originally occurring in p in particular they may involve $act(tr) \in M_I$. Let us decompose further σ and p as follows :

$\sigma = \underbrace{\sigma'}_{\sigma''.b}.d_0.\ldots.d_N$ where $b \notin M_I$ and $\forall i \leq N, d_i \in M_I$

$p = \underbrace{p''}_{p''' \xrightarrow{tr''}} \xrightarrow{tr_0} \ldots \xrightarrow{tr_M}$ where tr''' is a transition such that $act(tr''') \notin M_I$ and $\forall j \leq M$, tr_j is a transition such that $act(tr_j) \in M_I$

As in the case of input/output action a of $\sigma = a.\sigma'$ identification in $p.tr$, we need to identify the delays $d_0, \ldots, d_N$ of $\sigma = \sigma'.d_0.\ldots.d_N$ in $p.tr_0 \ldots tr_M$:

- When the time elapsed since the last input/output action (b) in σ is greater than the one elapsed since the last input/output action ($act(tr''')$) occurring in p ($\Sigma_{i=0}^{N} d_i > \Sigma_{j=0}^{M} act(tr_j)$), we deduce that p goes beyond σ. For example in this illustration :
  $\sigma = \underbrace{\epsilon}_{\sigma''} . \underbrace{a?5}_{b} . \underbrace{1.1.1}_{d_0.d_1.d_2}$
  $p = \underbrace{q_0 \xrightarrow{a?5} q_1 \xrightarrow{3}}_{p'''} \underbrace{q_2 \xrightarrow{c!6} q_3}_{tr'''} \underbrace{\xrightarrow{1} q_4 \xrightarrow{1} q_5}_{\xrightarrow{tr_0} \xrightarrow{tr_1}}$

However this condition is not enough because in the case where it is not satisfied. That is the delays may coincide ($\Sigma_{i=0}^{N} d_i = \Sigma_{j=0}^{M} act(tr_j)$) or the delays in p may be decomposed in a way to involve those at the end of σ ($\Sigma_{i=0}^{N} d_i \leq \Sigma_{j=0}^{M} act(tr_j)$) and still p may go beyond σ as shown here :

$$\sigma = \underbrace{\underbrace{\epsilon}_{\sigma''} \cdot \underbrace{a?5}_{b}}_{\sigma'} \cdot \underbrace{1.1}_{d_0.d_1}$$

$$p = \underbrace{\underbrace{q_0 \xrightarrow{a?5} q_1}_{p'''} \xrightarrow{3} \underbrace{q_2 \xrightarrow{c!6} q_3}_{tr'''}}_{p''} \xrightarrow{1} q_4 \xrightarrow{1} q_5 \text{ or } p = \underbrace{q_0 \xrightarrow{a?5} q_1}_{p'''} \xrightarrow{3} \underbrace{q_2 \xrightarrow{c!6} q_3}_{tr'''} \xrightarrow{1} q_4 \xrightarrow{1} q_5 \xrightarrow{1} q_6$$

The condition to consider besides is that $\sigma' \notin ttraces(p'')$. In the example, we have $a?5 \notin ttraces(p'')$. In both discussed cases, $Proj_{\mathbb{A}_i}(p, \sigma)$ is equal to the inductively computed projection $Proj_{\mathbb{A}_i}(p'', \sigma)$. That is the transitions $tr_0 \dots tr_M$ are ignored and the visiting of $p = p''.tr_0 \dots tr_M$ proceeds with p''.

- When the time elapsed since the last input/output action (b) in σ is less than or equal to the time elapsed since the last input/output action ($act(tr''')$) occurring in p ($\Sigma_{i=0}^{N} d_i \leq \Sigma_{j=0}^{M} act(tr_j)$) and $\sigma' \in ttraces(p'')$, we deduce that σ is a trace of p. So we have $Proj_{\mathbb{A}_i}(p, \sigma) = \{\sigma'_{p'',\mathbb{A}_i}.d_0 \cdots d_N \backslash \sigma'_{p'',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p'', \sigma')\}$. This an example of such case :

$$\sigma = \underbrace{\underbrace{\epsilon}_{\sigma''} \cdot \underbrace{a?5}_{b}}_{\sigma'} \cdot \underbrace{1.1}_{d_0.d_1}$$

$$p = \underbrace{\underbrace{\epsilon}_{p'''} \underbrace{q_0 \xrightarrow{a?5} q_1}_{tr'''}}_{p''} \underbrace{\xrightarrow{1} q_2 \xrightarrow{1} q_3 \xrightarrow{1} q_4}$$

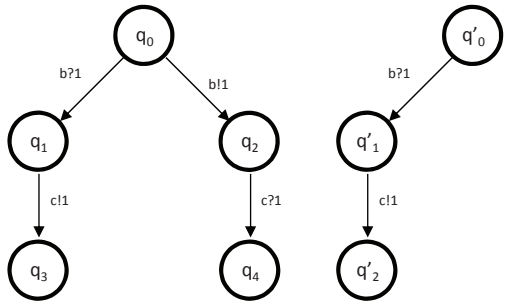
In the sequel we are interested in particular kind of TIOLTS called *TIOLTS with partitioned actions*. In such TIOLTS, a given channel can be used to send value or to receive value but not both.

Definition 59 (TIOLTS with partitioned actions) *A* TIOLTS with partitioned actions *is a TIOLTS $\mathbb{A}$ such that $Chan(\mathbb{A})$ is partitioned in two sets $Chan_I(\mathbb{A})$ and $Chan_O(\mathbb{A})$ (*i.e. *$Chan(\mathbb{A}) = Chan_I(\mathbb{A}) \cup Chan_O(\mathbb{A})$ and $Chan_I(\mathbb{A}) \cap Chan_O(\mathbb{A}) = \emptyset$) such that for any $tr \in Trans(\mathbb{A})$ if $act(tr)$ is of the form $c?u$ (respectively $c!u$) then we have $c \in Chan_I(\mathbb{A})$ (respectively $c \in Chan_O(\mathbb{A})$).*

TIOLTS with partitioned actions are simply TIOLTS in which one may differentiates channels used to receive values and channels used to send values.

Example 49 *In figure 7.7a, $\mathbb{A}$ is not a TIOLTS with partitioned actions because we have the channels b and c which are used at once to receive and send values. E.g. considering the transition $tr_1 : q_0 \xrightarrow{b?1} q_1$, we have $act(tr_1) = b?1$ is an input. The channel b is used again in $tr_2 : q_0 \xrightarrow{b!1} q_2$ where act $b!1$ is an output. The figure 7.7b illustrates a TIOLTS $\mathbb{A}'$ with partitioned actions. We have $Chan_I(\mathbb{A}') = \{b\}$ and $Chan_O(\mathbb{A}') = \{c\}$. That is b is used exclusively for inputs and c is used exclusively for outputs.*

Products of such TIOLTSs have a particular property: the projection of any of their traces is restricted to a singleton as stated in the following lemma.



(a) $\mathbb{A}$ is not a TIOLTS with partitioned actions

(b) $\mathbb{A}'$ is a TIOLTS with partitioned actions

Figure 7.7

Lemma 1 *Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be two TIOLTS with partitionned actions such that $Chan_I(\mathbb{A}_1) \cap Chan_I(\mathbb{A}_2) = \emptyset$ and $Chan_O(\mathbb{A}_1) \cap Chan_O(\mathbb{A}_2) = \emptyset$. For any $\sigma \in TTrace(\mathbb{A}_1||\mathbb{A}_2)$, for any $i \in \{1,2\}$ we have $Proj_{\mathbb{A}_i}(\sigma)$ is a singleton. We note $\sigma_{\mathbb{A}_i}$ the unique element of $Proj_{\mathbb{A}_i}(\sigma)$.*

Proof 1 (Lemma 1) *The proof is done by induction on the structure of σ:*

Basic case *: if σ is the empty trace then for any $p \in FP(\mathbb{A}_1||\mathbb{A}_2)$ we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\varepsilon\}$. Now $Proj_{\mathbb{A}_i}(\sigma)$ is defined as $\bigcup_{p \in FP(\mathbb{A}_1||\mathbb{A}_2)} Proj_{\mathbb{A}_i}(p,\sigma)$. Therefore $Proj_{\mathbb{A}_i}(\sigma)$ is $\{\varepsilon\}$ which is a singleton.*

Induction steps *:*

In all cases described thereafter, we suppose that for all prefix σ_p of σ such that $\sigma_p \neq \sigma$ we have $Proj_{\mathbb{A}_i}(\sigma')$ is a singleton, and we prove that $Proj_{\mathbb{A}_i}(\sigma)$ is a singleton.

- Let us suppose that σ is of the form $\sigma'.a$ with $a \in I_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2)) \cup O_M(Chan(\mathbb{A}_1) \cup Chan(\mathbb{A}_2))$. By hypothesis we have $Proj_{\mathbb{A}_i}(\sigma')$ is a singleton. Let us prove that $Proj_{\mathbb{A}_i}(\sigma)$ is a singleton.

  Since $\sigma \in TTrace(\mathbb{A}_1||\mathbb{A}_2)$, from Definition 20, we conclude that there exists a path p of the form $p'.tr$ where p' is a finite path and tr is a transition such that $\sigma' \in ttraces(p')$ and $a = act(tr)$.

  Now From Definition 58 we have:

  - *if $tr_{\mathbb{A}_i}$ is defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{p',\mathbb{A}_i}.act(tr_{\mathbb{A}_i}) \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p',\sigma')\}$,*
  - *if $tr_{\mathbb{A}_i}$ is not defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{p',\mathbb{A}_i} \backslash \sigma'_{p',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p',\sigma')\}$,*

  Now $Proj_{\mathbb{A}_i}(\sigma')$ is a singleton. Let us note it $\{\sigma'_{\mathbb{A}_i}\}$.

  From the two items above we have:

  - *if $tr_{\mathbb{A}_i}$ is defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{\mathbb{A}_i}.act(tr_{\mathbb{A}_i})\}$,*
  - *if $tr_{\mathbb{A}_i}$ is not defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{\mathbb{A}_i}\}$,*

  In both case $Proj_{\mathbb{A}_i}(p,\sigma)$ is a singleton.

  Now let us consider that there exists $l \in FP(\mathbb{A}_1||\mathbb{A}_2)$ such that
  $l \neq p$ and $Proj_{\mathbb{A}_i}(l,\sigma) \neq \emptyset$.

  From Definition 58, $Proj_{\mathbb{A}_i}(l,\sigma)$ is always defined as $Proj_{\mathbb{A}_i}(h,\sigma)$ for some h being a prefix of l unless we have :
  l is of the form $l'.tr'$ where l' is a finite path and tr' is a transition such that $\sigma' \in ttraces(l')$ and $a = act(tr')$.

So let us consider:

l is of the from $l'.tr'$ where l' is a finite path and tr' is a transition such that $\sigma' \in ttraces(l')$ and $a = act(tr')$.

From definition 58 we have:

- *if $tr'_{\mathbb{A}_i}$ is defined we have $Proj_{\mathbb{A}_i}(l,\sigma) = \{\sigma'_{l',\mathbb{A}_i}.act(tr'_{\mathbb{A}_i}) \backslash \sigma'_{l',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(l',\sigma')\}$,*
- *if $tr_{\mathbb{A}_i}$ is not defined we have $Proj_{\mathbb{A}_i}(l,\sigma) = \{\sigma'_{l',\mathbb{A}_i} \backslash \sigma'_{l',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(l',\sigma')\}$,*

Since $\sigma' \in ttraces(l')$ we have $Proj_{\mathbb{A}_i}(l',\sigma') \neq \emptyset$.

Besides $Proj_{\mathbb{A}_i}(\sigma') = \{\sigma'_{\mathbb{A}_i}\}$.

Therefore we conclude $Proj_{\mathbb{A}_i}(l',\sigma') = \{\sigma'_{\mathbb{A}_i}\}$.

Now from the two items above we have:

- *if $tr'_{\mathbb{A}_i}$ is defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{\mathbb{A}_i}.act(tr'_{\mathbb{A}_i})\}$,*
- *if $tr'_{\mathbb{A}_i}$ is not defined we have $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{\mathbb{A}_i}\}$,*

Now we have $act(tr) = act(tr')$ (both equal to a).

Now:

(A): $tr'_{\mathbb{A}_i}$ is not defined if and only if $tr_{\mathbb{A}_i}$ is not defined because it happens in both cases whenever a is of the form $c?v$ or $c!v$ with $c \notin Chan(\mathbb{A}_i)$.

(B): $tr'_{\mathbb{A}_i}$ is defined if and only if $tr_{\mathbb{A}_i}$ is defined: and $act(tr'_{\mathbb{A}_i}) = act(tr_{\mathbb{A}_i})$:

indeed $tr'_{\mathbb{A}_i}$ and $tr_{\mathbb{A}_i}$ are defined if and only if a is of the form $c?v$ or $c!v$ with $c \in Chan(\mathbb{A}_i)$.

In this case both $act(tr'_{\mathbb{A}_i})$ and $act(tr_{\mathbb{A}_i})$ are equal to $c?v$ or to $c!v$ depending on the fact that $c \in Chan_I(\mathbb{A}_i)$ or $c \in Chan_O(\mathbb{A}_i)$.

From (A) and (B) we deduce:

(C): that for any two paths p and l such that $Proj_{\mathbb{A}_i}(p,\sigma)$ and $Proj_{\mathbb{A}_i}(l,\sigma)$ are defined, we have $Proj_{\mathbb{A}_i}(p,\sigma) = Proj_{\mathbb{A}_i}(l,\sigma)$ and both equal to a singleton.

Since $Proj_{\mathbb{A}_i}(\sigma) = \bigcup_{p \in FP(\mathbb{A}_1||\mathbb{A}_2)} Proj_{\mathbb{A}_i}(p,\sigma)$ from (C) we deduce that $Proj_{\mathbb{A}_i}(\sigma)$ is a singleton.

- Let us consider that σ can be decomposed as $\sigma'.d_0 \cdots d_N$ where for all $i \leq N$, $d_i \in M_I$, and σ' is either the empty trace or a trace of the form $\sigma''.b$ where σ'' is a timed trace and b is an action such that $b \notin M_I$. By hypothesis we have $Proj_{\mathbb{A}_i}(\sigma')$ is a singleton.

  Since $\sigma \in TTrace(\mathbb{A}_1||\mathbb{A}_2)$, from Definition 20, we conclude that there exists a path p of the form $p''.tr_0 \cdots tr_M$ where for all $j \leq M$ tr_j is a transition such that $act(tr_j) \in M_I$, and where p'' is either the empty path or a path of the form $p'''.tr''$ where p''' is a finite path and tr'' is a transition such that $act(tr'') \notin M_I$ and such that $\Sigma_{i=0}^{N} d_i \leq \Sigma_{j=0}^{M} act(tr_j)$ and $\sigma' \in ttraces(p'')$.

  In this case, from Definition 58 we have:

  (A): $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{p'',\mathbb{A}_i}.d_0 \cdots d_N \backslash \sigma'_{p'',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(p'',\sigma')\}$.

  Now by hypothesis we have $Proj_{\mathbb{A}_i}(\sigma')$ is a singleton. Let us note $Proj_{\mathbb{A}_i}(\sigma') = \{\sigma'_{\mathbb{A}_i}\}$.

  From (A) we have:

  (B): $Proj_{\mathbb{A}_i}(p,\sigma) = \{\sigma'_{\mathbb{A}_i}.d_0 \cdots d_N\}$.

  Therefore $Proj_{\mathbb{A}_i}(p,\sigma)$ is a singleton.

  Now let us consider that there exists $l \in FP(\mathbb{A}_1||\mathbb{A}_2)$ such that

  $l \neq p$ and $Proj_{\mathbb{A}_i}(l,\sigma) \neq \emptyset$.

  From Definition 58, $Proj_{\mathbb{A}_i}(l,\sigma)$ is always defined as $Proj_{\mathbb{A}_i}(h,\sigma)$ for some h being a prefix of l unless we have :

  l is of the form $l''.tr_0 \cdots tr_{M'}$ where for all $j \leq M'$ tr_j is a transition such that $act(tr_j) \in M_I$, and where l'' is either the empty path or a path of the form $l'''.tr''$

where l''' is a finite path and tr'' is a transition such that $act(tr'') \notin M_I$ and such that $\Sigma_{i=0}^{N} d_i \leq \Sigma_{j=0}^{M'} act(tr_j)$ and $\sigma' \in ttraces(p'')$.
In this case, from Definition 58 we have:
(C): $Proj_{\mathbb{A}_i}(l,\sigma) = \{\sigma'_{l'',\mathbb{A}_i}.d_0 \cdots d_N \backslash \sigma'_{l'',\mathbb{A}_i} \in Proj_{\mathbb{A}_i}(l'',\sigma')\}$.
Now by hypothesis we have $Proj_{\mathbb{A}_i}(\sigma') = \{\sigma'_{\mathbb{A}_i}\}$. From (A) we have:
(D): $Proj_{\mathbb{A}_i}(l,\sigma) = \{\sigma'_{\mathbb{A}_i}.d_0 \cdots d_N\}$.
From (B) and (D) we conclude $Proj_{\mathbb{A}_i}(p,\sigma) = Proj_{\mathbb{A}_i}(l,\sigma)$ and since $Proj_{\mathbb{A}_i}(\sigma) = \bigcup_{p \in FP(\mathbb{A}_1||\mathbb{A}_2)} Proj_{\mathbb{A}_i}(p,\sigma)$ we conclude $Proj_{\mathbb{A}_i}(\sigma)$ is a singleton.

Local output consistency We are interested in particular product of TIOLTS where any reaction of a subsystem after a local trace is specified in the product for any path carrying such a trace. The intuition is that from the point of view of a subsystem its reaction after the same given local trace is consistent with all expected system behaviors requiring the subsystem to follow that particular trace.

Definition 60 (Local output consistency) *Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be two TIOLTS with partitioned actions. The $\mathbb{A}_1||\mathbb{A}_2$* is called locally consistent *if it satisfies the following property :*

$\forall i \in \{1,2\}$, $\forall \sigma \in TTraces(\mathbb{A}_1||\mathbb{A}_2)$, if there exists $r \in O_M(C) \cup M_I$ such that $\sigma_{\mathbb{A}_i}.r$ is in $TTraces(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_i}})$ then for any σ' satisfying $\sigma'_{\mathbb{A}_i} = \sigma_{\mathbb{A}_i}$, we have $\sigma'.r$ is in $TTraces(\mathbb{A}_1||\mathbb{A}_2)$.

A counterexample given in the following shows why such a property does not hold in full generality in the construction of a product.

Counterexample Consider the example in Figure 7.8. Consider the path $p = ((q_0, q'_0), a!5, (q_2, q'_1))$ (of course $p \in FP(\mathbb{A}_1||\mathbb{A}_2)$). We have $ttrace(p_{\mathbb{A}_1}) = a!5$. Let $r = b!5$, we have $a!5.b!5 \in TTraces(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}})$. However, $ttrace(p).r = a!5.b!5$ and $a!5.b!5 \notin TTraces(\mathbb{A}_1||\mathbb{A}_2)$. In fact, $e?5$ must occur in order for $\mathbb{A}_1||\mathbb{A}_2$ to accept successively $a!5$ then $b!5$ which is not the case for the path p.

Timed compositional testing The projection of a system $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_i$ reflects behaviors of $\mathbb{A}_i$ in the context of $\mathbb{A}_1||\mathbb{A}_2$. We show , under some assumptions, that if SUT_1 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$ and SUT_2 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}}$, we have $SUT_1||SUT_2$ *tioco* $\mathbb{A}_1||\mathbb{A}_2$.

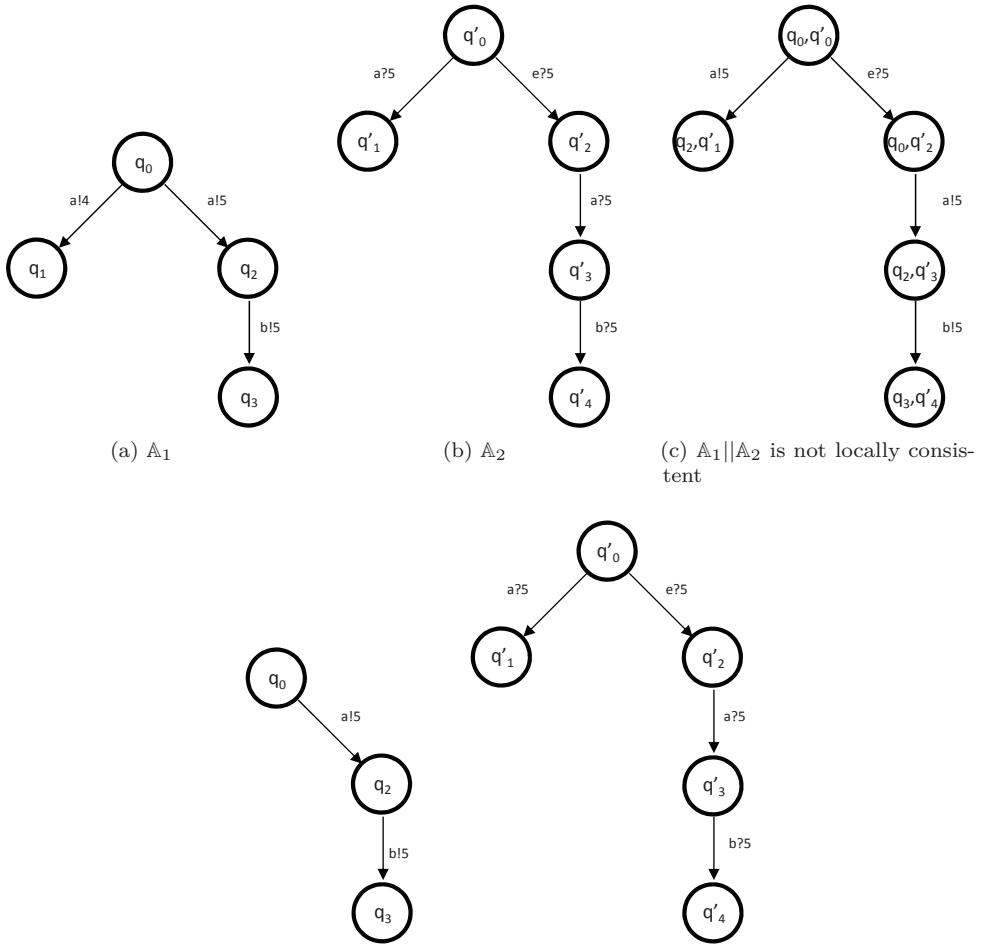
Theorem 1 *Let $\mathbb{A}_1$ and $\mathbb{A}_2$ be two TIOLTS with partitioned actions such that $Chan_I(\mathbb{A}_1) \cap Chan_I(\mathbb{A}_2) = \emptyset$ and $Chan_O(\mathbb{A}_1) \cap Chan_O(\mathbb{A}_2) = \emptyset$. Let us suppose that $\mathbb{A}_1||\mathbb{A}_2$ is locally consistent. Let SUT_1 and SUT_2 be two TIOLTS with partitioned actions such that $Chan(SUT_1) = Chan(\mathbb{A}_1)$, $Chan(SUT_2) = Chan(\mathbb{A}_2)$, $Chan_I(\mathbb{A}_1) = Chan_I(SUT_1)$ and $Chan_O(\mathbb{A}_2) = Chan_O(SUT_2)$. The following property holds:*

$$SUT_1 \text{ tioco } \mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}} \ \wedge \ SUT_2 \text{ tioco } \mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}} \ \Rightarrow \ SUT_1||SUT_2 \text{ tioco } \mathbb{A}_1||\mathbb{A}_2$$

.

Proof 2 *Let us suppose that there exists σ in $TTrace(\mathbb{A}_1||\mathbb{A}_2) \cap TTrace(SUT_1||SUT_2)$ such that there exists r in $M_I \cup O_M(Chan(SUT_1||SUT_2))$ satisfying $\sigma.r$ in $TTrace(SUT_1||SUT_2)$. Let us prove that $\sigma.r$ in $TTrace(\mathbb{A}_1||\mathbb{A}_2)$.*

Since $\mathbb{A}_1$ and $\mathbb{A}_2$ are TIOLTS with partitioned actions form Lemma 1 we have $Proj_{\mathbb{A}_1}(\sigma)$ and $Proj_{\mathbb{A}_2}(\sigma)$ are singleton that we note respectively $\sigma_{\mathbb{A}_1}$ and $\sigma_{\mathbb{A}_2}$.



(a) $\mathbb{A}_1$

(b) $\mathbb{A}_2$

(c) $\mathbb{A}_1||\mathbb{A}_2$ is not locally consistent

(d) $(\mathbb{A}_1||\mathbb{A}_2)_{\mathbb{A}_1}$: the projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_1$

(e) $(\mathbb{A}_1||\mathbb{A}_2)_{\mathbb{A}_2}$: the projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_2$

Figure 7.8: Counterexample to local consistency

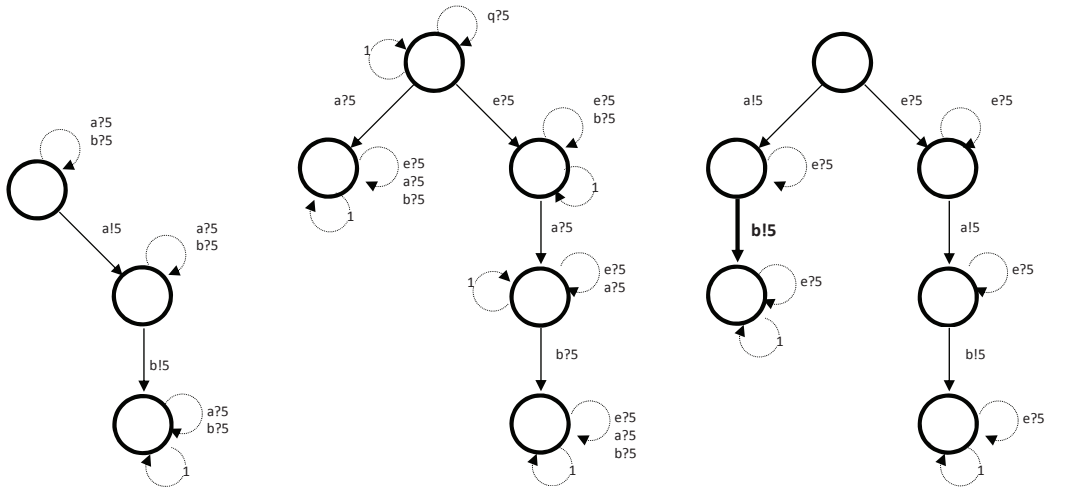
Moreover from Definition 58, we have $Proj_{\mathbb{A}_1}(\sigma)$ *in* $TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}})$ *and* $Proj_{\mathbb{A}_2}(\sigma)$ *in* $TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}})$ *and thus we have* $\sigma_{\mathbb{A}_1}$ *in* $TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}})$ *and* $\sigma_{\mathbb{A}_2}$ *in* $TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}})$.

Since SUT_1 *and* SUT_2 *are also TIOLTS with partitioned actions, we also prove that* $Proj_{SUT_1}(\sigma)$ *and* $Proj_{SUT_2}(\sigma)$ *exist and are unique. We note them respectively* σ_{SUT_1} *and* σ_{SUT_2} *and we have* σ_{SUT_1} *in* $TTrace(SUT_1||SUT_{2_{SUT_1}})$ *and* σ_{SUT_2} *in* $TTrace(SUT_1||SUT_{2_{SUT_2}})$. *Now from Definition 58,* $SUT_1||SUT_{2_{SUT_1}}$ *has the same set of states and the same initial state than* SUT_1 *and* $Trans(SUT_1||SUT_{2_{SUT_1}})$ *is a subset of* $Trans(SUT_1)$ *so we have* $TTrace(SUT_1||SUT_{2_{SUT_1}})$ *is a subset of* $TTrace(SUT_1)$. *Hence we have* σ_{SUT_1} *in* $TTrace(SUT_1)$. *We prove exactly the same way that* σ_{SUT_2} *in* $TTrace(SUT_2)$.

Now since SUT_1 *and* $\mathbb{A}_1$ *are defined over the same set of channels and since* $Chan_I(\mathbb{A}_1) = Chan_I(SUT_1)$ *and* $Chan_O(\mathbb{A}_1) = Chan_O(SUT_1)$, *we have* $\sigma_{\mathbb{A}_1} = \sigma_{SUT_1}$. *For the same reasons this time applied on* SUT_2 *and* $\mathbb{A}_2$, *we have* $\sigma_{\mathbb{A}_2} = \sigma_{SUT_2}$. *Since we have proven* σ_{SUT_1} *in* $TTrace(SUT_1)$ *and* σ_{SUT_2} *in* $TTrace(SUT_2)$, *we have* $\sigma_{\mathbb{A}_1}$ *in* $TTrace(SUT_1)$ *and* $\sigma_{\mathbb{A}_2}$ *in* $TTrace(SUT_2)$.

Now since $\sigma.r$ *is in* $TTrace(SUT_1||SUT_2)$ *there exists* i *in* $\{1,2\}$ *such that* $\sigma_{\mathbb{A}_i}.r$ *in* $TTrace(SUT_i)$ *(if* r *is in* M_I, *it is true for* $i = 1$ *and* $i = 2$). *Since* SUT_i *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_i}}$, *we have* $\sigma_{\mathbb{A}_i}.r$ *is*

in $TTrace(\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_i}})$. *The local consistency of* $\mathbb{A}_1||\mathbb{A}_2$ *allows us to conclude that* $\sigma.r$ *is in* $TTrace(\mathbb{A}_1||\mathbb{A}_2)$ *which ends the proof.*



(a) SUT_1 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$

(b) SUT_2 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}}$

(c) $SUT_1||SUT_2$ *is not conform to* $\mathbb{A}_1||\mathbb{A}_2$: non conform output

Figure 7.9: Counterexample to compositional testing

Discussion Consider the SUT in Figure 7.9. This example suggests that the non-satisfaction of the local output consistency property may cause that the implication of Theorem 1 does not hold anymore. Reconsider the product $\mathbb{A}_1||\mathbb{A}_2$ in Figure 7.8c which is not locally consistent for outputs. We suggest two SUT namely SUT_1 and SUT_2 respectively in Figures 7.9a– 7.9b such that SUT_1 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$ and SUT_2 *tioco* $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}}$. However as shown in Figure 7.9c, the product of SUT that is $SUT_1||SUT_2$ is not conform to $\mathbb{A}_1||\mathbb{A}_2$. This is because after the trace $a!5$, $SUT_1||SUT_2$ reacts with an unspecified output $b!5$ ($a!5.b!5 \notin \mathbb{A}_1||\mathbb{A}_2$).

The Theorem 1 states that we can deduce the conformity of $SUT_1||SUT_2$ to $\mathbb{A}_1||\mathbb{A}_2$ by reasoning on SUT_1 and SUT_2. Let us consider the contrapositive of Theorem 1 (with notations of Theorem 1):

$$SUT_1||SUT_2 \ \cancel{tioco}\ \mathbb{A}_1||\mathbb{A}_2 \ \Rightarrow\ SUT_1 \ \cancel{tioco}\ \mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}} \ \vee\ SUT_2 \ \cancel{tioco}\ \mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_2}}$$

.

This means that any non conformance (in the sense of *tioco*) of the system with respect to a given specification boils down to a non conformance of one of its subsystems with respect to a particular TIOLTS which is the projection of the specification on the concerned subsystem. In the next section, we show how to compute this projection in a symbolic framework and how Theorem 1 can be used to reason about conformance to sequence diagrams.

7.3 Compositional testing from sequence diagrams

In Section 7.2, we have studies in the context of the *tioco* theory how to relate the correctness of a system to a notion of correctness of its components. We have shown how to extract from a TIOLTS denoting a system, new TIOLTS obtained by projection mechanisms (Definition 56) denoting behaviors of subsystems in the context of the system of interest. We have then studied how to relate the correctness of the subsystems to the correctness of the system. The result

obtained is stated in Theorem 1.

In Chapter 6, we have shown how to translate a sequence diagram into TIOSTS and how to symbolically execute them in order to compute their associated behaviors in the form of a symbolic tree. In Section 7.3.1, we show how to adapt projection of TIOLTS to symbolic trees. In Section 7.3.2, we discuss relations between systems under test as defined in Section 7.1.1 and sequence diagram specifications. Since sequence diagram semantics can be computed as a symbolic tree, we show how to define a symbolic tree projection for any subsystem of SUT.

7.3.1 Projection mechanism

Let us show how to compute the projection of the symbolic tree (associated with a sequence diagram) in order to give a symbolic counterpart to the projection of TIOLTS. After the translation phase, a sequence diagram is associated with a set of TIOSTS $\mathbb{C}ol_{sd} = \{\mathbb{G}_1, \cdots, \mathbb{G}_k\}$. A subsystem $\mathcal{C}$ of sd is any subsystem over $\mathbb{C}ol_{sd}$ (see Definition 36). Intuitively, a subsystem of sd is semantically characterized as a composition of all the TIOSTS associated to a group of ports and possibly the TIOSTS characterizing messages between these ports. In Figure 4.1, behaviors of the subsystem corresponding to the component *calc* are depicted by lifelines associated to *intensity* and *speed* (here there are no messages exchanged between them).

The symbolic execution $SE(\mathbb{G}_{sd})$ characterizes in intention the set of all timed traces associated to sd and the behaviors associated with a subsystem $\mathcal{C}$ can be intuitively characterized as the restriction of those timed traces to the interface of $\mathcal{C}$ (consisting of all channel names occurring in $\mathcal{C}$). In order to identify those behaviors, our projection mechanism consists in hiding all the actions of transitions of $SE(\mathbb{G}_{sd})$ which are not defined on the interface of $\mathcal{C}$ (replacing them by the invisible action τ) and by retrieving actions defined in $\mathcal{C}$ for the other transitions.

To reach that goal, we use the transition naming introduced in Definition 35. Any transition st of $SE(\mathbb{G}_{sd})$ is associated with a ground transition $g(st)$, being a transition of $\mathbb{G}_{sd}$. Since $\mathbb{G}_{sd}$ results of a composition, $g(st)$ is associated with a name as a set of names of basic TIOSTS transitions. Let tr be, if defined, the transition of $\mathcal{C}$ such that $name_{\mathcal{C}}(tr) \subseteq name_{\mathbb{G}_{sd}}(g(st))$. We have to identify if the corresponding transition introduce an input or an output to modify $act(st)$ accordingly. Indeed it may happen that $act(st)$ is an output action but corresponds to an input in tr (let us recall that a synchronization between an input and an output results in an output as stated in Definition 35).

Definition 61 (Projection of symbolic trees) *The* projection of $SE(\mathbb{G}_{sd}) = (Init, ST)$ on $\mathcal{C}$ *is the couple* $SE(\mathbb{G}_{sd})_{\mathcal{C}} = (Init, ST_{\mathcal{C}})$ *such that for all* $st \in ST$*:*

- *if there exists* tr *such that* $name_{\mathcal{C}}(tr) \subseteq name_{\mathbb{G}_{sd}}(g(st))$*:*
  - *if* $act(st)$ *is* τ *or an input then we have* $st \in ST_{\mathcal{C}}$*,*
  - *if* $act(tr)$ *is an output then we have* $st \in ST_{\mathcal{C}}$*,*
  - *if* $act(st)$ *is an output* $c!z$ *and* $act(tr)$ *is an input* [1] *then we have*

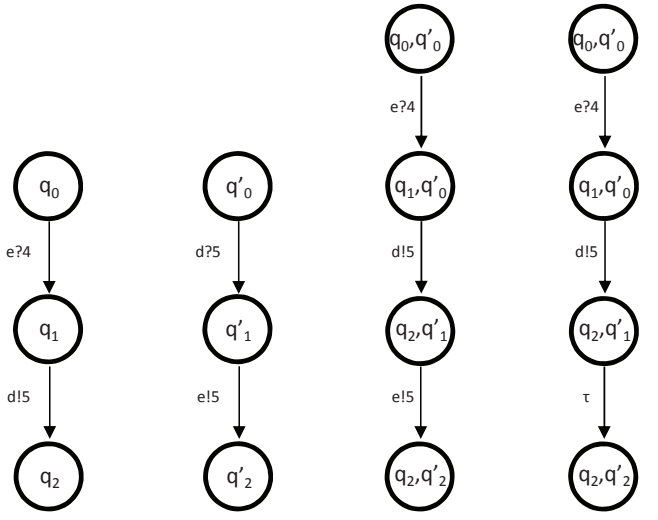
$$(source(st), c?z, target(st)) \in ST_{\mathcal{C}}$$

- *otherwise we have* $(source(st), \tau, target(st)) \in ST_{\mathcal{C}}$*,*

Technically, let us note that $SE(\mathbb{G}_{sd})_{\mathcal{C}}$ characterizes a subset of all the traces of $SE(\mathcal{C})$. The restriction results from the communications with other TIOSTS occurring in the definition of $\mathbb{G}_{sd}$.

[1] $act(tr)$ is then necessarily an input through channel c.

$SE(\mathcal{C})$ is symbolic counterpart to the projection defined in the numerical framework. As compared to Definition 56, Definition 61 is slightly different. Indeed, instead of defining the projection as a subset of $SE(\mathbb{G}_{sd})$ which would be in the spirit of Definition 56, we rather transform the paths of $SE(\mathbb{G}_{sd})$ in order to make disappear all actions that do not concern the subsystem $\mathcal{C}$. In order to illustrate these two different approaches, let us use an example in the frame of the TIOLTS formalism.

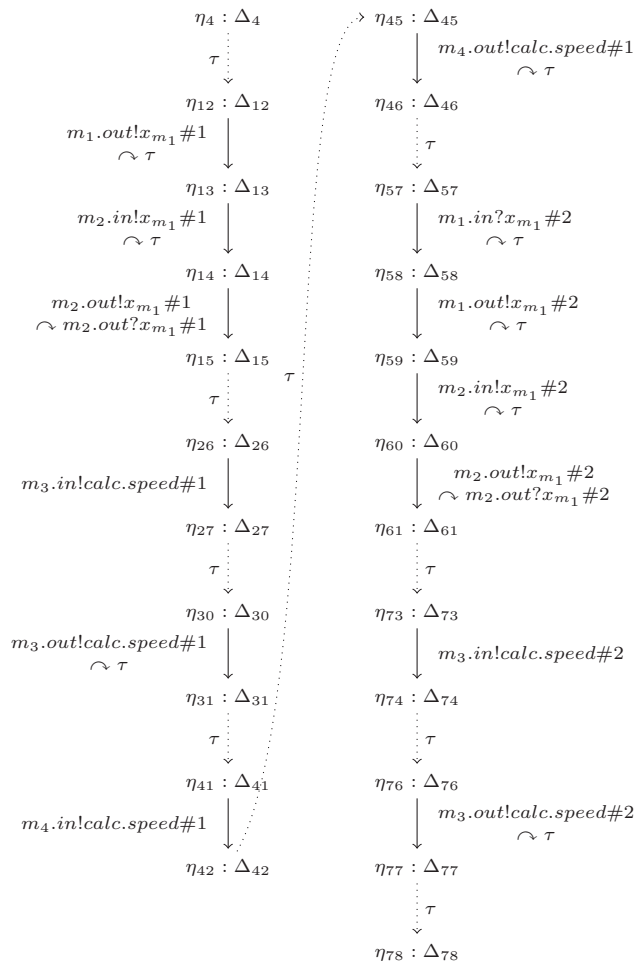


(a) $\mathbb{A}_1$ (b) $\mathbb{A}_2$ (c) $\mathbb{A}_1||\mathbb{A}_2$ (d) Projection of $\mathbb{A}_1||\mathbb{A}_2$ on $\mathbb{A}_1$ in the spirit of Definition 61

Figure 7.10

Figures 7.10a and 7.10b depict respectively two TIOLTS $\mathbb{A}_1$ and $\mathbb{A}_2$. If we define the projection $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$ according to Definition 56, we would observe that $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$ is $\mathbb{A}_1$. Now following the projection mechanism of Definition 61 (of course transposed to TIOLTS), we obtain the TIOLTS of Figure 7.10d. Even though, $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$ and the TIOLTS of Figure 7.10d are not built on the same set of states, it is clear that they are associated with the same timed traces since the sequence of actions occurring in the TIOLTS of Figure 7.10d is exactly the one occurring in $\mathbb{A}_1||\mathbb{A}_{2_{\mathbb{A}_1}}$.

Example 50 *In Figure 7.11a, we illustrate a path p of $SE(\mathbb{G}_{sd})$ where sd is the sequence diagram of Figure 4.1. We show in the same figure, the projection of p on $\mathcal{C} = \overline{\mathbb{G}_{lf_{calc.intensity}}}||\overline{\mathbb{G}_{lf_{calc.speed}}}$ associated with the calculator calc. Transformations of definition 61 are shown by curvy arrows. Instants at which actions occur are denoted by sum of symbolic durations introduced in the course of the symbolic execution ($\Delta_i = \sum_{j=0}^{i} \delta_j$). Similarly, $\Delta_{k\to i}$ means $\sum_{j=k}^{i} \delta_j$. In p after the projection, the reception of the intensity $m_2.out?x_{m1}\#0$ (m_2 in $Msg_{(ctrl.intensity,calc.intensity)}$) occurs at time instant Δ_{15}. The first new speed value emitted by calc after the first reception is not null ($0 \neq calc.speed\#1$) and the second value is equal to the last calculated speed ($calc.speed\#1 = calc.speed\#2$). From constraint $\Delta_{15\to 60} = 0.5$, we deduce that the duration between the first reception of the rain intensity by the calculator ($m_2.out?x_{m1}\#0$ at instant Δ_{15}) and the second reception ($m_2.out?x_{m1}\#1$ at instant Δ_{61}) is at least of $0.5s$.*



(a) Symbolic path and projection

$$\pi_t(\eta_{78}) \begin{cases} \delta_{15} < 0.1 \\ \Delta_{15 \to 31} < 0.5 \\ \Delta_{15 \to 60} = 0.5 \\ \delta_{61} < 0.1 \\ \Delta_{61 \to 77} < 0.5 \end{cases} \qquad \pi_d(\eta_{78}) \begin{cases} 0 \neq calc.speed\#1 \\ calc.speed\#1 = calc.speed\#2 \end{cases}$$

(b) Path conditions

Figure 7.11: Projection of the symbolic tree of the RWC system

7.3.2 Testing architecture

A sequence diagram specifies possible interactions between components ports. The sequence diagram sd_1 in Figure 7.12a depicts three messages m_1 and m_2 exchanged between three ports u and v. sd_1 is a sequence diagram $(\{lf_u, lf_v\}, \{m_1, m_2\})$.

In a sequence diagram, a subsystem is characterized as a group of lifelines (corresponding to ports) with possibly messages exchanged between them. The sequence diagram in Figure 7.12a is decomposed into two groups (delimited by a dotted line) characterizing behaviors of two subsystems : a group containing the lifeline associated with the port u and the message m_2 (see Figure 7.12b); and a group containing the lifeline associated with the port v and the message m_1(see Figure 7.12c). This decomposition maps to the architectural decomposition of the component system into two subsystems as illustrated in Figures 7.12d-7.12e-7.12f. Figure 7.12e depicts the first subsystem which is made of the component owning u and the connector c_2; and Figure 7.12f depicts the first subsystem which is made of the component owning v and the

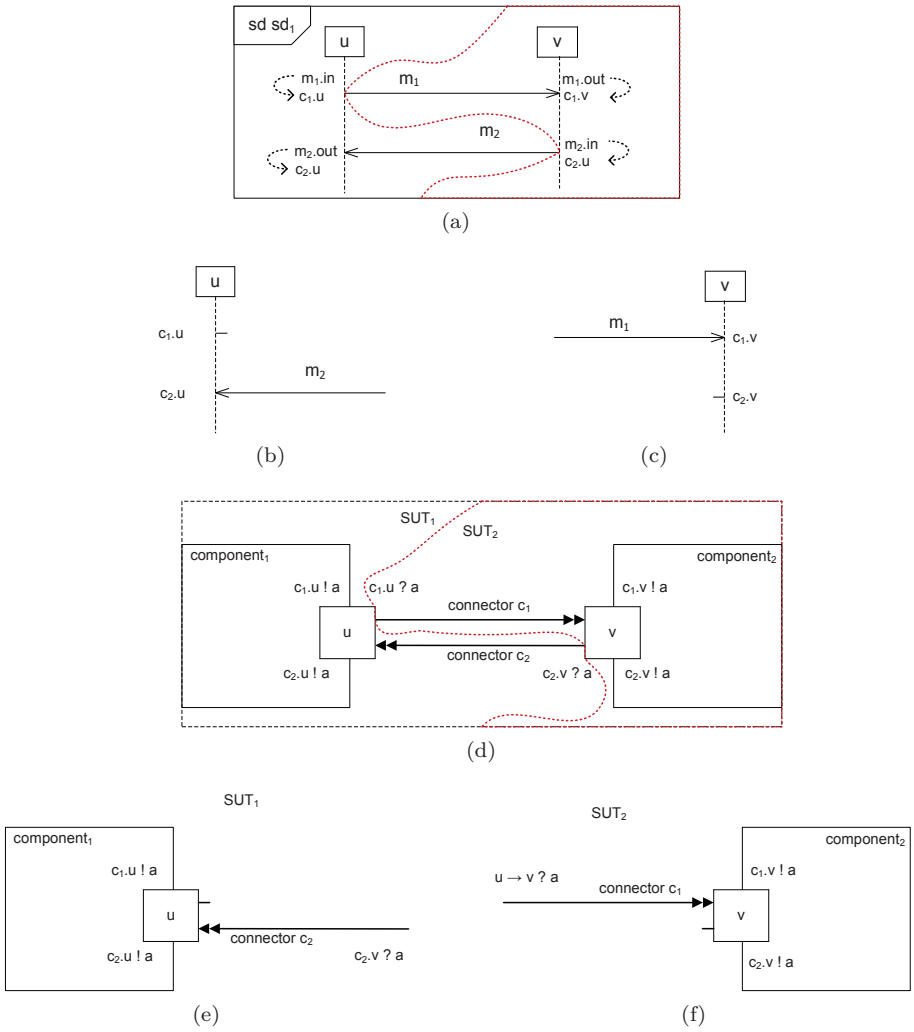


Figure 7.12

connector c_1. Let us structure all connectors occurring in the system in the set $Conn$. Similarly to the messages set $Msg = \cup_{u,v \in P \cup \{e\}} Msg_{(u,v)}$, $Conn$ is of the form $Conn = \cup_{u,v \in P \cup \{e\}} Conn_{(u,v)}$. Regarding the system in Figure 7.12d, $Conn$ equals $Conn_{(u,v)} \cup Conn_{(v,u)}$ where $Conn_{(u,v)} = \{c_1\}$ and $Conn_{(v,u)} = \{c_2\}$.

Now we want to characterize systems under test corresponding to such architectural decompositions. Recall that an SUT is defined over a set of channels, the same as its associated TIOLTS specification. In our symbolic framework, specifications are symbolic executions of TIOSTS built by the translation mechanism. These TIOSTS communicate over channels occurring in their definitions. Until now, we have named channels of the TIOSTS obtained by translation using the convention $m.in$ and $m.out$ where m is the message label. Recall that $m.in$ is used to represent a channel to receive values of the sender lifeline while the convention $m.out$ is used to represent a channel to emit values to the receiver lifeline. This was useful in the translation phase. However, in practice those message labels do have a counterpart in the real system. We need to characterize channels differently rather based on observability issues. When testing the system, values in transit are observed at the port level. We observe a value a at port u. Besides we assume additional observability capabilities of the tester related to the component system architecture: the tester observes if the exchanged value is an incoming or outgoing value; besides, the tester observes which connector conveys exchanged values. For instance, a value a may be observed at port u as being an emission (by the component owning u) through the connector c_1.

a may also be observed at u rather as being a reception coming from the connector c_2. Based on these hypothesises, we introduce a mapping which associates messages in the sequence diagram with connectors (as specified in the system architecture). We define such mapping as a function $f : Msg \rightarrow Conn$ such that if $f(m) = f(m')$ then exists ports u and v such that m and m' are in $Msg_{(u,v)}$. This assumption states that messages exchanged between two ports u and v are mapped onto a connector linking these two ports. Besides, it states that connectors attributed to messages are unidirectional, that is they vehicle values in one direction from a port to another. We have assumed such a property on connectors in order to ensure that channels are partitioned into input channels and output channels as will be examined later in the section.

In order to encode observability, discussed before, in our mathematical models of SUT, we introduce the following format of channels:

Recall that, in order to translate messages of sequence diagrams into TIOSTS, we used channels of the form $m.in$ and $m.out$ for a message m in $Msg_{(u,v)}$. Similarly, in order to denote, in system under test, the source and the target of a given connector c in $Conn_{(u,v)}$ such that $f(m) = c$, we introduce channels names from the point of view of the connector as follows: the channel $c.u$ (same as $f(m).u$) used to receive values from the port u and $c.v$ (same as $f(m_1).v$) used to receive values from the port v.

In this context, a system under test may be seen as a composition of two SUT: $SUT_1||SUT_2$ which is also compliant with the architectural decomposition of the system in terms of components and connectors. SUT_1 and SUT_2 are SUT over channels of the form: $c.p$ and $c.e$ where p is a port and e models the environment.

In the example, SUT_1 and SUT_2 corresponding to the decomposition from left to right in Figure 7.12d are defined respectively over these two sets of channels: $\{c_1.u, c_2.u, c_2.v\}$ and $\{c_1.v, c_2.v, c_1.u\}$. Both sets of channels are partitioned into input/output channels as follows:

$Chan_O(SUT_1)$:	$Chan_I(SUT_2)$:
$c_1.u, c_2.u$	$c_1.u$
$Chan_I(SUT_1)$:	$Chan_O(SUT_2)$:
$c_2.v$	$c_1.v, c_2.v$

Note that $Chan_I(SUT_1) \cap Chan_I(SUT_2) = \emptyset$ and $Chan_O(SUT_1) \cap Chan_O(SUT_2) = \emptyset$.

After having named channels in SUT based on observability hypothesis relating to the component system architecture. We now show how to transform the specification as symbolic tree in order to take into consideration changes in the channels naming.

Recall that, besides the technical artifacts such as the channel *start* and channels of the form $wait_{o_1.o_2}$ (see Chapter 6), the channel names occurring in the symbolic execution $SE(\mathbb{G})_\delta$ are of the form $m.in$ and $m.out$ as it is the case in the translation of the sequence diagram $\mathbb{G}_{sd}$. The renaming of channels in $SE(\mathbb{G})$ is defined as follows:

- τ is left unchanged,
- all actions on channels of the form *start* and $wait_{o_1.o_2}$ are replaced by τ,
- all actions of the form $m.in!a$ (respectively $m.in?a$) with m in $Msg_{(u,v)}$ are renamed $f(m).u!a$ (respectively $f(m).u?a$),
- all actions of the form $m.out!a$ (respectively $m.out?a$) with m in $Msg_{(u,v)}$ are renamed $f(m).v!a$ (respectively $f(m).v?a$).

In this naming, we observe either from the point of view of the components or from the point of view of the connectors:

- $c.u!a$ can be understood as an output from a component point of view, c being a connector conveying values from u to v. The component owning u sends a to the port v through c,
- $c.u?a$ is a reception from the connector point of view, the value a is to be sent to v,
- $c.v!a$ can be understood as an output from the point of view of a connector. The connector brings the value a to the port v,
- $c.v?a$ corresponds to a reception from a component point view. The component receives at its port v the value a computed by the component owning v.

In the sequel, we note $Ren(SE(\mathbb{G}))$ the symbolic execution $SE(\mathbb{G})$ after the renaming.

Example 51 *Let us apply the renaming on the symbolic execution of the RWC system. For that purpose, we consider given a mapping f of messages on connectors. The transformation is illustrated in Figure 7.13 by curvy arrows. For example, $m_1.in?x_{m_1}\#2 \curvearrowright f(m_1).e?x_{m_1}\#2$ denotes the renaming of the action $m_1.in?x_{m_1}\#2$ to $f(m_1).e?x_{m_1}\#2$ where $f(m_1)$ is in $Conn_{(e,ctrl.intensity)}$ (note that m_1 is in $Msg_{(e,ctrl.intensity)}$).*

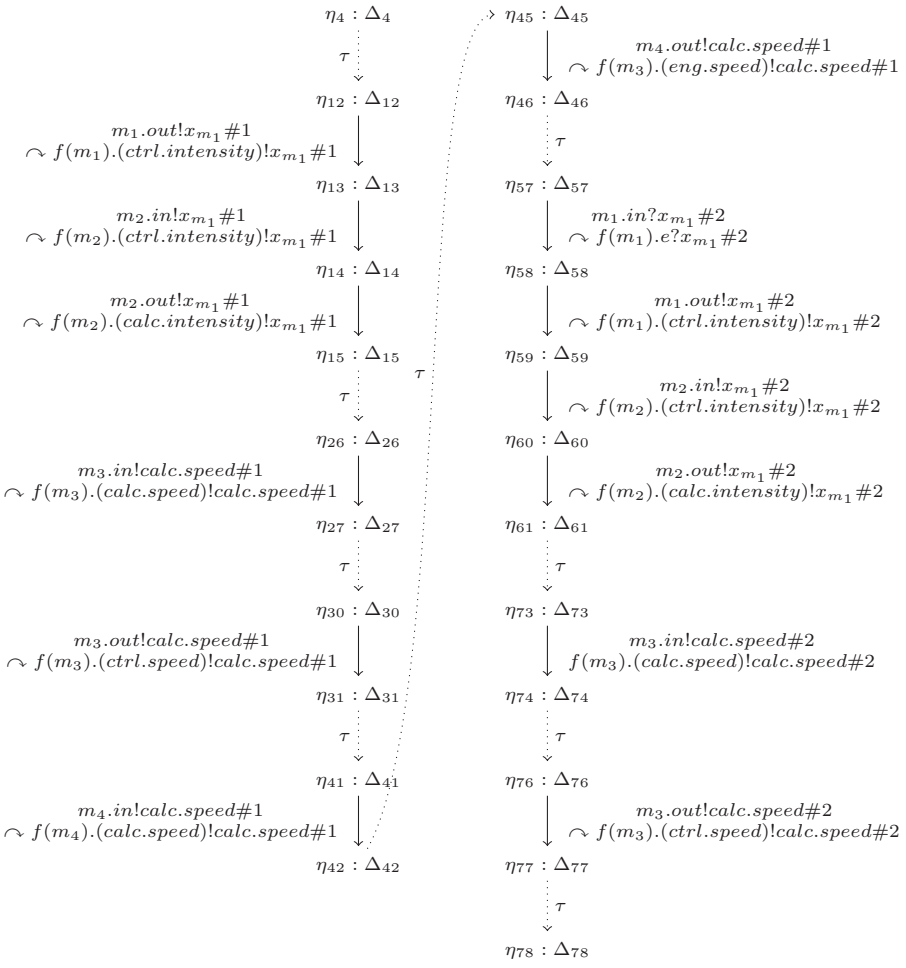


Figure 7.13: Channels renaming in the symbolic tree of the RWC system

In our symbolic framework, we need to give a counterpart to *tioco* which defines the conformance of SUT with respect to a TIOLTS. Therefore, we define in the following the conformance of SUT with respect to the symbolic trees being either symbolic executions or projection of symbolic executions. We abstract those two possible cases as the so-called *Symbolic Trees over a signature* (F, C) where F is a set of fresh variables and C is a set of channels. Symbolic Trees over a signature (F, C) are couples $(Init, R)$ where $Init \in \mathcal{S}_{sat}$ satisfies the same properties as in Definition 46 and $R \in \mathcal{S}_{sat} \times Act(\Sigma_F) \times \mathcal{S}_{sat}$.

Definition 62 (tioco) *Let $STre$ be a Symbolic Tree over a signature (F, C) and SUT be an Implementation Under Test over C. SUT* conforms to *$STre$, denoted SUT tioco $STre$, if and only if:*
$\forall \sigma \in TTraces(STre) \cap TTraces(SUT)$, $\forall r$ of the form $c!a$ or belonging to M_I we have:

$$\sigma.r \in SUT \Rightarrow \sigma.r \in TTraces(STre)$$

We want to make use of Theorem 1, so that testing a system with respect to a sequence diagram, amounts to testing subsystems with respect to projections. For that, we need the local consistency property to be satisfied. As defined for a product of TIOLTS, we need to check that every local trace (which belongs to the projections of the system tree on the subsystems) is consistent with all behaviors characterized by the symbolic tree of the system. Usually, symbolic trees are big structures (may be infinite) and encodes too many concrete (local) traces. In practice, we propose to test this property. For that purpose, we use the local traces that we have built by interacting with a SUT associated with the subsystem as we discuss it in the following.

Let us denote $STre$ the symbolic execution $SE(\mathbb{G}_{sd})_\delta$ after the renaming. Consider a SUT $SUT_\mathcal{C}$ corresponding to the subsystem $\mathcal{C}$ of $\mathbb{C}ol_{sd}$.

Let $\sigma_\mathcal{C}$ be a trace built by interacting with a $SUT_\mathcal{C}$ such that $\sigma_\mathcal{C}$ is in $STre_\mathcal{C}$ (i.e. $\sigma_\mathcal{C}$ is in $SUT_\mathcal{C} \cap STre_\mathcal{C}$).

The property of local consistency does not hold:

If there exists a prefix $\sigma'_\mathcal{C}$ of $\sigma_\mathcal{C}$ of the form $\sigma''_\mathcal{C}.act$ where act is an output or a duration such that there exists a trace σ'' which belongs to $STre$ (in the sense of Definition 53) such that $proj(\sigma'', \mathcal{C})$ is $\sigma''_\mathcal{C}$ and $\sigma''.act$ does not belong to $STre$.

σ'' belongs to a path (or several paths) of $STre$ (in the sense of Definition 52) and according to our definition of the projection (Definition 61): for any such path p there exists a path $p_\mathcal{C}$ in $STre_\mathcal{C}$. Obviously $\sigma''_\mathcal{C}$ belongs to $p_\mathcal{C}$.

From Definition 61, we have $\sigma''.act$ belongs to p if and only if $\sigma''_\mathcal{C}.act$ belongs to $p_\mathcal{C}$.

If for all such p, we have either $\sigma''_\mathcal{C}.act$ belongs to $p_\mathcal{C}$ or there exists a symbolic transition st such that $\sigma''_\mathcal{C}.act$ belongs to $p_\mathcal{C}.st$ (see Definition 61), we can deduce that for all σ'' such that $proj(\sigma'', \mathcal{C})$ is $\sigma''_\mathcal{C}$ and for all p such that σ'' belongs to p, we have either $\sigma''.act$ belongs to p or $\sigma''.act$ belongs to $p.st$.

Therefore, in order to test the local consistency regarding $\sigma''_\mathcal{C}$, we can consider working only on the projection $STre_\mathcal{C}$: we have to test that act is consistent with all paths of $STre_\mathcal{C}$ to which belongs $\sigma''_\mathcal{C}$. Figure 7.14 depicts the algorithm which tests if the local output consistency holds on $STre_\mathcal{C}$, regarding any trace $\sigma_\mathcal{C}$ of $SUT_\mathcal{C}$ as discussed before. If $\sigma_\mathcal{C}$ permits to prove that the local output consistency does not hold, the algorithm returns $FAIL$, otherwise it returns $PASS$.

Example 52 (Counterexample to local output consistency in a symbolic framework)

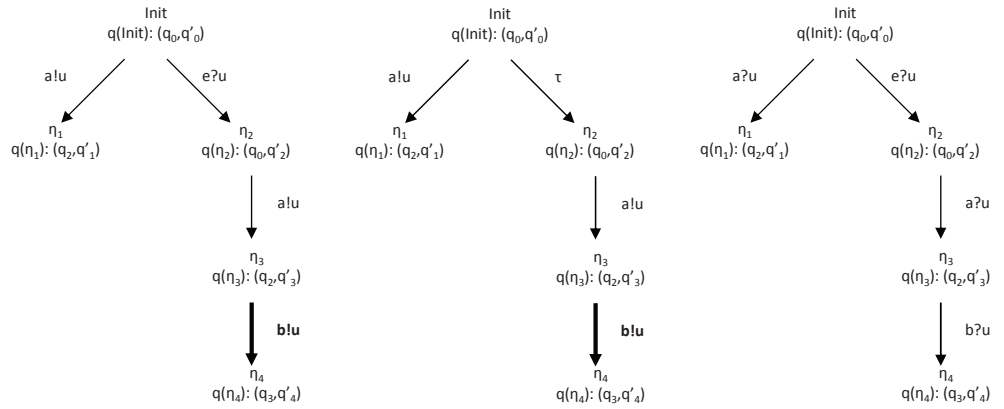
Algorithm 1:

Data: $\sigma_{\mathcal{C}}$ is a timed trace of $SUT_{\mathcal{C}}$, $STre_{\mathcal{C}}$ is the projection of $Ren(SE(\mathbb{G}_{sd})_{\delta})$ on $\mathcal{C}$

```
1 begin
2     for prefix σ''_C.act of σ_C where act is an output or a duration do
3         for path p in STre_C s.t. σ''_C belongs to p do
4             if σ''_C.act does not belong to p ∨
5                   there does not exist a transition st of STre_C s.t. σ''_C belongs to p.st then
6                 return FAIL
7     return PASS
```

Figure 7.14: Testing local output consistency

Consider the example in Figure 7.15. Consider first the trace $\sigma_{\mathcal{C}} = a!5.b!5$ *of* $SUT_{\mathcal{C}}$ *and* $STre_{\mathcal{C}}$*. Here* $\sigma_{\mathcal{C}} = \sigma''_{\mathcal{C}}.act$ *where* $\sigma''_{\mathcal{C}} = a!5$ *and* $act = b!5$*. In the example, we have two paths* $p^1_{\mathcal{C}} = (Init, a!u, \eta_1)$ *and* $p^2_{\mathcal{C}} = (Init, \tau, \eta_2).(\eta_2, a!5, \eta_3)$ *such that* $\sigma''_{\mathcal{C}}$ *belongs to both of them. We have* $\sigma''_{\mathcal{C}}.b!5$ *belongs to* $p^2_{\mathcal{C}}$*. However, for* $p^1_{\mathcal{C}}$ *there does not exist a symbolic transition st of* $STre_{\mathcal{C}}$ *such that* $\sigma''_{\mathcal{C}}.b!5$ *belongs to* $p^1_{\mathcal{C}}.st$ *which violates the local consistency.*



(a) $STre$ is not locally consistent

(b) $STre_{\mathcal{C}}$: the projection of $Ren(SE(\mathbb{G}_{sd})_{\delta})$ on $\mathcal{C}$

(c) $STre_{\mathcal{C}'}$: the projection of $Ren(SE(\mathbb{G}_{sd})_{\delta})$ on $\mathcal{C}'$

Figure 7.15: Counterexample to local consistency in a symbolic context

Discussion

We have reformulated *tioco* with respect to symbolic trees. This permits to use the symbolic execution as a reference for testing SUT. Since we have defined how to symbolically execute sequence diagrams, this allows us to use symbolic executions of sequence diagrams as references for testing. We have discussed how to relate sequence diagrams to the architecture of systems under test. We have identified, using projection mechanisms, symbolic trees denoting behaviors of any subsystem (i.e. group of selected components or connectors) in the context of the whole system. We relate conformance of subsystems (to their projected trees) and conformance of the whole system (to the symbolic tree associated with its sequence diagram) thanks to Theorem 1. In order to be applicable, Theorem 1 requires the reference specification product (formulated as a TIOLTS) to be "local output consistent". Transposed to symbolic trees, it means that for any $\sigma_{\mathcal{C}}$ built by a tester while interacting with a subsystem, if $\sigma_{\mathcal{C}}$ belongs to the projected symbolic tree associated with the subsystem, and if an output or duration r can occur according to the projected symbolic tree, then for any trace σ of the system symbolic tree whose projection is $\sigma_{\mathcal{C}}$, we have $\sigma.r$ is a trace of the symbolic tree of the system. Instead of proving this property on

the whole symbolic tree of the sequence diagram, we propose to prove it just for traces that are built concretely during a subsystem testing process. Generally, deriving the proof with respect to the whole symbolic tree of the sequence diagram can be impossible because it is often an infinite structure. On the contrary, when testing a subsystem, we only build a finite number of finite traces and it is sufficient to prove that "local output consistency" is not "broken" by these traces, which is done by the algorithm of Figure 7.15 (which tests that the "local output consistency" property holds) by proving that traces built when testing the subsystems do not break the property. This result is a first step towards the definition of a modular testing process, in which a tester identify a partition of the system into smaller ones, for which it is easy to define a concrete testing architecture. Indeed, the result permits (as long as the local output consistency property holds) to ensure that any fault, occurring at the system level, could be observed at the level of one of the subsystems of the partition, as long as the used testing algorithm uses the subsystem projected trees as reference specifications, and as long as the testing algorithm is complete. A direct extension to this work will be to adapt the algorithm in [31] for that purpose.

Chapter 8

Related Work

Contents

The Unified Modeling Language (UML) includes many types of diagrams to capture the system structure and dynamics, among the latter are sequence diagrams. Sequence diagrams are visual formalisms for scenario-based specifications and is a standard of the OMG consortium (Object Management Group, for modeling object-oriented systems) [74]. The predecessors of sequence diagrams are SDL (Specification and Description Language) [49] and MSC (Message Sequence Chart) [51]. MSC is an ITU-T standard (ITU Telecommunication Standardization Sector) [50] whose first version MSC-92 emerged in turn from SDL. Then successive revisions gave MSC-2000. The latter was one of the basis to make early versions of UML 2.0 sequence diagrams. There are few technical differences [45] between the MSC and sequence diagrams in their modeling power (e.g. sequence diagrams exhibit few more combining operators such as strict sequencing, and there are differences in the definition of timing properties: in MSC timers may be used). We use rather the UML technology which is closely related to model-driven development, executable models, code generation and round-trip engineering. However, situating the works related to MSC is in the scope of those related to sequence diagrams.

The related approaches that we have selected focus on three main axes of interest:

- **The synthesis of state-based models from sequence diagrams and MSC.** The definition of the target state-based formalism depends on the kind of analysis to be pursued on scenarios (e.g. model driven development of systems by code generation, model checking). In this thesis, our concern is symbolic execution, and in particular how it can be used for testing.

- **The use of sequence diagrams and MSC in testing.** We discuss variant uses of scenario-based models in the testing process: some works use scenarios as a modeling notation to describe tests, some other deal with tests generation from scenarios with different level of formality.

- **The exploitation of sequence diagrams and MSC by projection**. Scenarios represent the dynamics of the intended cooperation between the entities of the system. Hence, by projection, unitary behaviors can be derived from the scenarios.

Besides, we have identified some transversal comparison criteria relevant to our work :

- *Structuring scenarios with combining operators.* We discuss which approaches support operators dedicated to the composition of scenarios such as *sequence*, *iteration* and *choice*. Note that in sequence diagrams, it is possible to directly draw a nested operator in another operator region. In the MSC paradigm, HMSC (High level message sequence charts) [68] are rather used for that purpose. HMSC are flow graphs with control nodes (for operators conditions) and the other nodes are just references to basic MSC [1] defined elsewhere, in another diagram. MSC may also be structured by modalities called LSC (Live Sequence Chart) [24]: LSC adds another semantics to MSC, they can express mandatory behavior, and not only possible behavior.

- *Annotation of scenarios with timing features.* We pay attention to whether considered scenarios are constrained by timing guards. Also, we look at the expressive power of these guards to capture relations between execution instants.

- *Symbolic denotation of scenarios.* The usual practice is to use concrete data values (in finite domains) while treating time symbolically. This brings about classically problems of state explosion. Seldom approaches handle both data and time requirements symbolically in the scenarios.

8.1 Synthesis of state-based models from scenarios

We have synthesized TIOSTS automata from timed sequence diagrams. There is a huge literature on synthesizing state-based models from scenarios. We discuss only some of them in this section which are either more known or closely related to our work.

8.1.1 Basic scenarios

Early works (e.g. [56, 43, 44, 89, 4]) along this axis addressed generation of automata from basic MSC without time annotations and for different purposes than ours (symbolic execution). In order to enable their integration in the development process, authors in [56] translate MSC into statecharts [42] integrated later in UML and for which verification and code generation tools were developed. In the same spirit, the approach in [43, 44] considers generation from LSC. Authors in [89] focus on the generation of structured statecharts from a collection of basic sequence diagrams. Hierarchy and alternatives are implicit behaviors and are deduced automatically from merging all the behaviors of the sequence diagrams in that collection. The work in [4] translates a collection of basic MSC into a set of concurrent automata (one per system entity) with buffers. By taking into account the communication architecture in the translation, this work is the closest to ours in that sense, and differs only in the kind of conducted analysis. In fact, the product of the automata may generate behaviors not present in the entry scenarios (e.g. Each entity of the system chooses to start a different scenario, not being aware of others entities choices). These behaviors are called implied scenarios. Their analysis aims at the inference of (undesirable) implied behavior and the construction of correct deadlock free models.

[1] We say basic MSC (or basic sequence diagram) when it does not include combining operators.

8.1.2 Combined scenarios

Structured scenarios have been considered later in the literature (e.g. in [83, 84, 53, 7]). In these approaches the timing constraints are not yet supported.

The work in [83, 84] defines a methodology for incremental elaboration of scenario based specifications structured as HMSC. Authors focus on the implied scenarios detection. A specification is a set of scenarios, their composition using HMSC operators (similar to sequence diagram operators *alt*, *loop*, etc.) may not provide the required system behavior. Implied scenarios may appear as a result of unexpected components interactions. In order to accommodate implied scenario acceptance and rejection, the approach synthesizes state-based models as labeled transition systems (LTS) for both intended and undesirable behaviors. The communication mechanism is encoded by the parallel composition of LTS synchronizing on the concrete exchanged values (message labels). Based on the generated state-based models, authors present a technique to provide feedback on the existence of implied scenarios.

In [53], authors describe a translation of a sequence diagram containing combining operators into an automaton for model checking. This automaton is used as an observer process in the SPIN tool [48] to check whether a sequence diagram can be satisfied by a given set of UML state machines modeling entities of the system. The subset of operators handled is quite complete but the work avoids some problems linked to concurrency, like in the translation of the choice operator.

The work presented in [7] is close to ours: authors infer the communication mechanism from structured sequence diagrams. This communication mechanism is meant to encode coordination protocols implicitly. *Constraint automata* are generated for connectors from scenarios. A constraint automaton describes the desired input/output behavior at the ports of the components. In fact, constraint automata encode constraints on data assigned to ports, and one port is defined per entity of the system (a constraint tells that the data has to be equal to the concrete exchanged value). The constraint automata generated can then be used to generate Reo circuits [8], which provides the glue code (synchronous, asynchronous, broadcast communication, etc.). In addition, we consider scenarios with data and timing constraints denoted symbolically in the translation.

8.1.3 Time annotations and symbolic analysis

Timing constraints have been considered in more recent works [61, 92, 88]. Synthesis of a network of *timed automata* [3] from a set of LSC charts is given in [61]. LSC charts have been extended with timing features: Charts are equipped with clock variables and thus may be annotated in the timed automata style with clock constraints and assignments as clock resets. The real-time model checker UPPAAL [14] is then used to check the consistency of the set of charts [2]. Note that a timed automaton is associated with symbolic structures [3] to store symbolic representation of the states, for timed analysis purposes. A similar approach was adopted in [92] but with translation to *time petri nets* [69]. Here, the entry scenarios are sequence diagrams with a variant form of timing constraints with the MARTE::VSL language. Also, analysis of time petri nets that the authors use relies on the construction of a data structure [4] abstracting the state space to classes with the same time firing conditions. Clearly, our work is similar to these works in sense of making use of symbolic techniques to represent the state space in an abstract manner for

[2] Consistent means that there does not exist an infinite message sequence, i.e. sequence of concrete values exchanged, that satisfies all the universal LSC.

[3] e.g. The difference bound matrices (DBMs) are data structures to describe zones (a zone correspond to a set of constraints).

[4] The so-called States Classes Graph (SCG) [16] are used, among other techniques, in the state reachability analysis of time petri nets.

time variables, but data variables encoding constraints on exchanged values are also symbolically treated in our framework rather than enumerated. In addition, we support another form of constraints with MARTE::VSL language which allows us to constrain time instants. However, these works address verification problems while we focus on symbolic execution-based techniques for black box testing.

An overview of the selected approaches along this axis, i.e. synthesis of state-based models from sequence diagrams and MSC, is given in Table 8.1 including a comparison with our approach.

Approach	*Notation*	*Combining operators*	*Time constraints*	*Target state -based model*	*Symbolic interpretation*	*Use*	*Tool support*
Kruger et al. 1998 [56]	MSC	no	no	statechart	no	*code generation*	no
Harel et al. 2000 [43]	LSC	no	yes	statechart	no	*code generation*	yes (Play-engine)
Whittle et al. 2000 [89]	SD	no	no	statechart	no	*code generation*	yes
Alur et al. 2003 [4]	SD	no	no	concurrent automata	no	*verification*	yes
Uchitel et al. 2004 [83, 84]	MSC	HMSC	no	LTS	no	*verification*	yes
Knapp et al. 2006 [53]	SD	yes	no	interaction automata	no	*model-model-checking*	yes (+SPIN)
Arbab et al. 2008 [7]	SD	yes	no	constraint automata	no	*glue code generation*	yes (+Reo)
Larsen et al. 2010 [61]	LSC	no	yes	timed automata	time variables	*model-model-checking*	yes (+UPAAL)
Zhu et al. 2010 [92]	SD	yes	yes (MARTE)	time Petri net	time variables	*verification*	yes (+ROMEO)
our approach	SD	yes	yes (MARTE)	TIOSTS +TIOLTS	time+data variables	*symbolic execution*	yes (+Diversity)

Abbreviations:
MSC: Message Sequence Chart
HMSC: High-Level MSC
LSC: Live Sequence Chart
SD: Sequence Diagram
MARTE: Modeling and Analysis of Real-Time and Embedded systems
LTS: Labeled Transition System
TIOLTS: Timed Input Output Labeled Transition System
TIOSTS: Timed Input Output Symbolic Transition System

Table 8.1: Comparing our approach to other synthesis of state-based models from scenarios

8.2 Scenario-based testing

We use sequence diagrams as a reference specification in a conformance testing framework. In this section, we give an overview of the usage of sequence diagrams in testing in general.

8.2.1 Earlier work

Testing based on UML scenario models is the subject of a long thread of works [90, 19, 13, 35, 9, 2, 71]. These approaches use/derive sequence diagrams as test cases. The strength of such approaches is that they enable the industrial integration of testing in the practices and tools of the designers of UML models. These works tackle the test generation problem from a more practical perspective. In [19], test cases are derived from use cases, which are structured and detailed using UML activity and sequence diagrams. Sequence diagrams are used in [35] and in [90] for describing test specifications, which are extended with method parameters, and return values for method calls for actual testing. The work in [13] uses UML sequence diagrams for describing system use cases. Corresponding to each use case, a set of test cases for testing such use case are then derived. The UBET tool [15] supports test generation from a HMSC model, in which test generation is primarily driven by the edge-coverage criteria in a HMSC. In [59], the play-engine tool for Live Sequence Charts (LSCs), which are an extension of MSCs, has been extended to support testing of scenario-based requirements.

8.2.2 Concrete test generation from scenarios

The work in [64, 63] gives an algorithm to derive tests from UML 2.0 sequence diagram. The derived tests are themselves sequence diagrams. Authors define operational semantics [65, 64, 63]

for their entry sequence diagrams which applies as well to the derived ones. The test derivation algorithm is an adaptation of the *ioco* based testing algorithm in [82]. The subset of the sequence diagrams does include time features in the semantics (defined in [63]), however it does not include them in the test derivation. They consider besides the repetition, the choice and the strict sequencing operators that we handle in our approach, assertion and negative operators (*neg* and *assert*) for forbidden and required behaviors. In fact, UML states that behaviors are defined as traces, and traces may be valid or invalid. The *assert* operator allows one to specify the only valid behaviors that can occur starting from the moment the *assert* region is entered. The *neg* operator specifies a behavior which is considered to be invalid, and implies that all other behaviors happening since the *neg* region is entered are valid. In our approach, all the specified behaviors by a sequence diagram are considered to be valid, as if we had enclosed such behaviors in an *assert* region, except that we allow the underspecification of messages coming from the environment. The execution system that the authors defined to capture the semantics of a sequence diagram is interpreted as a labeled transition system (LTS). This allowed them to define test derivation in the fashion of [82]. The intuition is that the execution models attributes a meta property *mode* to each executed action (emission/reception of a message). For instance, when entering an *neg* operator region (captured by a special label on an unobservable action τ_{neg}), the execution system sets the *mode* value of next actions to invalid. From the semantics defined for the sequence diagram, the proposed algorithm generates a sequence diagram of test events which ends with a verdict *pass*, $fail$ or *inconc* (for inconclusive i.e. behaviors which does not allow to conclude on the conformance). The algorithm works as follows: (*i*) The diagram is supplied with an input (external receptions from the environment), and the test continues recursively with all the states which are reachable after that input. (*ii*) The diagram is checked: if the observed output is a specified response then the test continues recursively, otherwise the test sequence terminates with a verdict. In fact, if the unspecified output was observed while the execution mode is valid (in an assert region) then the verdict is $fail$, otherwise the verdict is *inconc* (typically in *neg* region). (*iii*) When the diagram accepts no stimulation the test terminates with a verdict. It is $fail$ if the execution mode is invalid (in a neg region), otherwise it is an *accept*.

Authors in [75] use sequence diagrams in the testing activities. They presented an approach to generate test cases (expressed as UML scenarios) from state-based UML models guided by test objectives (also expressed as scenarios). The UML state-based models are transformed into IOLTS (Input Output Labeled Transition System). Then, the test case derivation is based on *ioco*. Note that the derived test cases may be structured (e.g. using the choice operator).

Test generation from MSC was studied extensively in [10, 11]. The approach denote the semantics of a HMSC as a partial order graph in which each node corresponds to communication actions. The graph is used during test generation. Besides, the test generation takes as input the subsystem in the MSC format which represents the SUT (System Under Test). Generated tests (as scripts) have the form of a tree (data) structure where nodes are communication actions or verdicts. The authors distinguish three test criteria. (*a*) Trace Testing: in which each trace through an MSC is created as a separated test script. This is the simplest test strategy. (*b*) Branch Testing: in which each test represents a separate path through an MSC that contains branching such as alternatives. (*c*) Completion Testing: The messages sent by a test script will be dependent upon how the SUT behaves at execution time, but will be fixed within the branch taken. Authors in [10] discuss concurrent test generation issues: The SUT can be stimulated/observed by a set of parallel test components (each running their own test script independently). The test components can be autonomous processes running over a distributed system and can synchronize their behavior by dedicated communication channels carrying coordinating messages. The final verdict is computed automatically from the individual verdicts by a master test component. In the test architecture, coordination channels between the testers have been introduced. This can allow information to be shared between the testers and increase the observational power of testing. Authors assume

that all coordinating messages can be implemented with negligible latency compared to ordinary messages that communicate with the SUT. Authors were interested also in timed test generation in [10]. They use a basic mechanism that consists in constraining two communication actions by a time interval that specifies a window of time in which they occur relative one to another. Tests are generated for constraints expressing an exact waiting delay as follows: a timer is started after the referenced action is executed, and the constrained action is suspended when the timer has expired. For intervals, two timers are considered. Depending on their timeouts, verdicts are emitted. In order to prevent false passes the timers are pessimistic: They restrict more the interval. For example, the upper bound timer is started just before the reference event (necessarily a send) and the lower bound timer begins just after. And so, no pass is emitted when the constrained event occurs just after the upper bound is expired, or a fail just before lower bound expires. Note that when two constraints defined in two alternative regions with a reference event outside these regions, two timers are started because the tester does not know in advance which alternative branch will be chosen: for example, if the SUT reacts and makes the execution goes to the region different from the one being tested, then both timers are canceled and the verdict is inconclusive. Clearly, this work gives more technical solutions to generate and execute tests and less links with the testing theory. Authors were interested in how to test distributed systems. We are rather interested in simplifying the testing thanks to our result on the compositionality.

8.2.3 Symbolic test generation from scenarios

In [28, 27], symbolic techniques are used to generate test inputs from information contained in a class diagram and a sequence diagram. Transformation rules are defined to obtain a directed graph VGA (Variable Assignment Graph) from these diagrams: It describes the effect of the message exchanges on the variables of the system. Paths in the VGA encode all the possible executions of the system which may be huge or infinite due to unbounded loops in the sequence diagram. The authors define coverage criteria for sequence diagrams to select relevant paths: all Messages coverage, each message must be sent at least once; all conditions coverage, each condition in each decision must be evaluated to both $true$ and $false$; and all message paths coverage, each message path must be traversed at least once. The last criterion cannot be satisfied when the number of paths is infinite. For each selected path, system variables are treated symbolically, that is, a new fresh variable is introduced when a variable is redefined. Test inputs are determined by solving the system of path conditions. In order to evaluate the relevance of their generated test inputs, the authors introduce faults in the specification as a sequence diagram (among the most frequent in the design process, e.g. a missing alternative or a modified alternative). In such situations, the condition coverage criterion is demonstrated to be efficient in fault detection. Notice that the test inputs were used to test the specification model and it is not clear how these inputs help to test the (model of the) implementation with respect to the specification.

Testing based on symbolic denotation of scenarios has been considered recently in other works [79, 37, 78]. While in the case of a MSC a lifeline can represent only one concrete process, the approach extends MSC with a *symbolic lifeline* which represents at any point of the interaction some/all processes from a collection: a guard may be associated with a communication action for selecting which subset of processes from that collection actually perform the action. [79] describes a methodology for testing systems with large number of behaviorally similar processes, i.e. process classes, based on SMSC (Symbolic message Sequence Chart). The testing framework takes as input besides the SMSC specification, a user test purpose, also as a SMSC, which represents a particular behavior of the specification to be tested. First, an abstract test case, i.e. denoted symbolically, is generated in the form of a SMSC, satisfying the test purpose. Concrete test cases as MSC, where a set of concrete lifelines are instantiated from each symbolic lifeline, are not generated directly from the abstract test case. Rather, a minimal set of test case templates

are derived from the abstract test case. A template is an intermediate representation which allows to determine the minimum number of processes from each class of processes required to represent a distinct realization of the abstract test case thanks to the analysis of selection guards. Thus, for any two instantiations of two different templates, we obtain necessarily two distinct concrete test cases. This method guarantees an optimal coverage of all the abstract test case behaviors by instantiating each one of its templates to a concrete test case with the required minimum number of processes. Besides, given subsets of representative processes for each process class, the concrete test case for that concrete process configuration can be generated from the already generated template without remodeling or re-executing the system. Finally, the concrete test cases are classically experimented against the implementation within the *ioco* conformance relation framework. In this work, authors use symbolic techniques to analyze scenarios where lifelines are denoted symbolically while we focus on the use of such techniques to analyze scenarios which are structured by combining operators and guarded by timing constraints.

8.2.4 Distributed testing with scenarios

Testing in distributed architecture with MSC has been investigated in [11, 10, 18, 25].

The work [25] defines a distrusted testing architecture for conformance testing from MSC with different assumptions on the observational power of the tester. The system behavior modeled by the MSC consists in message exchanges between the users and the subsystems which are distributed at different physical locations. This work distinguishes three kind of messages in the MSC: border messages exchanged between the users and the subsystems; user messages exchanged between users; and internal messages exchanged between the subsystems. The users behavior is performed by testers and the subsystem behavior is observed by testers. As usual in conformance testing, the conformance of an implementation with respect to its specification is expressed using a formal relation called *conformance relation*. Notice that the SUT is assumed to be modeled by partial orders (between communication actions) represented by MSCs. Traces correspond to the linearisation of all these partially ordered actions. Linearisation means simply here computing all the possible traces by interleaving the concurrent behaviors. The authors define three conformance relations: $b\text{-}conf$, $t\text{-}conf$, and $a\text{-}conf$. The relation $b\text{-}conf$ expects each tester to see a (local) trace which can be found in the projection of some specification traces after hiding the user messages, on the interface of the concerned user; $t\text{-}conf$ expects to find for each global trace of the implementation, a global trace in specification which has the same projection on all the users interfaces. $a\text{-}conf$ expects to find all the implementation traces in the specification. The latter assumes the tester to observe the internal messages between the subsystems. Test cases are all the possible concrete sequences of communication actions which can be observed by the tester. For example, under $b\text{-}conf$ and $t\text{-}conf$ the test case sequence does not contain internal communication actions between the subsystems. For a test case, verdicts are generated based on observational power of the testers. For $b\text{-}conf$, a global verdict is *pass* if the local testers have verdicts *pass*, otherwise the global verdict is $fail$. $t\text{-}conf$ requires two steps of testing: first, observe a local trace at each port, and then check if the overall trace is consistent with the specification and deduce a verdict (local verdict is computed here). For $a\text{-}conf$, the global verdict is *pass* if the observed global trace is specified. The last algorithm allows to verify the implementation at the subsystem level since the internal messages are observable and thus considered in the global trace. This work is in the frame of our concerns since our goal is also to decompose the testing task. Authors focus on distributed testing [46] while we have stated different results which relate to the compositional testing in the fashion of [17], besides in our approach time is taken into consideration.

Authors in [18, 85] are interested in distributed testing with MSC. More precisely, they look into how a test scenario can be correctly implemented in a distributed test architecture. The main

issue in implementing a distributed test is to guarantee that all the test events are executed in the right order, especially if the test components run concurrently. A coordination by message can be implemented between test components to generate a correct MSC test implementation, i.e. without false passes (i.e. masking SUT faults). This task is automated: the transformation of a test specification into a test implementation with a required coordination [11, 10]. Another solution is presented in [18]: Observable quiescence action called *null event* is introduced in the MSC and models a sufficient delay for the SUT to become quiescent and all the pending messages, including coordination ones, to arrive. Note that communications are assumed to be asynchronous and messages transit over unbounded FIFO channels. The work suggests two algorithms for generating test implementations and discusses their fault detection power in terms of false passes. Both algorithms rely on the addition of coordinating messages and null events in order to avoid races in the generated MSC of the test implementation. It is interesting to investigate links with our work because we have introduced the necessary elements in the semantics to capture the underlying communication mechanism which is used in [18] to realize the test scenario.

8.2.5 Testing criteria for scenarios

Authors in [6] present coverage criteria that determine the modeling elements of the *UML communication diagrams*[5] that a behavioral test must cover in order to be considered adequate. Authors consider the following coverage criteria: (a) *Condition Coverage Criteria*: any decision condition is evaluate to both $true$ and $false$. (b) *Full Predicate Coverage Criterion*: any clause in each condition is evaluate to both $true$ and $false$. (c) *Each Message on link Criterion*: any message is covered at least once. (d) *All Message Paths Criteria*: given a message, cover any path (sequence of messages) that contains that message at least once. This is done for all the messages in the diagram. (e) *Collection Coverage Criteria*: the participant may represent a collection of objects, in this case, the diagram is instantiated with any subset of objects in that collection. Note that when a sequence diagram contains an unbounded loop, the *All Message Paths Criteria* criterion is not doable. In practice, testers may set a bound. The approach in [77] defines control-flow coverage criteria for sequence diagrams: (f) *All-branches criterion* requires testing to execute enough start-to-end paths to cover all conditional behavior, similar to traditional branch coverage. We have used simply the criterion (c) during our simulations. It is interesting to validate our results for the other criteria however, we deal most of the time with (unbounded) looping behaviors. This requires to explore additional criteria borrowed from testing which are more suitable for these kind of behaviors such as *restriction by inclusion criterion* [36] and eventually adapt it to our timed context.

8.3 Eliciting unitary behaviors from scenarios

The exploitation of scenario models by the projection mechanism appears rarely in the literature. The study of the resulting properties of an assembly of components, separately validated or tested, receives much more attention (e.g. [91, 17]) than approaches such as ours, trying to infer component requirements from the whole system model. Among the latter [58] describes a complete tool framework that goes from the specification of high-level services as MSC scenarios to the derivation of models of the components to be designed/chosen using projection techniques.

Authors in [57, 58] use projection directly on MSC, they do not consider complex MSC with combining operator (e.g. alternatives and loops), just messages and guards. In fact, after having selected the component for which they want to construct an automaton specification, they project

[5] formerly called *a UML collaboration diagrams* contain the same modeling elements as sequence diagrams and describe the exchange of messages between participants but without the time dimension : messages can be numbered to show the exact order in which they are exchanged

each of the given MSCs on this component by removing all other components axes (lifelines in sequence diagram parlance), as well as message arrows that neither start nor end at the axis of the concerned component. Then they turn every remaining message into a transition of the automaton.

In [25], a conformance testing framework is defined for MSCs without intermediate representation as input output transition systems or finite state machines. The MSC traces which correspond to the linearisation of the partially ordered communication actions of that MSC are projected to obtain traces at the subcomponent level. The resulting trace is used by a distributed tester to conduct tests on the subcomponent locally. Our contribution is similar but is defined in a symbolic framework where time and data are handled as first-order structures rather than enumerated.

Chapter 9

Implementation and experiments

Contents

We have attributed operational semantics to sequence diagrams with timing features by translating them into a set of TIOSTS. We prototype our approach in the frame of the UML model-based development environment Papyrus and the symbolic simulator Diversity. The diversity tool has a generic entry language that we used to encode TIOSTS. The translation rules are implemented as a model to text transformation: TIOSTS specifications are generated in the entry text format of Diversity from the sequence diagram represented as an instance of the UML metamodel in Papyrus. Those specifications are symbolically executed to obtain a symbolic execution tree which characterizes in intension all concrete behaviors specified by the sequence diagram. We applied our approach on an example of a case study from the railways industry which allowed us to conduct symbolic simulations at a larger scale. We put a limit on the generated tree size by considering coverage criteria. In particular we are interested in achieving 100% message coverage defined for sequence diagrams [6, 77].

In the first section of this chapter, we present the tools that we used. The second section discusses implementation issues concerning the implementation of the TIOSTS formalism in Diversity. We also give in this section the implemented transformation algorithms. The third section provides the case study description. Finally the fourth section is dedicated to the experimentations and the evaluation of the scalability of our approach with regard to the coverage criteria.

9.1 Tools

In this section, we present respectively the Papyrus and the Diversity tools which are both developed at the CEA.

9.1.1 Papyrus

Papyrus [62] is a graphical editing tool for UML 2 as defined by OMG. Papyrus implements most of the OMG specification of UML in particular the sequence diagram metamodel. In addition, Papyrus provides an implementation of MARTE and offers a textual editor for VSL. We use papyrus to produce sequence diagrams in accordance with the OMG specification.

9.1.2 Diversity

Diversity [1] is a symbolic automatic analysis and testing tool. Diversity has a generic entry language called *xfsp* (e*x*tensible *f*ormal *sp*ecification) which has previously been used to encode specifications in SDL, Statecharts of Statemate, UML statemachines, ESTELLE, IF and Simulink stateflow. In our case, we encode the TIOSTS formalism with xfsp. Diversity generates a symbolic tree which represents all the possibles executions of the system. The symbolic tree is obtained by simulating the system specification with input symbols rather than concrete values for data. Each path of the tree has a constraint on input symbols, commonly called *a path condition*, for the execution to follow that particular path. Sequences of concrete test inputs are computed by solving these path conditions using a constraint solver. For that purpose, Diversity integrates solvers such as CVC3[2], OMEGA, Yices [3] and Z3.

In this section, we give some introductory background on a subset of the language xfsp and introduce the symbolic execution mechanism in Diversity.

9.1.2.1 Entry language xfsp

The language xfsp is flexible to capture different specifications based on communicating automata. We illustrate in Figure 9.1 the coding in xfsp of a simple IOSTS `G` (no timing features are specified) that computes the absolute value of its input (see the declaration of `G` as a `statemachine` of kind `or` in the xfsp syntax, line 6). We explain next how such a specification can be understood intuitively by making links with IOSTS syntax.

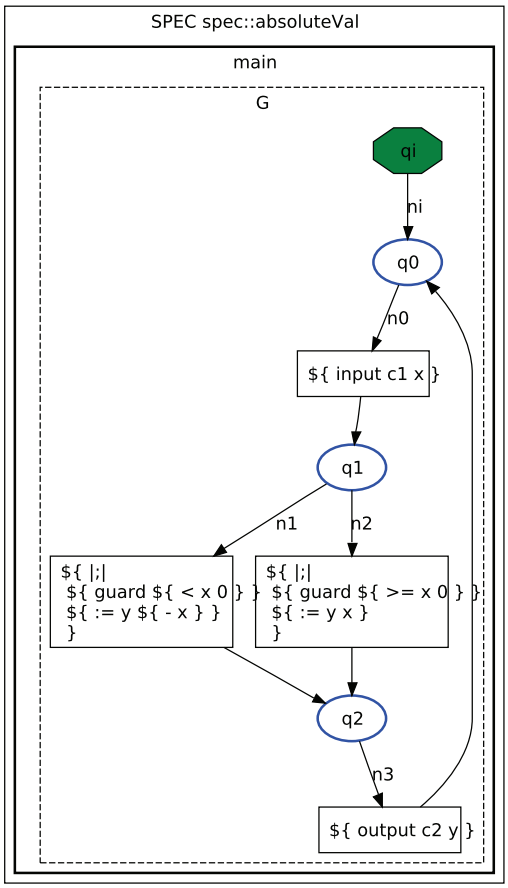
The IOSTS `G` has data variables `x,y` of type Integer (lines 8—9). It communicates over channels `c1,c2` (lines 10—11). In xfsp, each state is defined by a set of outgoing transitions. In the example, the state `q0` (line 17) has one outgoing transition namely `n0` and the state `q1` (line 22) has two outgoing transitions `n1,n2` both targeting the state `q2`. The transition `n0` denotes a reception of a value on channel `c1` stored in `x` (see line 19, `input c1(x)` encodes the communication action $c1?x$ in IOSTS syntax). Transitions `n1,n2` are exclusive: they are taken only if the value stored in `x` is respectively either a positive or a strictly negative integer (see the guards respectively lines 24 and 28). In the first case, the variable `y` is assigned with `x`, otherwise it is assigned with `-x` (see statements lines 25 and 29). Hence, `y` contains the absolute value of the input value stored in `x`. Then, the transition `n3` outputs this calculated value on channel `c2` (see line 34, `output c2(y)` encodes the communication action $c2!y$). Finally, the xfsp code specifies from where each channel gets data that may be either acquired from the environment or from another IOSTS (same for emitted data). In the example we consider only one IOSTS, hence $c1$ receives values from the environment and $c2$ emits towards the environment (see lines 39—43). We use this mechanism later in the chapter to encode the parallel composition of TIOSTS resulting from the sequence diagram translation.

[1] formerly named AGATHA tool set [76]
[2] *http : //www.cs.nyu.edu/acsys/cvc3/*
[3] *http : //yices.csl.sri.com/*

absoluteVal.xfsp

```
1  @xfsp< system , 1.0 >:
2
   @main:
4    statemachine< and > main {
     @machine:
6     statemachine< or > G {
       @declaration:
8        var integer x;
         var integer y;
10       public port input c1( integer );
         public port output c2( integer );
12     @machine:
       state< initial > qi {
14       @ni --> q0 {
          }
16      }
       state q0 {
18       @n0 --> q1 {
             input c1(x);
20       }
        }
22     state q1 {
          @n1 --> q2 {
24           guard (x >= 0);
             y = x;
26        }
          @n2 --> q2 {
28           guard (x < 0);
             y = -x ;
30        }
       }
32     state q2 {
          @n3 --> q0 {
34           output c2(y);
          }
36      }
     }
38   @com:
     connect< env > {
40      input G->c1;
     }
42   connect< env > {
        output G->c2;
44   }
  }
```

(a) xfsp code



(b) Visualization in Diversity

Figure 9.1: Simple IOSTS specification in Diversity

9.1.2.2 Symbolic execution

Diversity takes a specification of a system in xfsp as input and generates a symbolic tree. Let us recall once again that this tree characterizes exhaustively all the behaviors (i.e. traces) of the system by simulating the specification not for concrete input values, but for symbolic ones. The simulation determines constraints on these symbolic values for each behavior, that is the path condition (noted *PC* from now on). Consider the IOSTS specification in Figure 9.1. Its corresponding symbolic execution tree generated by Diversity is given in Figure 9.2. Nodes are symbolic states called *execution contexts* (EC) in Diversity which include besides the PC, an assignment of variables at each point during the execution and reached state of the IOSTS. Initially, the PC is equal to *true* and the variables `x` and `y` are assigned respectively to symbolic values `pid#2:x#0` and `pid#2:y#0` (see the label associated with the root node of the tree `EC 0`).

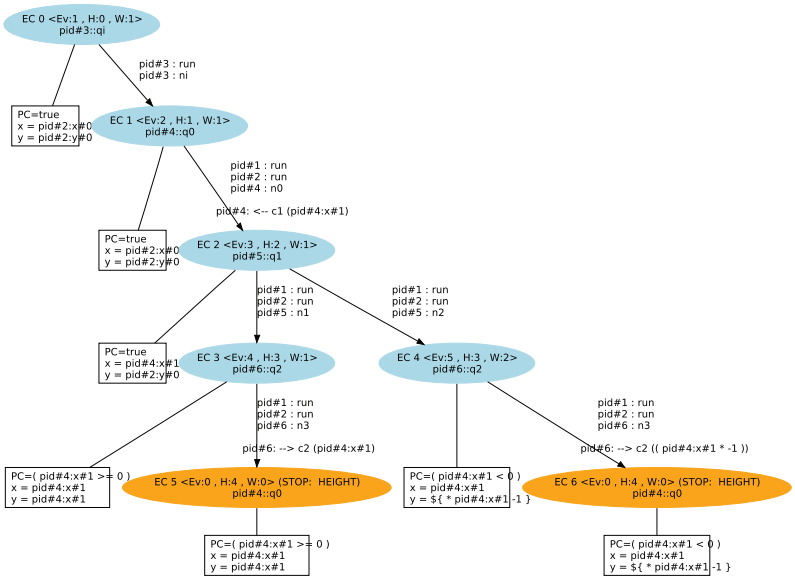
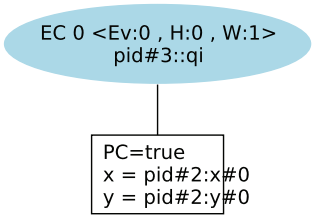


Figure 9.2: Symbolic execution tree in Diversity



At each point of the execution, the variable assignments and the PC are updated. For example, when the transition `n0` (`q0` $\rightarrow$ `q1`) is taken, the IOSTS receives an input `x` from the environment (the communication action $c1?x$ is performed) and hence semantically `x` is assigned a new fresh symbolic value `pid#4:x#1`, the current state becomes `q1` and the PC is left unchanged (see `EC 2`). From `q1` two possible transitions may be executed `n1,n2`. In the symbolic tree, this corresponds respectively to the branches `EC 2` $\rightarrow$ `EC 3` and `EC 2` $\rightarrow$ `EC 4`. The PC in `EC 2` is `pid#4:x#1 >= 0` and `y` is set to `pid#4:x#1`, the value stored in `x` which is the absolute value of `x` when it is a positive integer. Similarly, `EC 4` reflects the execution when `x` is negative (PC is equal to `pid#4:x#1 < 0` and `y={* pid#4:x#1 -1}`, i.e. `y={pid#4:x#1 * (-1)}` infix notation).

More generally, the discussed example is also illustrative of symbolic execution of a program fragment computing the absolute value of an input that may be specified in an imperative manner in Diversity. Our concern is state based specifications. The symbolic execution tree covers every possible input value. In particular for the example, both alternatives of `x` positive or negative integer are taken into consideration symbolically. The execution indicates which value is returned in both cases.

Trace generation Constraints of PC are solved with CVC3 in Diversity in order to obtain the test inputs. Symbolic inputs introduced during the symbolic execution are concretized with values which satisfy the PC. Figure 9.3 shows the tests generated based on the symbolic tree in Figure 9.2 using the solver CVC3.

Note that the IOSTS of the example models a reactive system in the sense that it continuously interacts with the environment: the IOSTS has a looping behavior where, at each iteration, the system receives an input on channel `c1` and emits a result on `c2`. Consequently, the symbolic tree

is infinite. Actually, we have chosen to stop the simulation when the tree height equals 4. In fact, Diversity may be parametrized with stop criteria. For this part of the symbolic tree (computed up to height 4), the generated test inputs meet the path coverage criteria. The symbolic tree has two execution paths corresponding exactly to the two cases `x>=0` and `x<0`. Consequently, the first scenario denotes a trace $c1?0.c2!0$ of the first path and the second scenario denotes a trace $c1? - 1.c2!1$ of the second path. These traces define the inputs $c1?0$ and $c1? - 1$ which make the execution follow those particular paths.

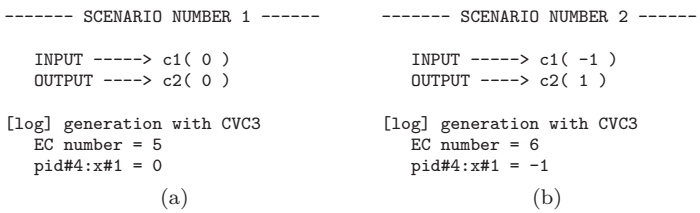


Figure 9.3: Test inputs generation in Diversity

Parametrization For the symbolic simulation, Diversity allows to define exploration strategies and stopping criteria.

Classically, search algorithms like *Depth First Search* (DFS) or *Breadth First Search* (BFS) are implemented. In addition, some other heuristic algorithms are implemented like the *HIT-OR-JUMP* algorithm [22]. In brief, the algorithm takes as input a sequence of transitions to cover and a parameter integer N and returns the path of maximal length N containing (a prefix of) that sequence (where some intermediate transitions may come in between).

Diversity offers some widely used coverage criteria in testing: states coverage, transitions coverage, formulas coverage and paths of a maximal length coverage [36]. Also it performs a more sophisticated criterion in order to avoid combinatory explosion : the *inclusion criterion* [36], based on the inclusion of EC, i.e. symbolic states. This criteria allows stopping the symbolic execution when it detects that an encountered EC is included in another already computed one. Intuitively, the inclusion means that the reached states are the same and that the constraints induced by the assignment of variables and the PC are stronger in the encountered EC than in the already computed one. The symbolic tree in Figure 9.4 was obtained with respect to the inclusion criterion. Symbolic states `EC 5` and `EC 6` are detected to be included in `EC 1` where the reached state is `q0`. In `EC 5` for instance, `PC={ pid#4:x#1 >= 0 }` is stronger than `PC=true` in `EC 1`. This means that the behavior expected after `EC 5` has already been covered.

9.2 Implementation

The implementation consists in accomplishing two tasks:

- encoding the TIOSTS formalism in Diversity
- implementing the translation rules as a model to text transformation in Papyrus.

9.2.1 Implementation of TIOSTS

We saw in Section 9.1.2.1 how an elementary IOSTS may be coded in Diversity. Now we present how we implemented a TIOSTS in Diversity. A TIOSTS `Ge` implementation is given in Figure 9.5.

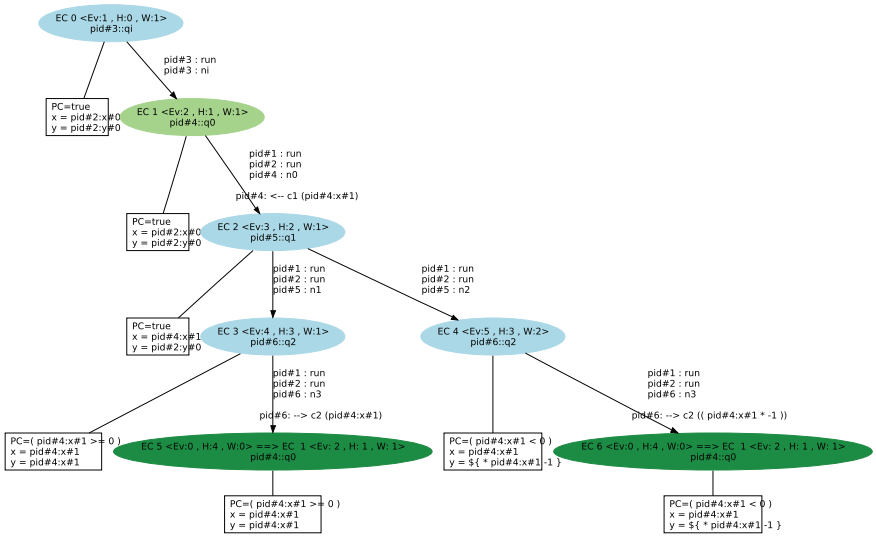


Figure 9.4: Symbolic execution tree in Diversity : inclusion criterion

In the same Figure, we also illustrated how the composition of two TIOSTS namely `G||Ge` is encoded. The implementation of `G` is entirely given in Figure 9.1 and can be seen as a TIOSTS implementation itself because it does not exhibit any timing features as discussed later in the section. Recall that `G` computes the absolute value of an integer value input. In the system, `Ge` uses `G` to compute the absolute value of a new integer value sent each 0.5 time slot. It requires the answer to be sent within 0.2 time slot.

The implementation of TIOSTS is similar to the implementation of IOSTS discussed in Section 9.1.2.1 except that it introduces additional code to handle the timing features that we discuss in the following. We first introduce two global variables `T` and `d` (lines 2—4) where `T` captures the current time and `d` captures the duration of the last executed transition in the system. Initially `T` is assigned 0 (see line 53). Each time any transition is executed, `d` is assigned a new fresh symbol and `T` is increased (see lines 56—58). E.g. after executing three transitions in the system, we may have this setting : `d` is assigned `pid#1:d#3` and `T` is assigned `pid#1:d#1 + pid#1:d#2 + pid#1:d#3` hence the current time (stored in `T`) is a sum of fresh durations. We can now discuss how a TIOSTS transition is implemented. Consider the following transition named $n5$: $q4 \xrightarrow{\{t2\} \quad t2[i\_t2]-t1[i\_t2]<0.2 \quad true \quad c2?w \quad [i\_t2 \leftarrow i\_t2+1]} q3$ (denoting the reception of the absolute value by `Ge`). Its implementation is given in (lines 43—47). What is important to mention is that the time variable ($t2$) associated with the transition is updated explicitly in the code (`t2[i_t2] = T`, line 43) and the index `i_t2` is incremented subsequently (`i_t2 = (i_t2 + 1)`, line 47). The weak form (refer to Definition 26) of the time guard $t2[i\_t2] - t1[i\_t2] < 0.2$ is directly expanded in the code (lines 44—45):
`tguard (t2[i_t2] - t1[i_t2] < 0.2) || (i_t2 < 0) || (i_t2 > (size t1))` . Finally, the data guard is omitted since equal to *true* and the output $c2?w$ is encoded by `input c2(w)` exactly as in the case of IOSTS. Actually, Figure 9.5 specifies the composition `G||Ge`. In the code we indicate explicitly which channels match for synchronizations because state machines in xfsp do not share channels. For example, `G` sends the absolute value on channel `c2` (see the action `output c2(y)`, line 34 in Figure 9.1) and `Ge` receives that value on channel `c2` (see the action `input c2(w)`, line 46). The correspondence is stated by connecting both channels in lines 61–64. Note that when two transitions are synchronized, both associated time variables are updated with the value of `T` being a global variable of the system.

```
@xfsp< system , 1.0 >:
@declaration:
  var real T;
  var real d;
@main:
  statemachine< and > main {
  @machine:
  statemachine< or > G {
    @declaration:
      var integer x;
      var integer y;
      public port input c1( integer );
      public port output c2( integer );
    @machine:
      ...
    }
   statemachine< or > Ge {
    @declaration:
      var integer u;
      var integer w;
      public port output c1( integer );
      public port input c2( integer );
      var vector<real> t1 ;
      var vector<real> t2 ;
      var integer i_t1 = 1;
      var integer i_t2 = 1;
    @machine:
      state< initial > qi {
       @ni --> q3 {
       }
      }
      state q3 {
       @n4 --> q4 {
         t1[i_t1] = T;
         tguard (t1[i_t1] - t1[i_t1 -1] == 0.5) ||
            (i_t1 < 0) || (i_t1 > (size t1)) ||
             (i_t1 - 1 < 0) || (i_t1 - 1 > (size t1)) ;
         i_t1 = (i_t1 + 1);
         input u;
         output c1(u);
       }
      }
```

(a) xfsp code

```
      state q4 {
       @n5 --> q3 {
         t2[i_t2] = T;
         tguard (t2[i_t2] - t1[i_t2] < 0.2) ||
                (i_t2 < 0) || (i_t2 > (size t1)) ;
         input c2(w);
         i_t2 = (i_t2 + 1);
       }
      }
     }
    @moe:
     @init{
       T = 0;
     }
     @irun{
       input d;
       guard (d >= 0) ;
       T = (T + d);
     }
    @com:
     connect< rdv > {
       output Ge->c1;
       input G->c1;
     }
     connect< rdv > {
       output G->c2;
       input Ge->c2;
     }
   }
```

(b)

Figure 9.5: TIOSTS specification in Diversity

Symbolic execution and test input generation We illustrate the timed symbolic execution by giving in Figure 9.7a a symbolic state in the execution tree. The generation of the tree was stopped at height 4 where all states and transitions were covered. The execution reaches the symbolic state `EC 6` (called execution context (EC) in Diversity) where the states `q0` of `G` and `q3` of `Ge` were revisited again. Let us look at the assignment of variables which relate to the time. First, three fresh durations `pid#1:d#1`, `pid#1:d#2` and `pid#1:d#4` were successively cumulated in `T` (`T={+ pid#1:d#1 pid#1:d#4 pid#1:d#2}`). The time variables `t1` and `t2` and their associated indexes are updated as follows: `t1[0]=pid#1:d#1`, `t2[0] = {+ pid#1:d#1 pid#1:d#4 pid#1:d#2}`, `i_t1=1` and `i_t2=1`. The path condition PC states that there is at most 0.2 between `t1[0]` and `t2[0]` (that is `(pid#1:d#4 + pid#1:d#2)< 0.2`). There is no constraint on `t1[0]` and `t1[-1]` because it corresponds to the location `i_t1-1` of `t1` being out of bound and hence $WF(t1[i\_t1] - t1[i\_t1 - 1] = 0.5)$ evaluated to $True$. PC constraints are solved with CVC3 in order to obtain the scenario in Figure 9.7. Symbolic inputs introduced during the symbolic execution are concretized with values which satisfy the PC. Figure 9.3 shows the tests generated based on the symbolic tree in Figure 9.2 using the solver CVC3. The scenario denotes the following trace $1.c1! - 3.(0.05).(0.1).c2!3$. Let us remark the delay of $0.15 = 0.05 + 0.1 < 0.2$ between the first communication $(c1! - 3)$ between `G` and `Ge` (the latter asks to compute the absolute value of -3) and the response $(c2!3)$ of the former.

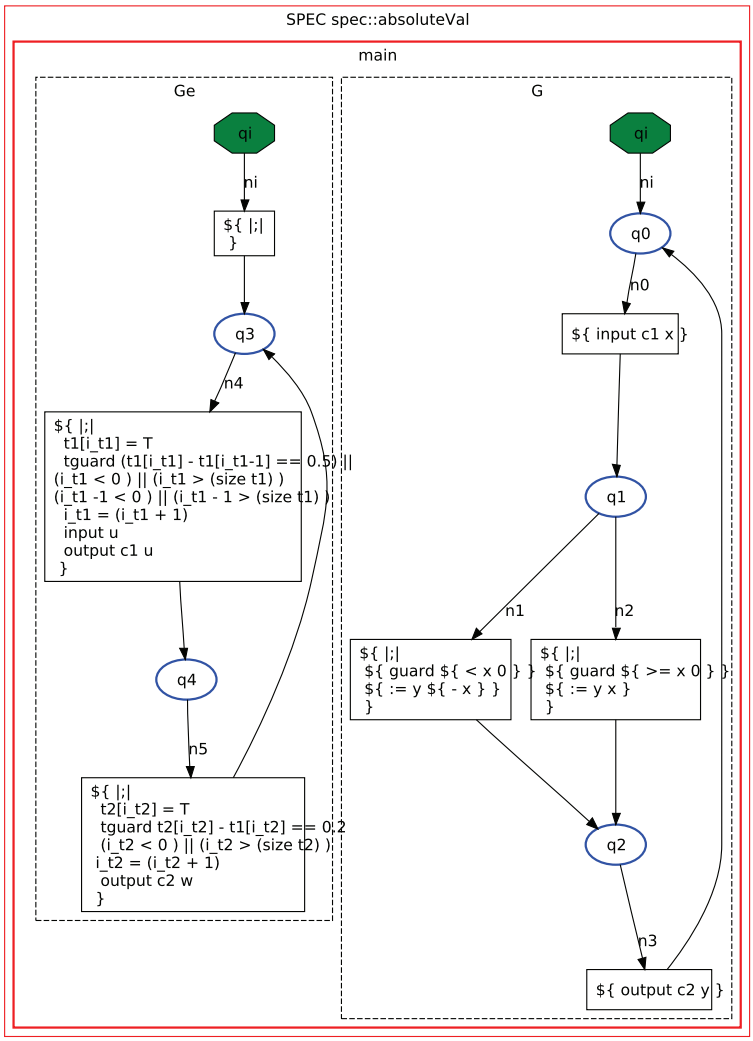
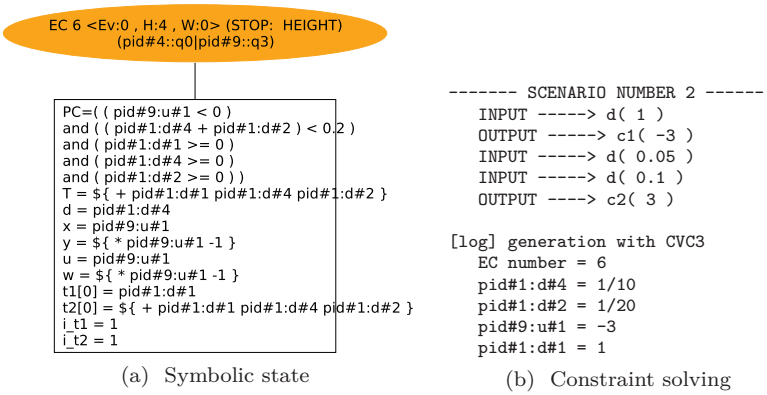


Figure 9.6: TIOSTS: graphical view in Diversity



(a) Symbolic state (b) Constraint solving

Figure 9.7: Test inputs generation form TIOSTS symbolic execution

9.2.2 Implementation of the translation rules

We present in this section the implementation of the translation rules as model to text transformation. The transformation takes as input a UML MARTE sequence diagram in accordance with our use, as described in Chapter 2, and produces a set of TIOSTS corresponding to the sequence diagram translation as defined formally in Chapter 6. In practice the transformation generates

xfsp code encoding these TIOSTS in Diversify. The transformation is mainly made up of two sub transformation algorithms: a transformation for the messages which is quite straightforward and a much more subtle transformation for the lifelines. The difficulty comes from the fact that we synthesize an automaton per lifeline while the behaviors are captured by the metamodel globally at the operators level. In other words, units of behaviors (such as sending/reception of messages) belonging to different lifelines are grouped by region then structured with operators. Therefore, we need to extract from these groups of behaviors those concerning each single lifeline without loss of information about their structuring with operators. This requires the understanding of how the graphical positioning of behaviors (units of behaviors and operators frames) in the diagram is captured by the metamodel. This particular point is also discussed in this section.

The reader of this section is supposed to be familiar with the metamodel constructs introduced in Section 2.2 of Chapter 2.

Message Translation

We suggest an algorithm to automatically implement a message TIOSTS as an `OR` state machine in the Diversity parlance. The algorithm is given in figure 9.8 and takes as inputs a UML element *Message*, a mapping [*OccurrenceSpecification*, *TimeObservation*] which allows to obtain the time variables associated with the message and a mapping [*Message*, *Constraint*] which allows to obtain the constraint associated with the message. From a practical perspective, it would be slow to parse the collection of constraints of an *Interaction* and the collection of observations of the model each time a message or a lifeline is translated. We structure this information in the mentioned mappings once and for all. We assume in the algorithm that the message is timed, that is two time variables and a constraint are defined for the message (which is not the case for simple messages).

Algorithm 2: TranslateMessage

Data: m: Message,
observations: Map[OccurrenceSpecification, TimeObservation],
constraints: Map[Message, Constraint]

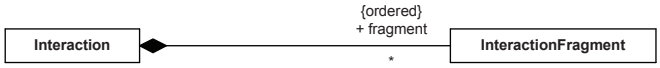
1 **begin**
2 create state machine $machine_{<m>}$ of kind `OR` ;
3 $t_1 \leftarrow observations.get(m.getReceiveEvent())$;
4 $t_2 \leftarrow observations.get(m.getSendEvent())$;
5 add to $machine_{<m>}$ the variables ;
6 $< t_1 >, i_{<t_1>}, < t_2 >, i_{<t_2>}$, fifo $f_{<m>}, x_{<m>}$;
7 add to $machine_{<m>}$ the ports ;
8 $< m >_{in}, < m >_{out}$;
9 add to $machine_{<m>}$ the state q as initial, the state q' as final ;
10 add to q the transition targeting q', having the statements ;
11 $i_{<t_1>}$ `= 0;` $i_{<t_2>}$ `= 0;`
12 add to q' the transition targeting q', having the statements ;
13 `input` $< m >_{in}(x_{<m>})$; `push` $f_{<m>}$ `(` $x_{<m>}$ `);`
14 $< t_1 >$`[`$i_{<t_1>}$`] = T;` $i_{<t_1>}$ `=` $i_{<t_1>}$ `+ 1` ;
15 $constr \leftarrow constraints.get(m)$;
16 add to q' the transition targeting q', having the statements ;
17 `output` $< m >_{out}$(top $f_{<m>}$); `pop` $f_{<m>}$;
18 $< t_2 >$`[`$i_{<t_2>}$`] = T` ;
19 `tguard` $WF(< constr.getSpecification() >)$; $i_{<t_2>}$ `=` $i_{<t_2>}$ `+ 1` ;
20 **return** $machine_{<m>}$

Figure 9.8: Model to text transformation: Generate xfsp machine (TIOSTS in Diversity) for the UML element *Message*

Lifeline translation

Each lifeline is associated with a TIOSTS. In order to define this automata, we need to deduce the order on units of behaviors within each lifeline.

Graphical order Recall that these units of behaviors correspond to the *Interaction Fragments* in the metamodel. How the metamodel captures the graphical order of the *Interaction Fragments* in the diagram is a key point to deduce the local order on lifelines. For instance within an elementary sequence diagram, the *Interaction Fragments* which are in this case exactly *Occurrence Specifications*, execute on each lifeline in their graphical order from top to bottom of the line. However, the *Interaction Fragments* graphical order in general is not captured at the lifeline level but rather at the *Interaction* level as we will see in the following. Let us focus on the composition relation between an *Interaction* and *CombinedFragment* elements:



It states that all the *fragments* of the interaction are ordered (see the label on the composition line *ordered*). The order comes from the vertical coordinates of the fragments in the diagram as expressed in the (OMG) specification, in natural language :

Excerpt from UML specification "[...] In Sequence Diagrams these InteractionFragments are ordered according to their geometrical position vertically. The geometrical position of the *InteractionFragment* is given by the topmost vertical coordinates of its contained *OccurrenceSpecifications* or symbols." (page 501)

"[...] The vertical position of an *OccurrenceSpecification* is given by the vertical position of the corresponding point. The vertical position of other *InteractionFragments* is given by the topmost vertical position of its bounding rectangle." (page 486)

Example 53 *Consider the sequence diagram of Figure 9.9. The fragments associated with the interaction sd_1 are recorded in the order of their vertical coordinates that is : the emission and the reception of m_1 (at y_1) then the emission and the reception of m_2 (at y_2) then the loop operator (at y_3) then the emission and the reception of m_5 (at y_4). Note that this order is not of course the order of execution : e.g. according to the semantics of sequence diagrams, the emissions of m_1 and m_2 may occur in any order whereas graphically m_1 occurs before m_2 ($y_1 < y_2$).*

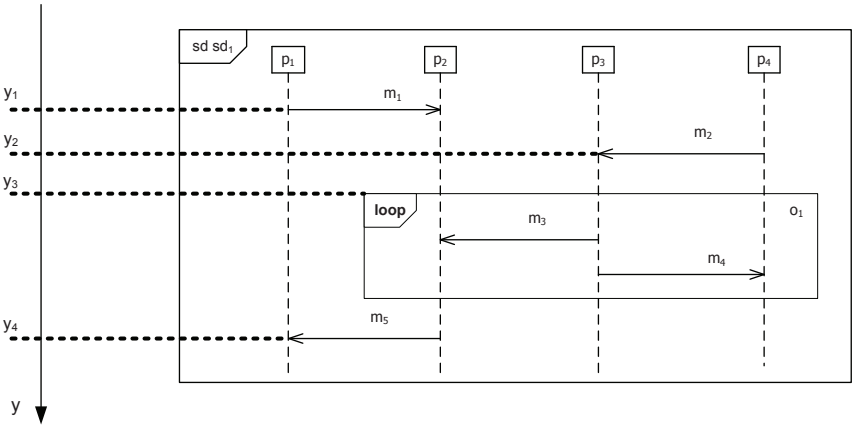


Figure 9.9: Graphical order of fragments implied by the metamodel

Note that fragments inside the loop operator region are not considered as fragments of the interaction explicitly. In fact, similarly to an interaction, any operator region, i.e. element InteractionOperand *in the metamodel in Figure 2.9, contains fragments which are ordered the same way : e.g. the fragments of the loop operator region have fragments ordered as follows : the emission and the reception of m_3 then the emission and the reception of m_4. This allows recursive structuring of the sequence diagram by the metamodel. In some way, a region of an operator is considered as if it were an interaction itself.*

Local order on lifelines We now show how to deduce the order of fragments (including *CombinedFragments*) on each lifeline. The *Interaction* is composed of a set of *InteractionFragments*. The fragments are ordered according to their geometrical position vertically. In order to deduce the order of fragments on each lifeline, let us consider the association between the *Lifeline* element and the *InteractionFragment* element. In fact, any fragment is related to a lifeline by the association with the role *covered*. For a given lifeline, by visiting the ordered fragments which belong to the interaction and extract only those which are covered by that lifeline, we get a set of fragments having the same vertical position, the one of the lifeline. If the fragment is a *CombinedFragment*, its nested fragments are also visited. In fact, in order to obtain the TIOSTS automaton of a lifeline, the fragments of the interaction are traversed recursively in a depth first manner. The formulation of the textual syntax we attribute to sequence diagrams in Chapter 4.3, takes into account this projection algorithm at the level of the lifeline expression.

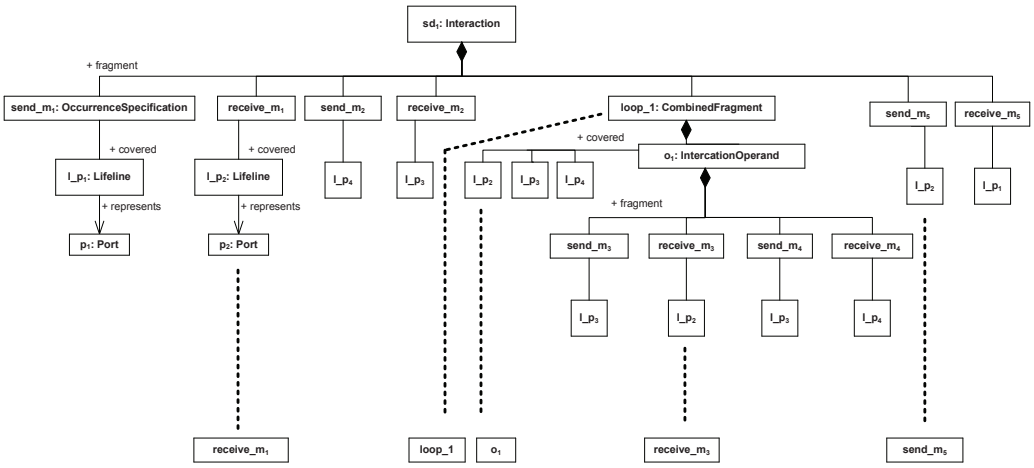


Figure 9.10: Local order on lifelines

Example 54 *Consider again the sequence diagram in Figure 9.9. A partial overview of its structuring with the metamodel elements is given in Figure 9.10. Also, we illustrate in the figure how to obtain the local order of fragments on the lifeline associated with the port p_2 from the set of fragments of the interaction sd_1. The fragments are presented in order (graphic one) from left to right. The fragments of the loop* CombinedFragment*, contained the region o_1, are also orderly represented. Here, we can see how the depth-first traversal of the fragments set works. As stated before, fragments directly related to the interaction are visited in order, for the port p_2 only one fragment is kept which is the* OccurrenceSpecification *$receive_{m_1}$ before arriving at the fragment $loop_1$. The latter contains fragments as well, they are visited at their turn keeping only the fragment $receive_{m_3}$ covering the lifeline of p_2. Finally, the remaining fragments of the enclosing interaction are visited where only the fragment $send_{m_5}$ concerns p_2.*

Recursive algorithm The core of the lifelines transformation routine is given in figure 9.11.

Algorithm 3: TranslateLifelines

```
Data: lifelines: Set[Lifeline],
fragments: List[InteractionFragment]
1  begin
2      machines: Map[Lifeline,state machine] ;
3      for frag next in fragments do
4          if frag is ActionExecutionSpecification then
5              machines ← translateAction(machines, frag) ;
6          if frag is MessageOccurrenceSpecification then
7              machines ← translateComAction(machines, frag) ;
8          if frag is CombinedFragment then
9              machines1,machines2: Map[Lifeline,state machine] ;
10             operand1, operand2: Operand ;
11             for i ∈ {1, 2} do
12                 operand_i ← frag.getOperands()[i]  /* if i is a valid index */ ;
13                 machines_i ← TranslateLifelines(operand_i.getFragments(), operand_i.getCovereds())
14             operator ← frag.getInteractionOperator() ;
15             if operator is loop then
16                 machines' ← TranslateLoop(machines_1, operand_1);
17                 machines ← TranslateSeq(machines, machines') ;
18             if operator is alt then
19                 machines' = TranslateAlt(machine_1, machine_2, operand_1, operand_2) ;
20                 machines ← TranslateSeq(machines, machines') ;
21             if operator is strict then
22                 machines' = TranslateStrict(machine_1, machine_2, operand_1, operand_2) ;
23                 machines ← TranslateSeq(machines, machines') ;
24     for lf in lifelines do
25         if lf has no machine_<lf> in machines then
26             create state machine machine_<lf> of kind OR ;
27             add to machine_<lf> the state q as final/initial ;
28             machines.put(lf, machine_<lf>) ;
29     return machines
```

Figure 9.11

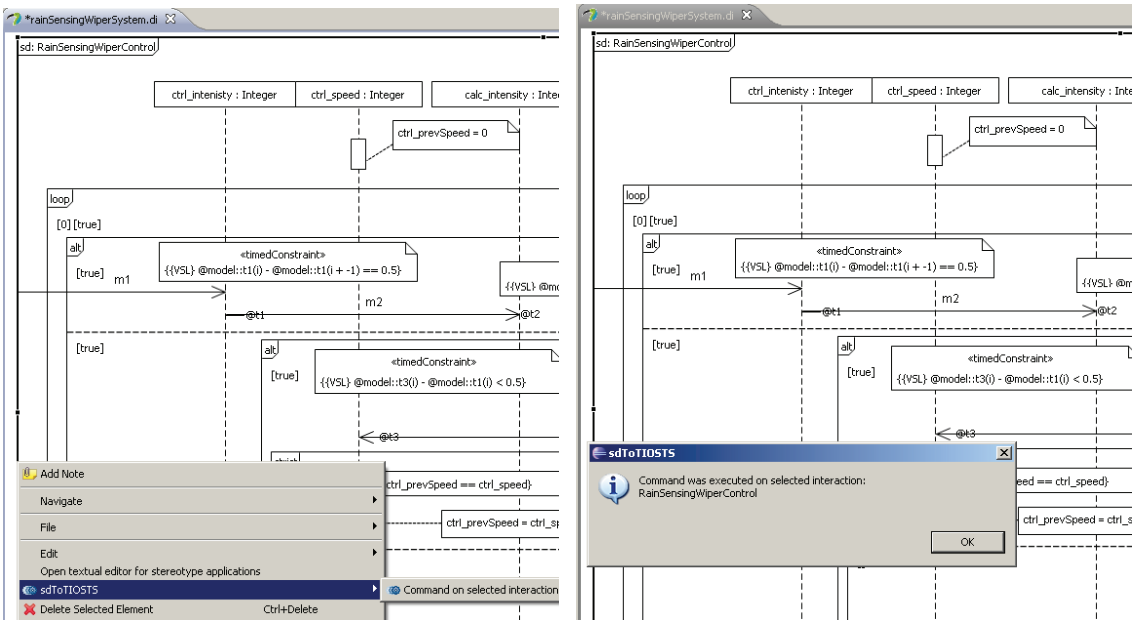
The algorithm takes as input a set of UML Lifeline elements and an ordered list of UML InteractionFragments elements. It returns a mapping [*Lifeline*,state machine] that associates an OR state machine with each lifeline of the input set. In the recursive phase of the algorithm, the routine is called on the fragments and the lifelines of the nested combined fragments at the first level. This is the case when the fragments are combined ones. Otherwise the fragment in the input list is either a MessageOccurrenceSpecification or a ActionExecutionSpecification. The first UML element denotes a sending or reception of a message and the second one denotes an assignment action or underspecification action. Thus no recursive call is needed but the call of respectively the routines *TranslateComAction* and *TranslateAction*. These routines increase the state machine of the concerned lifeline (covered by the fragment) by a transition to denote its associated action.

Recall that an Interaction or an Operand of a combined fragment has an ordered list of fragments but the fragments in the list belong to different lifelines: the order here is given geometrically by the vertical position of the fragment in the diagram. A lifeline may be concerned by the Interaction or an Operand without having any fragment in the list [4]. This case is trivial, the algorithm generates a state machine with one state : see lines $27-29$. This phase of the algorithm comes naturally after the completion of loop parsing the fragments list because if no state machine was generated for the lifeline (line 26) then the lifeline would have no fragment in that list. Following the explanations given previously about the local order on lifelines, we now show the first phase of the algorithm. Assume without loss of generality that all the fragments of the list belong to one lifeline. Parsing the list in order (line 3 *next in fragments*) and concatenating the generated states/state machines in sequence constructs the expected machine for the lifeline in

[4]For example being in a strict operand, the lifeline behavior has to wait for other lifelines to complete theirs in that same operand before leaving the operand.

terms of behavior precedence. We generalize to a group of lifelines simply by considering the projection of the list of fragments based on the belonging of a fragment in the list to a lifeline [5].

Plug-in



(a) (b)

Figure 9.12: Papyrus plug-in

We have implemented the translation in the Java programming language as an Eclipse plug-in `sdToTIOSTS`. The use of the plug-in is illustrated in Figure 9.12. It is executed on the sequence diagram of RWC system in Papyrus.

The automation of the translation allows us to conduct experiments on larger system models like the railway use case that we present in the next section.

9.3 Application to a railway use case: A train reversing direction of traction

In early design phases, requirements are still expressed in natural language. We believe that sequence diagrams come close to the way one would specify dynamic requirements in a natural language. The goal of this section is to illustrate by means of an example in the frame of railway systems, that the constructs of sequence diagrams have the expressive power to capture the behavior of the system at the requirement level. Starting from requirements described in natural language, we suggest to specify the system behavior as a sequence diagram following these steps :

- We present the system architecture as a flow-oriented UML composite diagram [73] which describes the components of the system, their ports and the way they are connected.
- We identify the combining operators.

[5] The belonging of a fragment to a lifeline is available in the meta model by the association covered.

- We identify the set of conveyed messages through this architecture and their functional roles.
- We relate time and data constraints to these combined interactions.

In the rest of the section, we show how to apply our modeling methodology to the design of this railway use case. The objective of the use case is to operate a train along the tracks, in the desired direction of traction, while ensuring that all safety parameters and delays are always respected. This analysis is based on a working document specifying the use case in natural language.

9.3.1 The Automatic Train Control (ATC) system overview

The ATC is a railway system described in this chapter from the perspective of the French based railway industry. The system consists of the following components, as shown in Figure 9.13 :

- Automatic Train Supervisor (ATS) manages the train movements according to the train timetable and sends instructions for that purpose.
- Automatic Train Protector (ATP) ensures the basic safety requirement which is to keep a safe distance between the two trains. The ATP is made in its turn of two components namely the carbone ATP (cATP) and the wayside ATP (wATP) such that,
  - the wayside ATP (located along the tracks) evaluates the possible danger ahead of the train.
  - the carborne ATP (located on the train) evaluates cabin activation and analyzes the traction authorizations.
- Automatic Train Operator (ATO) provides controls to replace the driver such as stop and start commands.
- Rolling Stock RS (the train itself)

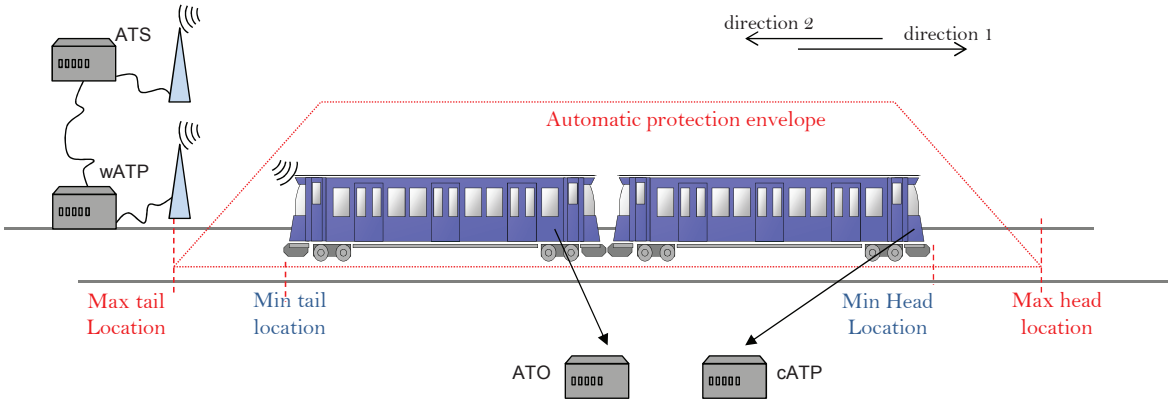


Figure 9.13: Components of the Automatic Train Control (ATC): the ATS and the wayside ATP are on a wired network. The carbon ATP and ATO on the train are part of the network on the train. The components on the train communicate with the ATS and the wayside ATP via a radio link — Protection envelope to ensure safety of the train

Actually, some components are redundant in the ATC system. We simplified redundancy not relevant to our study. We are interested in a specific use case of the system namely *reversing direction of traction.*

Use case

A train can move towards one direction at a time: direction 1 or direction 2. When moving towards direction 1, the front cabin is considered as active, otherwise the rear cabin is active. First, the ATS demands to shift the train direction of traction. Then, safety messages are exchanged between the components ATP and ATO. The timing constraints relate to the temporal validity of the messages. Finally, the ATP provides the RS with the authorization to traction in the chosen direction.

In order to ensure the safety of the train, it is essential to ensure that the train does not run into obstacles, or no other trains runs into the rear of this train. This is done by maintaining, at all times, a protection envelope (as shown in Figure 9.13) for the train, which is maintained obstacle free. The ATO and the carbone ATP ensure that this safety envelope is always maintained.

9.3.2 ATC system architecture as a composite diagram

Recall that we consider a system as an assembly of components with entry points, represented by ports, and connectors as glue between them. Before defining the sequence diagram for the ATC system, we introduce the architecture of the system as a UML composite structure diagram. Figure 9.14 represents such a composite structure diagram that we qualify as *the system architecture*.

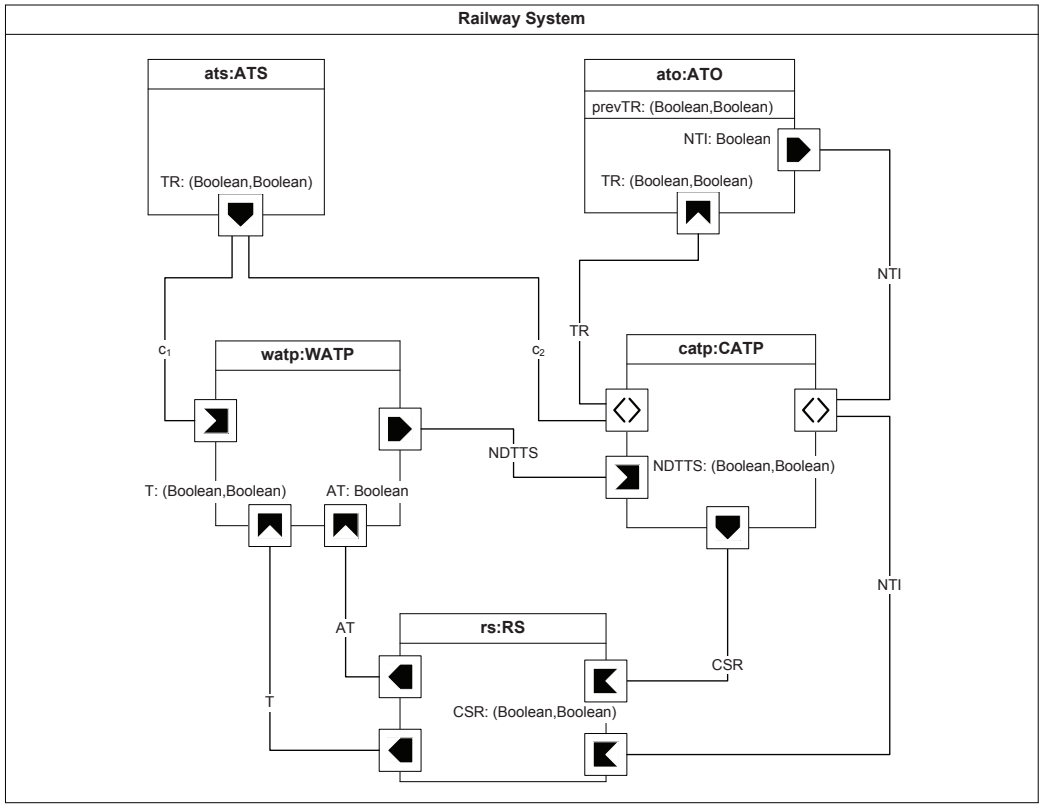


Figure 9.14: Railway system signature as composite diagram

Components Five components which exchange data through their ports, are introduced according to the system description given previously: the *ats*, the *ato*, the *watp*, the *catp* and the *rs* (drawn as boxes). They correspond to the UML parts in the diagram: part *ats* of type ATS, part *ato* of type ATO, etc. Some/all components may have variables used for computations,

Table 9.1

Port	Description	Type	Example and signification
TR	Traction request	$Boolean \times Boolean$	$(true, false)$ request traction in direction 1
NTI	No traction inhibition	$Boolean$	$true$ traction release in direction 1
NDTTS	No danger to traction switch	$Boolean \times Boolean$	$(true, false)$ safe for traction in direction 1
CSR	Authorized to traction and active cabin	$Boolean \times Boolean$	$(true, false)$ authorize traction in direction 1
T	Traction	$Boolean \times Boolean$	$(true, false)$ current traction is in direction 1
AT	Active train	$Boolean$	$true$ train is active

called calculation variables. The variable $prevTR$ is a calculation variable of the component *ato* and stores the last direction applied to the rolling stock.

Ports Each component owns ports to communicate with other components. The component *ats* communicates the traction request through its port $ats.TR$. The components $catp, watp$ receive the request through their ports TR. Table 9.1 summarizes the characteristics of the component ports in terms of functional description and typing. Also, are given some examples of concrete values that ports may be assigned with. Remember that ports are considered in our framework as particular variables which store exchanged values between components. For instance, the port TR denotes a *traction request* in one of the two possible directions. The port TR is of type $Boolean \times Boolean$. When it is assigned the value $(true, false)$, the traction is requested in the direction 1.

The connectors represent the communication media between the ports of the components. In the example, there are nine connector lines between entities: an example of a connector is the one linking the ports TR of the components *catp* and *ats*.

9.3.3 ATC system behavior as a sequence diagram

In this section, we present the sequence diagram *sd reverseDirectionOfTraction* illustrated in Figure 9.15 of the ATC system. Notice that we have not shown the message exchanges and local actions, just the skeleton of the main applied combining operators. And so regions of the diagram were hidden by special blocks with the key word *ref* inside the operator frames. This shorthand is available in UML to ease the specification of complex behaviors. The ref blocks are labeled by the name of the sequence diagram defining the hidden region and shown elsewhere in what follows (e.g. the *ref* block labeled with *sd initialization* hiding the upper region of *sd reverseDirectionOfTraction* in Figure 9.15, is defined in another diagram: see Figure 9.16). The specification consists of the five diagrams represented in figures 9.15–9.19.

9.3.3.1 Reversing direction of traction

The sequence diagram *sd reverseDirectionOfTraction* represents all the intended interactions between all the components of the ATC system each time the train changes direction. For any port of a component, there is a lifeline is the diagram. A total of fifteen lifelines take part in the interactions. We give the textual expression of *sd reverseDirectionOfTraction* as a couple of sets :

$(\cup_{1 \leq i \leq 16}\{m_i\}, \{lf_{ats.TR}, lf_{ato.TR}, lf_{ato.NTI}, lf_{catp.TR}, lf_{catp.NTI}, lf_{catp.CSR}, lf_{catp.NDTTS},$

$lf_{watp.TR}, lf_{watp.NDTTS}, lf_{watp.TA}, lf_{watp.TDIR}, lf_{rs.NTI}, lf_{rs.CSR}, lf_{rs.TA}, lf_{rs.TDIR}\}$).

The first set is the set of messages. As mentioned before, the set of messages is not represented yet but is explored later in the section. The second set is the set of lifelines. An element of this set for example is the lifeline $lf_{ats.TR}$ associated with the port $ats.TR$. This lifeline captures at this port level, all the requests emitted by the component ats to change the direction of traction.

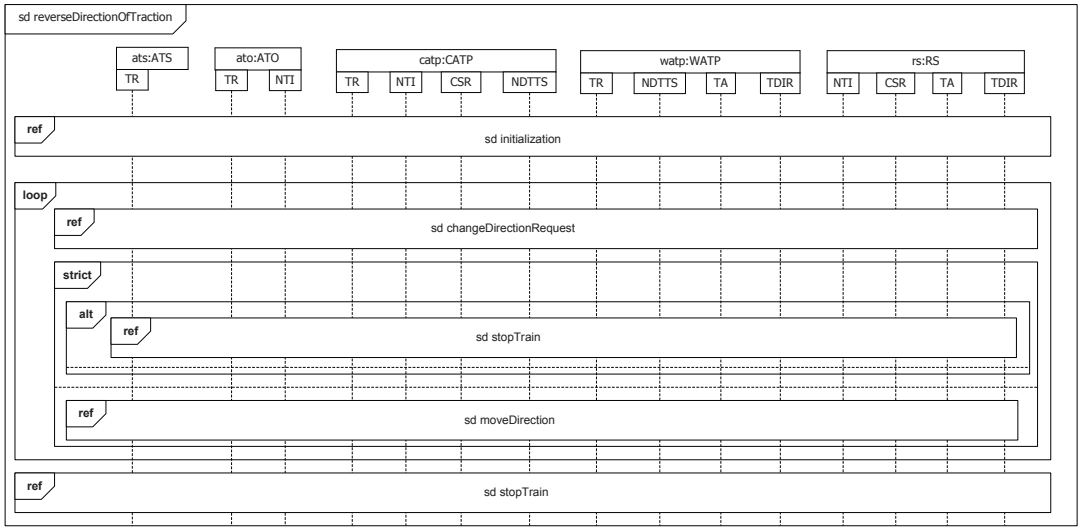


Figure 9.15: Use case: Reversing direction of traction as sequence diagram

Combining operators

The *sd reverseDirectionOfTraction* contains at the first level of operators hierarchy (in terms of frame nesting) respectively from top to bottom: a *ref* block labeled *sd initialization*, a *loop* operator and a *ref* block labeled *sd stopTrain*. The sequence diagram is built that way because the train is initially stopped then after running while repeatedly changing direction of traction (the *loop* iteration number is unknown beforehand) the train is stopped again. Now consider the hierarchy inside the *loop* region, we have respectively a *ref* block labeled *sd changeDirectionRequest* and a *strict* operator sequencing an *alt* region with a *ref* block labeled *sd stopTrain* then a *ref* block labeled *sd moveDirection*. This specifies the cyclic behavior of the system including starting the train. Firstly, a request to traction in a given direction is emitted (*sd changeDirectionRequest*). Then, there are two possibilities captured by the *alt* operator: The train is expected to halt (*sd stopTrain* in the first region of the *alt* is executed) or not (the second region of the *alt* is empty [6]). In fact, if the requested direction does not change or when the train is started, stopping the train is meaningless in these cases. However, safety requirements are still checked to keep the train moving in a given direction (*sd moveDirection*).

Initialization

The initialization (specified in Figure 9.16) consists of two assignment actions. The first action initializes the variable $ato.prevTR$ with the value $(false, false)$. This variable stores the train last direction of traction. When the execution starts, no traction has been applied yet and thus this specific value is distinct from both direction values (that is $(true, false)$ for direction 1 and $(false, true)$ for direction 2). The second action assigns the port (considered as a variable too) $rs.AT$ with the value $false$. In general, the values communicated through this port are considered

[6] When the second region of an *alt* operator is empty, it is renamed to the operator *opt* and drawn without the second region in the frame of the operator.

as heartbeats making the system aware that the rolling stock (train) is active. Initially the train is not active and so the value available on this port AT (Active Train) of the component rs is set to $false$.

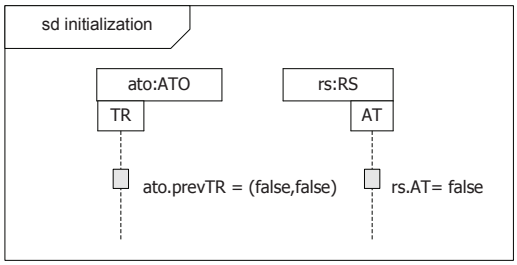


Figure 9.16: Use case: Initialization

9.3.3.2 Reversing direction of traction request

The sequence diagram in Figure 9.17 describes how a new direction of traction request propagates in the ATC system. Four components take part in the described interactions, exactly via their port TR (Traction Request) namely $ats, ato, catp, watp$ by exchanging messages (arrows in Figure 9.17).

Messages

The messages represent the data conveyed between two ports. In the example, the message m_1 (respectively m_2) transmits the request for a direction of traction by the component ats to the component $catp$ (respectively $watp$) through their dedicated ports TR.

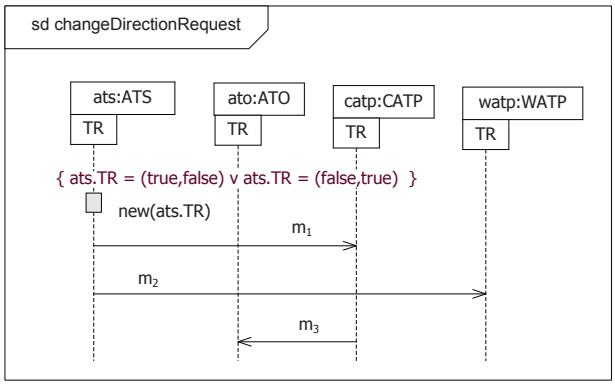


Figure 9.17: Use case: Reversing direction of traction request

Let us see what behavior is specified here. Firstly, at the component ats level, the action new is performed on the port TR. Its effect is to assign a random value to that port, here a new direction of traction [7]. It can be the same as the former direction assigned to the port in a previous iteration of the enclosing $loop$ operator. However, the data constraint $ats.TR = (true, false) \vee ats.TR = (false, true)$ states that the new value is a request for a traction either in direction 1 or direction 2: the traction may be requested by the ATS component in one direction only. This is one of the *safety requirements* of the ATC system.

[7] This represents an underspecification since the way the new direction of traction is computed by the ATS according to the train timetable is abstracted away.

Finally after communicating the new value to components *catp* and *watp*, the former notifies the component *ato* via message m_3 of that value.

9.3.3.3 Stopping the train

The *sd stopTrain* is shown in Figure 9.18. The diagram provides the process to stop the train: the train must be stopped in order for it to change direction of traction. The component *ato* (Automatic Train Operator) is responsible for this decision which depends on the previous applied direction (stored in the variable $ato.prevTR$): if the newly received request of traction (on the port $ato.TR$) is in the same direction, the train does not stop. This is what the data constraint $ato.prevTR <> ato.TR \wedge ato.prevTR <> (false, false)$ expresses. Note that the second part of the conjunction is needed for the constraint to hold at the first iteration, when the train is initially stopped (the variable $ato.prevTR$ is set to $(false, false)$).

In order to make the train grind to a halt, the component *ato* informs the component *catp* (both located on the train) of its decision to inhibit the traction. In fact, the port NTI (No Traction Inhibition) of the component *ato* is assigned with $false$ (see the assignment atom on the lifeline $lf_{ato.NTI}$) and thus this value is sent to the *catp* (through the message m_4). In its turn, the *catp* forwards the traction inhibition decision to the train itself rs (through the message m_5). The train is then stopped (see the atom $rs.T = (false, false)$, no traction T is applied). However the train is still active (see the atom $rs.AT = true$) because it is just a temporary halt before running again. Both informations (conveyed respectively by messages m_6 and m_7) are transmitted, in any order, to the component *watp* that is located on the tracks. In this case, the *watp* tells the *catp* that there is a danger to traction in both directions ($watp.NDTTS$ is set to $(false, false)$ and sent through m_8, remember that $NDTTS$ stands for No Danger To Traction Switch). Next, the cabin oriented to the previous direction is switched off, so at this point of the stopping process both cabins are off ($catp.CSR$ is set to $(false, false)$) and finally the *catp* informs the rs (through m_9) of the cabin statuses. Note that the *strict* operator forces the sequencing described before [8].

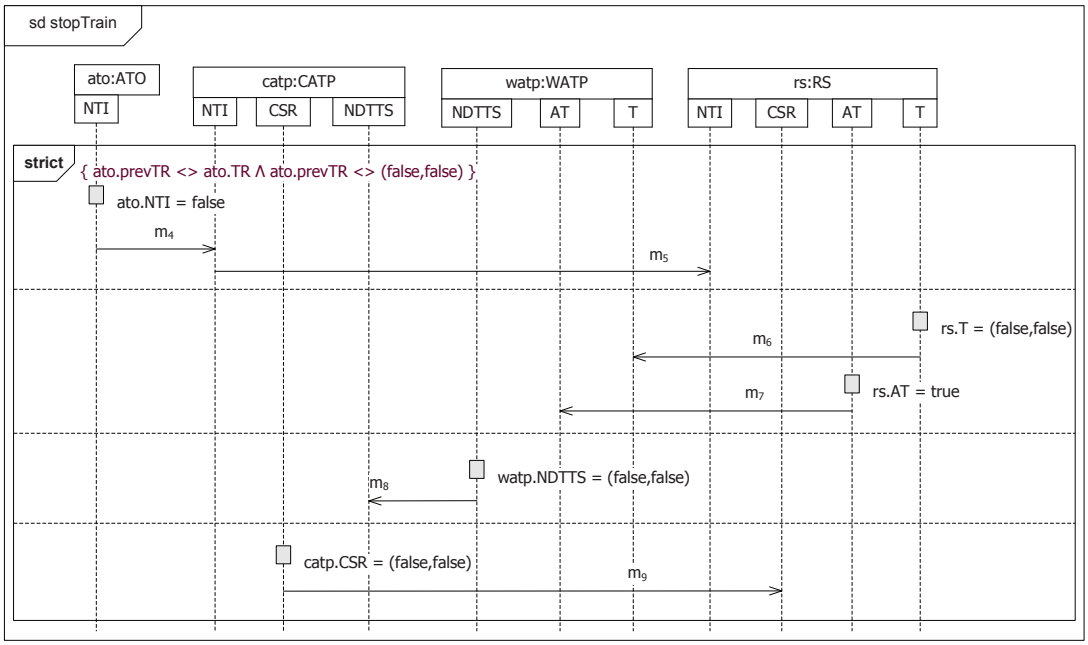


Figure 9.18: Use case: stopping the train

[8] The *strict* operator frame encloses more than two regions, it is a shorthand of nesting successive *strict* operators and forces the enclosed regions to happen one before the other from top to bottom

9.3.3.4 Moving towards the chosen direction of traction

We illustrate in Figure 9.19 the *sd moveDirection*. It specifies how to get the train moving in a given direction of traction.

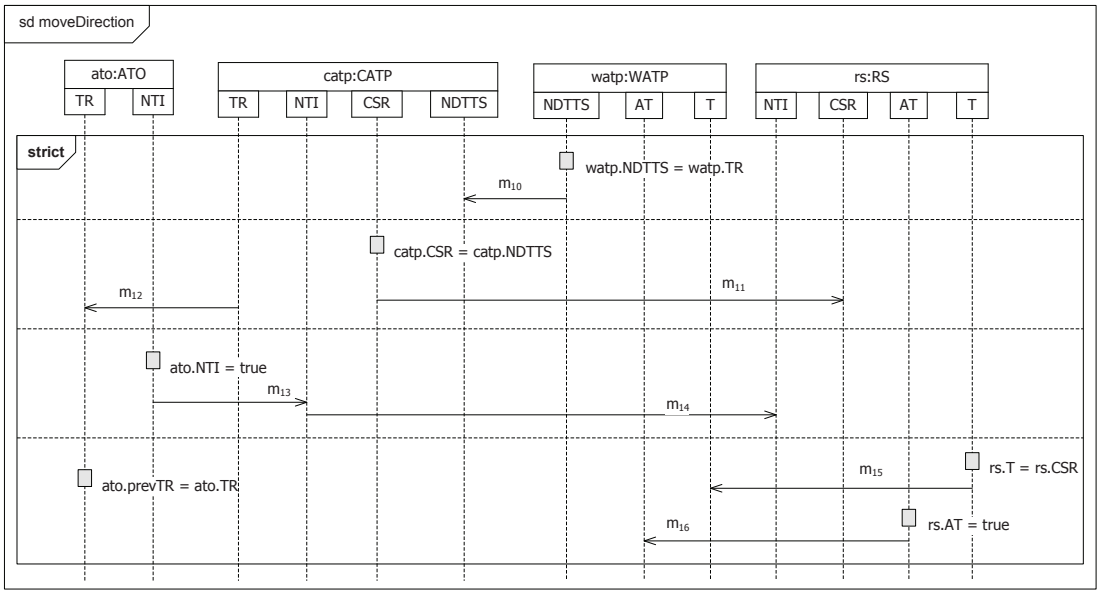


Figure 9.19: Use case: moving towards the chosen direction of traction

Consider the first two regions of the *strict* operator. The component *watp* (on the tracks) states that it is safe to run in the desired direction by updating the value on the port $NDTTS$: The port is assigned with the already received value on the port $watp.TR$ of that same component, so its value now is either $(true, false)$ or $(false, true)$. Then, the *watp* acquaints the *catp* (on the train) of this safe situation (message m_{10}). The latter changes the value on its port CSR accordingly, which denotes the activation of either the front or the rear cabin. Next the *watp* notifies the train rs (message m_{11}) and confirms the direction of traction to the *ato* (message m_{12}). Note that the activation of the adequate cabin occurs necessarily after guarantees are given to realize the traction in the wanted direction safely: here, the $NDTTS$ value is updated before changing the CSR value, thanks to the *strict* operator. This is another safety requirement of the ATC system.

Now consider the last two regions of the *strict* operator. The *ato* decides to release the traction inhibition (the port $ato.NTI$ is assigned to $true$) and informs the *catp* (message m_{13}). The latter authorizes the rs to apply the traction (message m_{14}). After, the rs starts the traction T towards the required direction ($rs.T = rs.CSR$) and then sends the applied traction to the *watp* (message m_{15}). Meanwhile, the *watp* also receives the information that the train is still active (message m_{16}) and the *ato* stores the current applied direction of traction ($ato.prevTR = ato.TR$). Again another safety requirement of the ATC system is implied here : two preconditions must hold in order to get the traction applied in a given direction that are the cabin located on the good side of the train is activated and the traction inhibition is released.

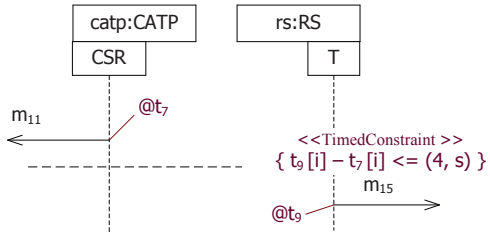
9.3.3.5 Time requirements

The time requirements of the ATC system mainly concern the temporal validity of the messages propagated in the system, to ensure safety. On the other hand, the constraints on the movement and the active state of the train ensure availability.

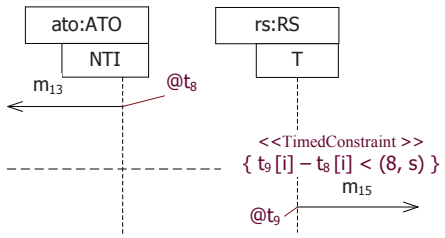
1. The Train is said to be active if it has been moving in the last 4s. The Train should always be active. *(This ensures availability.)*

2. Validity time of messages:

   (a) The clearance to proceed, CSR, issued by *catp* (message m_{11}) is valid only for 4 seconds. That is, the train rs must start moving with that duration (message m_{15}).

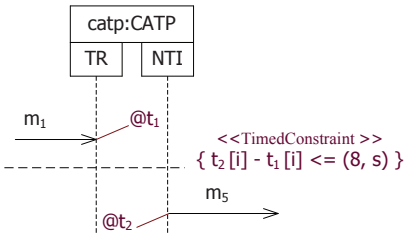


   (b) The request NTI to move (message m_{13}) issued by the ATO is valid for 8 seconds till the train rs actually runs (message m_{15}).

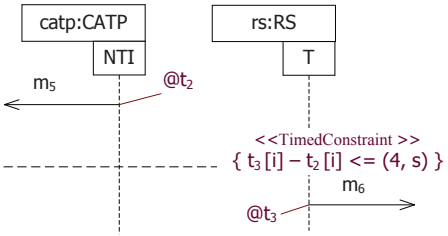


   The train moves in the direction permitted by the carborne ATP if both the CSR message (from cATP) and the NTI message (from ATO) are valid.

3. While changing direction of movement, the instruction NTI (traction inhibition) to brake should be issued by *catp* the carborne ATP (message m_5), within 8 seconds of reception of TR the request to change traction direction (message m_1).

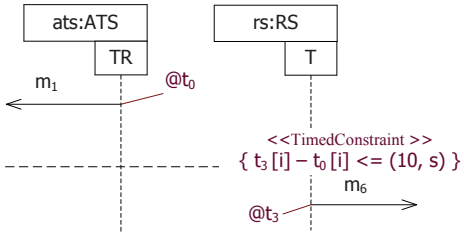


4. The train rs must stop (message m_6) within 4 seconds of the *catp* issuing the instruction NTI to brake (message m_5), traction inhibition.

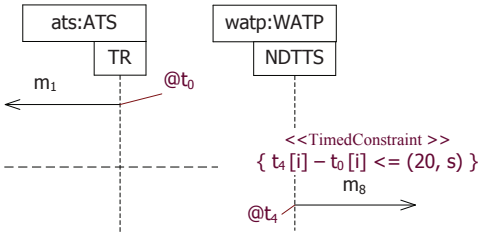


5. Once the ATS has issued instruction TR to change the direction of traction (message m_1),

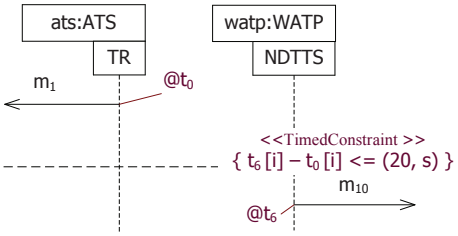
   (a) the train rs should stop moving in the earlier direction of traction T within 10 seconds (message m_6).



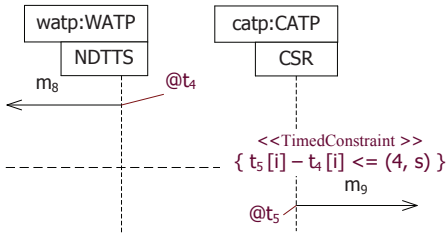
(b) the wayside ATP *watp* should declare the earlier direction unsafe (*NDTTS* through message m_8) within 20 seconds.

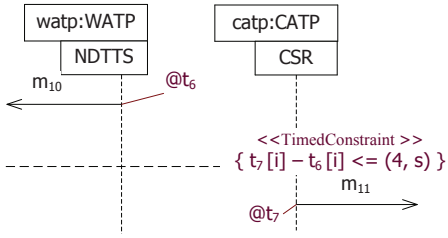


(c) the wayside ATP *watp* should declare the new direction safe for traction (*NDTTS* through message m_{10}) within 20 seconds.

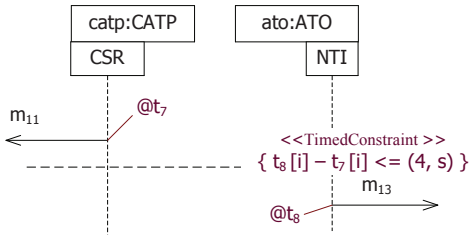


6. After the wayside ATP *watp* declares a direction of traction as safe/unsafe (*NDTTS*), the carborne ATP *catp* should grant/revoke permission *CSR* to move in that direction within 4 seconds.





7. ATO issues instruction *NTI* to move (release brakes, message m_{13}) if the current direction permitted by the carborne ATP (message m_{11}) is valid (and is same as the direction requested by the ATS this is guaranteed in the first modeling part of the ATC system behavior).



9.4 Experiments

In this section, we firstly discuss our experience in modeling component systems behavior as UML MARTE sequence diagrams. Then we comment some collected experimental results obtained by symbolically executing the sequence diagram as being a set of communicating TIOSTS obtained by automatic generation. Those results relate to the coverage of the messages defined by the the sequence diagram. We ground our analysis on the RWC system and the ATC system specified by the sequence diagrams respectively depicted in Figures 4.1–9.15.

9.4.1 Modeling effort

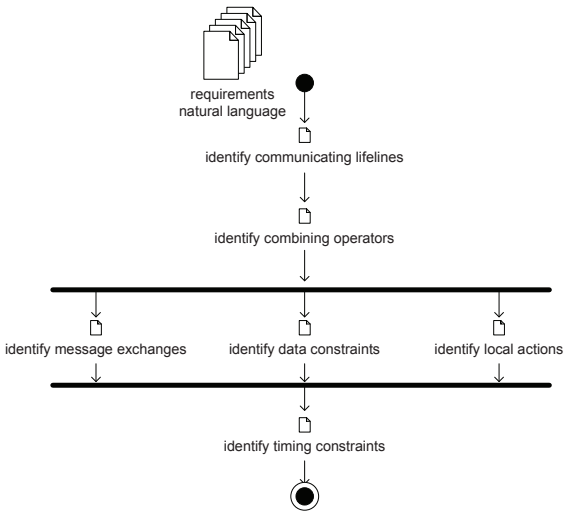
In our context, the modeling effort relates to the identification of the different modeling features used in the sequence diagram from the requirements. We have illustrated the different identification tasks in Figure 9.20a which involve human intervention to analyze the requirement documents. The requirements for the ATC system are given in natural language and using truth tables. The truth tables reflect relations between compatible control requests in the system at different phases of the use case: e.g. "T1 is set to false; T2 remains at false" where for any i in $\{1, 2\}$, Ti is $true$ if the traction is applied in direction i. Message exchanges and timing constraints are expressed in natural language: E.g. "ATP receives a change direction request from ATS and set NTI to false after maximum 8 seconds". Analyzing these kind of excerpts of the requirement document allows us to deduce units of executions on each lifeline. The elaboration of the behavior structuring with the combining operators required however the decomposition of the system overall behavior as sub behaviors: E.g. change direction request, stop the train, activate the cabine, start the traction, etc. At this step the role of each component may not be known yet. Some characteristics of the ATC resulting sequence diagram are given in Figure 9.20b. We give those of the sequence diagram of the RWC system which is our running example through the thesis.

The lifeline number is the number of ports in the component architecture (since each lifeline represents a port). Therefore it already gives an idea about the system size and complexity in terms of concurrent executions at the simulation phase (since each port is associated with its own time scale). What is important to see is that the number of combining operators, especially the non deterministic choice operator (*alt*) and the iteration operator (*loop*), does not increase significantly (4 and 5 operators respectively for the RWC and ATC system) when the system size increases. This is a comforting finding for the modeler and is coherent with the fact that we are at a higher level of abstraction. Of course the number of units of behavior and constraints relating to them is much more important (e.g. going from 4 messages and 3 timing constraints in the RWC system to 16 messages and 10 timing constraints in the ATC system). This phase of the design takes more time but is more straightforward.

9.4.2 Symbolic execution

We first generate TIOSTS from the sequence diagrams *sd RainSensingWiperControl* and *sd reverseDirectionOfTraction* respectively associated with the RWC and the ATC systems using our plug-in `sdToTIOSTS`. Each sequence diagram is then associated with a set of TIOSTS. The goal is to simulate symbolically such specifications. Table 9.2 shows some statistics about these specifications sizes: number of automata (corresponds to the number of lifelines + number of messages), total number of states, total number of transitions. Clearly the growing number of involved ports (represented by the lifelines) makes the specification tend to grow.

Let us start by executing symbolically our running example, the RWC system. Results are shown in Table 9.3. All results are obtained using the Diversity tool.



(a) Modeling tasks

	RWC (academic example)	ATC (industrial example)
lifelines nbr.	5	15
combining operators nbr.	4	5
messages nbr.	4	16
local actions/data constraints nbr.	5	16
timing constraints nbr.	3	10

(b) Statistics about modeling features

Figure 9.20

	automata nbr.	total states	total transitions
RWC	9	64	166
ATC	31	252	656

Table 9.2: Entry automata characteristics

	max. height	max. width	symbolic states	time	transition coverage	covered messages
RWC	7	3577	5000	$2m32s$	$\frac{33}{163}$	$m_1.in$
	8	120736	150 000	$12m50s$	$\frac{48}{163}$	$m_1.in$

Table 9.3: Symbolic execution of RWC

The results in Table 9.3 show the coverage achieved in terms of transition coverage and message coverage computed in a breadth first search manner (BFS). Message coverage is one of criteria defined in the literature for scenarios [6]. It states that any message is covered at least once. We stopped the simulation twice when the number of symbolic states in the tree reached respectively 5000 and 150 000 states. This is because we observed that despite the growing size of the symbolic tree (number of states and also the width), we had only covered the sending of the message m_1 (corresponding to the channel $m.in$ in the TIOSTS). This is explained by the fact that sequence diagrams characterize behaviors which are highly concurrent and hence result in interleaving too many behaviors in between synchronized executions.

However, some of the interleaving is not relevant because it does not affect the behaviors expected locally at the lifeline level (i.e. port level). A typical example is the interleaving of two simple (not timed) assignment actions happening on two different lifelines before a synchronization. For a given lifeline, the other assignment action happening before or after its assignment occurs is irrelevant. In full generality, all transitions with unobservable actions (such that a transition

resulting from the translation of an assignment atom, refer to Chapter 6) and not associated with a time variable or (and) a timing constraint are concerned by this finding. For that purpose, we suggest the following optimization in the symbolic tree computation.

Partially ordered $\tau-$transitions reduction We synchronize unobservable actions performed by concurrent TIOSTS. Intuitively, all $\tau-$transitions, i.e. transitions with an unobservable action τ or with an underspecification action $new(x)$, which can be executed from a "global state" (result from a composition) and belong each one to a different basic TIOSTS are executed together. This results in a single symbolic state in the execution tree. And so two $\tau-$transitions which belong to the same TIOSTS (alternatives or non deterministic) are synchronized each one on its side as described before. Such technique appears in the literature, called *Partial order reduction* (POR) [86]. We use it simply for transitions whose actions are not input nor output actions (i.e. underspecification/unobservable actions) and such that they are not associated with timing features. This guarantees that there is no information loss at the semantical level. We show that we obtain a significant reduction of the size of the symbolic tree.

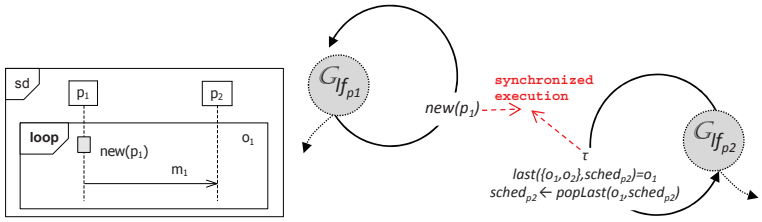


Figure 9.21: $\tau-$transitions reduction

For that purpose, we compare the symbolic execution tree size in Table 9.4 computed in BFS manner before and after the reduction for a simple sequence diagram depicted in Figure 9.21a. The sequence diagram specifies an iterative behavior consisting in two ports p_1 and p_2 exchanging a message m_1: in side the loop region, a new value ($new(p_1)$) is conveyed from p_1 to p_2. We stop the simulation when the unique message in the diagram sending and reception are covered. To achieve the coverage, the simulation without considering the optimization computes a tree of size 5–4–17 (corresponding respectively to height–width–states of the tree). When the optimization is activated, the size of the tree is 4–2–11. That is, the size of the tree decreases.

	max. height	max. width	sym. states	sym. transitions	time (s)	covered messages
sd (Fig.9.21a)	τ reduc.					
	4	2	11	10	≈ 0	$m_1.in$,$m_1.out$
	no reduc.					
	5	4	17	16	≈ 0	$m_1.in$,$m_1.out$

Table 9.4: Symbolic execution of *sd* in Figure 9.21a

In order to comfort these results, we suggest to parametrize the maximum number of the loop iterations. Recall that, according to the sequence diagram semantics, the loop (without a guard) may iterate infinitely many times before leaving the loop region. In fact, the behavior corresponding to any $n \in \mathbb{N}$ iteration of the loop is a subset of all the behaviors specified by such a loop (without guard), in particular it is a subset of the behavior corresponding to $n+1$ iterations. In Table 9.5, we compare the effect of the optimization when the loop may be left after respectively 1 and 2 iterations (this is our new stop criteria).

sd	max. loop itr	max. height	max. width	sym. states	sym. transitions	Time (s)
sd (Fig.9.21a)	1	τ reduc.				
		4	2	11	10	≈ 0
		no reduc.				
		5	4	17	16	≈ 0
	2	τ reduc.				
		10	10	54	64	≈ 0
		no reduc.				
		10	36	135	171	≈ 0

Table 9.5: Symbolic execution: Parametrized loop iteration number

When the maximum iteration number is set to 1, we have the same results as in Table 9.4 where the message was covered once. In the case where the maximum iteration number is 2, the size of the tree decreased from 10–36–135 to 10–10–54. All these first results indicate that our optimization may allow our simulation to scale better.

Large scale simulation Let us carry on now with the rest of the experiments on the RWC and ATC systems. The results of the symbolic execution of both sequence diagram specifications are given in Table 9.6. Symbolic trees were generated by activating the optimization discussed above.

sd	max. height	max. width	sym. states	time	transition coverage	covered messages
RSW	τ reduc. (5000 states)					
	7	4602	5000	$10m23s$	$\frac{68}{166}$	$m_1.in, m_1.out$
	τ reduc. (11 072 states)					
	15	8 973	11 072	$18m50s$	$\frac{137}{166}$	$m_1.in, m_1.out, \ldots, m_4.in, m_4.out$
ATC	τ reduc. (15 000 states)					
	17	12 171	15 000	$21m32s$	$\frac{547}{656}$	$m_1.in, m_1.out, \ldots, m_{16}.in, m_{16}.out$

Table 9.6

The goal is to cover all the messages. This was achieved for the RWC system for a tree of size 15–8 973–11 072 in 18 minutes and 50 seconds. The size of the tree covering all the messages in case of the ATC system is 17–12 171–15 000 computed in 21 minutes and 32 seconds. Let us remark the relatively small jump in the size of the tree between the RWC and ATC systems although the ATC system has more lifelines: ATC has 15 lifelines while RWC has only 5 of them. This difference can be explained by the fact that the sequence diagram of the RWC specifies a highly concurrent behavior with its two nested non-deterministic choice operators (*alt*). On the other hand, the sequence diagram of the ATC system introduces much more synchronization points (with the use of the *strict* operator) which reduces significantly the specified concurrent behaviors. Applied to these two non trivial sequence diagrams in terms of lifelines number and complex structuring operators, the results show a good scalability of our approach.

Conclusion

One of the main challenges of the achieved implementations was mechanizing the link between the model based environment Papyrus where sequence diagrams are designed and the formal tool Diversity where they are symbolically simulated. We have implemented the chain until the symbolic execution and we are currently implementing the projection mechanism. The scalability of our approach is no less important challenge. We obtained promising simulation results for the railways case study which required further computing optimizations in the generation of the symbolic tree.

Chapter 10

Conclusion

Contents

10.1 Thesis summary

We have proposed to use a subset of sequence diagrams with timing annotations to specify behaviors of component-based systems. Models written using sequence diagrams permit to capture the behaviors of systems by focusing on message exchanges sequencing and constraints over them. We have shown how to associate semantics with such models. Traces correspond to sequences of emissions and receptions (over ports occurring in the sequence diagram) separated by durations. In order to define the set of traces associated with a sequence diagram, we have begun by associating them with symbolic automata called Timed Input Output Symbolic Transition Systems (TIOSTS) using translation mechanisms. TIOSTS are extensions of Input Output Symbolic Transition Systems (IOSTS), that we defined by adding timing constraints on transitions. Those timing constraints are inherited from those that can be written using MARTE. TIOSTS semantics is characterized as a set of traces. Semantics of sequence diagrams models are simply semantics of TIOSTS associated with them. We have then shown how to compute such semantics by symbolically executing those TIOSTS associated with sequence diagrams. Symbolic execution permits to characterize, in intention, classes of equivalent traces. Traces are equivalent when they follow the same path of the symbolic execution. A path in a symbolic execution characterizes a path in the TIOSTS together with constraints on data and time, to follow that path in particular. We have defined symbolic execution of TIOSTS from the previously defined symbolic execution of IOSTS and we have extended the symbolic execution engine Diversity to implement TIOSTS symbolic execution. The chain from sequence diagrams to symbolic execution has been implemented by chaining the Papyrus editor for UML based models with the Diversity tool. This chaining has been successfully applied on a large scale railway use case: we have covered all the messages exchanged in the sequence diagram up to a given depth in the symbolic execution tree.

We have also studied how to extract symbolic behaviors of subsystems from the symbolic behaviors of systems. This is done using projection techniques applied to the symbolic executions of sequence diagrams. Such projections characterize behaviors of subsystems as they are constrained by the whole system. We have then proposed to use the conformance relation *tioco* to define the notion of correctness of a system with respect to a sequence diagram. We have been interested in breaking up the testing process of systems into testing separately the components (or subsystems) composing them. For that aim, we have established a new compositionality result for modular testing based on *tioco* which relates the correctness of components to the correctness of the system altogether. What distinguish our result (from [17]) is that the components are tested with respect to specifications obtained by projection. That is, they are tested not for all the behavior they exhibit, but for part of these behavior that are required to realize the system. Our

result holds when the specification of the system satisfies the *local output consistency* property. Intuitively, the property means that the local behavior of a component does not depend on some other component decision, which makes its behavior consistent with all behaviors of the system. The next step has been to relate such results to testing with sequence diagrams. We have shown how to test whether the symbolic tree of a sequence diagram, as being a specification of the whole system behavior, is compliant with the property of local consistency. We have characterized a system under test, realizing some component (or some subsystem), based on observability issues related to underlying communication architecture. The result of compositional testing from sequence diagram flows naturally in this case from those previously established in more general context. Components (or subsystems) are separately tested with respect to their unitary behaviors derived from sequence diagrams. Our result states that any fault of the system, is necessarily a fault of some subsystem and hence can be detected while testing the subsystems in isolation.

10.2 Future work

Optimization of symbolic execution An immediate perspective to our work is to explore more coverage criteria in the generation of the symbolic execution and tune them to our context. What is special about our context is that we reason about symbolic tree projections. Since some behaviors project the same way, we may use partial order reduction method [86]. However, we need to pay attention to handle time properly (in the experiments, we have used such method only on untimed parts of the behaviors which has already given good reduction of symbolic tree size). Besides, another possibility is to use a stop criteria called restriction by inclusion [36]: the symbolic execution stops when the reached state is included, in terms of variables interpretation, in another already encountered state. It is not possible to use this criteria directly as it is formulated with time variables because they are growing structures throughout the execution. We can use it, however, on untimed parts of the behaviors because we have separated the data and time constraints in our context. We plan to investigate this use and how we can apply it in full generality, that is with time variables.

Online testing A natural future work is the adaptation to our context of the runtime testing algorithm in [31] based on *tioco* conformance relation. We suggest this algorithm to check the conformance of a component implementation against its specification obtained by projection. Besides the specification, the algorithm takes as input, behaviors to be tested in order to pilot the testing generation. Such behaviors are called test purposes. Both the specification and the test purpose are characterized as symbolic trees: the specification is the unitary symbolic tree obtained by projection; and the test purpose is a finite sub tree of that unitary tree. The algorithm operates online by acting in a way that maintains the test executions stay within the behaviors specified by the test purpose. The algorithm interacts with a system under test realizing some components, observes its response at any time and whenever it is possible a verdict is emitted (e.g. illegal observed outputs or delays lead to a $FAIL$ verdict). This algorithm needs to be adapted in order to take into account our version of TIOSTS and its associated symbolic execution structure.

Distributed systems Another perspective, is to combine our compositional results with testing in distributed architectures. In such architectures, the decomposition of the system corresponds to the physical deployment of components. In distributed architectures, a tester is placed at each port. Most of the time, testers cannot communicate with each other. Recent works have appeared in the literature [46, 47] defining a new conformance relation *dioco* which compares

local traces of the system under test with projections of the specification, only if the execution reaches quiescent states. Those states are stable in the sense that the implementation cannot perform any output without receiving additional input from some local tester. Authors propose an algorithm to construct local test cases from global test cases satisfying an identified property eliminating some form of nondeterminism induced by distribution. A first step, in this direction, is to characterize a new conformance relation *dtioco* which is the timed extension of *dioco* and then extend Theorem 1 to *dtioco* (note that the work in [46, 47] does not consider compositionality issues).

Refinement Another possible extension to our work, is the refinement of sequence diagram specifications and tracing it throughout the design layers. Refinement may be architectural in terms of further decompositions into subsystems. It may be behavioral: e.g. refining messages into groups of messages while paying particular attention to timing constraints guarding their executions; or eliminating concurrent behaviors and some forms of nondeterminism. Refinement may consist also in making links between the sequence diagram specification of the system and specifications (of parts) of components behaviors expressed with other UML notations such as activity diagrams or state machine diagrams [73].

Bibliography

[1] Marc Aiguier, Pascale Le Gall, Delphine Longuet, and Assia Touil. A temporal logic for input output symbolic transition systems. In *Proc. of APSEC*. IEEE Computer Society, 2005.

[2] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A state-based approach to integration testing based on uml models. *Inf. Softw. Technol.*, 2007.

[3] Rajeev Alur and David L. Dill. The theory of timed automata. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*. Springer, 1992.

[4] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *IEEE Transactions on Software Engineering*. IEEE, 2003.

[5] Wilkerson L. Andrade, Patricia D. L. Machado, Thierry Jéron, and Herve Marchand. Abstracting time and data for conformance testing of real-time systems. In *Proceedings ICSTW Workshops*. IEEE, 2011.

[6] Anneliese Andrews, Robert France, Sudipto Ghosh, and Gerald Craig. Test adequacy criteria for uml design models. *Journal of Software Testing, Verification and Reliability*, 2003.

[7] Farhad Arbab and Sun Meng. Synthesis of connectors from scenario-based interaction specifications. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08. Springer-Verlag, 2008.

[8] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. Springer, 2002.

[9] Paul Baker, Paul Bristow, Clive Jervis, David King, and Bill Mitchell. Automatic generation of conformance tests from message sequence charts. In *Proc. of the international conference on Telecommunications and beyond: the broader applicability of SDL and MSC*. Springer, 2003.

[10] Paul Baker, Paul Bristow, Clive Jervis, David King, and Bill Mitchell. Automatic generation of conformance tests from message sequence charts. In *SAM*. Springer, 2003.

[11] Paul Baker, Clive Jervis, and David King. An optimised algorithm for test script generation. In *Patent GB18137.0*, 2000.

[12] Boutheina BANNOUR, Christophe GASTON, and David SERVAT. Eliciting unitary constraints from timed Sequence Diagram with symbolic techniques: application to testing. In *Proc. of APSEC*. IEEE, 2011.

[13] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cowsuite approach to planning and deriving test suites in uml projects. In *Proc. of International Conference on The Unified Modeling Language*. Springer, 2002.

[14] Gerd Behrmann, Alexandre David, and Kim Gulstrand Larsen. A tutorial on Uppaal, 2005.

[15] bell labs. Ubet, 1999. http://cm.bell-labs.com/cm/cs/what/ubet/.

[16] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 1991.

[17] Machiel Van Der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco, 2003.

[18] Sergiy Boroday, Alexandre Petrenko, and Andreas Ulrich. Implementing msc tests with quiescence observation. In *Proc. of FATES*. Springer, 2009.

[19] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *Proc. of the International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Springer, 2001.

[20] Laura Brandan Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *FATES 04*. Springer, 2004.

[21] Laura Brandan Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *In Proc. of FATES*. Springer-Verlag, 2004.

[22] Ana Cavalli, David Lee, Christian Rinderknecht, and Fatiha Zaidi. Hit-or-jump: An algorithm for embedded testing with applications to in services, 1999.

[23] NASA Ames Research Center. http://babelfish.arc.nasa.gov/trac/jpf.

[24] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts, 2001.

[25] Haitao Dan and Robert M. Hierons. Conformance testing from message sequence charts. *ICST*, 2011.

[26] Bahrami Diane, Faivre Alain, and Lapitre Arnault. Diversity/tg automatic test case generation from matlab/simulink models. http://www.erts2012.org/Site/0P2RUC89/8B-2.pdf.

[27] T. Dinh-Trong. *A systematic approach to testing UML designs*. PhD thesis, Ph.d. thesis, Colorado State University, 2007.

[28] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test uml design models. In *Proc. of the International Symposium on Software Reliability Engineering*. IEEE Computer Society, 2006.

[29] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 2002.

[30] Jose Pablo Escobedo, Christophe Gaston, Pascale Le Gall, and Ana Cavalli. Testing web service orchestrators in context: A symbolic approach. *Proc. of SEFM*, 2010.

[31] Jose Pablo Escobedo, Christophe Gaston, and Pascalle Le Gall. Timed Conformance Testing for Orchestrated Service Discovery. In *Proc. of FACS*. Springer, 2011.

[32] J.P. Escobedo, P. Le Gall, C. Gaston, and A. Cavalli. Observability and controllability issues in conformance testing of web service composition. In *Proc. of TestCom/FATES*. Springer, 2009.

[33] A. Faivre, C. Gaston, and P. Le Gall. Symbolic Model Based Testing for Component oriented Sytems. In *Proc. of TestCom*. Springer, 2007.

[34] Alain Faivre, Christophe Gaston, Pascale Gall, and Assia Touil. Test purpose concretization through symbolic action refinement. In *Proc. of TestCom/FATES*. Springer, 2008.

[35] Falk Fraikin and Thomas Leonhardt. Seditec " testing based on sequence diagrams. In *Proc. of ASE*. IEEE Computer Society, 2002.

[36] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Proc. of TestCom*. IEEE Computer Society, 2006.

[37] Ankit Goel. *Parametrized validation of UML like models for reactive embedded systems*. PhD thesis, Ph.d. thesis, National University Of Singapore, 2009.

[38] Object Management Group. A uml profile for marte: Modeling and analysis of real-time embedded systems, ccsl. 2009. http://www.omg.org/spec/MARTE/.

[39] Object Management Group. A uml profile for marte: Modeling and analysis of real-time embedded systems, gcm. 2009. http://www.omg.org/spec/MARTE/.

[40] Object Management Group. A uml profile for marte: Modeling and analysis of real-time embedded systems, hlam, 2009. http://www.omg.org/spec/MARTE/.

[41] Object Management Group. A uml profile for marte: Modeling and analysis of real-time embedded systems, vsl, 2009. http://www.omg.org/spec/MARTE/.

[42] David Harel. Statecharts: A visual formalism for complex systems, 1987.

[43] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. Springer-Verlag, 2000.

[44] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Proc. of In formal methods in software and system modeling*. Springer, 2005.

[45] Øystein Haugen. Comparing uml 2.0 interactions and msc-2000. In *SAM*. Springer, 2004.

[46] Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Controllable test cases for the distributed test architecture. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*. Springer, 2008.

[47] Robert M. Hierons, Mercedes G. Merayo, and Manuel Nunez. Scenarios-based testing of systems with distributed ports. In *Proceedings of the 2010 10th International Conference on Quality Software*. IEEE, 2010.

[48] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[49] ITU-T. *ITU-T Rec. Z.100 – Formal description techniques (FDT) – Specification and Description Language (SDL)*, 2002.

[50] Telecommunication Standardization Sector (ITU-T). http://www.itu.int/ITU-T/.

[51] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1997.

[52] J.-C. King. A new approach to program testing. *Proc. of the international conference on Reliable software, Los Angeles, California*, 1975.

[53] Alexander Knapp and Jochen Wuttke. Model checking of uml 2.0 interactions. In *Proc. of MoDELS*. Springer, 2006.

[54] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *In 11th International SPIN Workshop on Model Checking of Software*. Springer, 2004.

[55] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 2009.

[56] Ingolf Kruger, Radu Grosu, Peter Scholz, and Manfred Broy. From mscs to statecharts. In *International workshop on distributed and parallel embedded systems*. Kluwer Academic Publishers, 1998.

[57] Ingolf H. Kruger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Ph.d. thesis, Technische Universitat Munchen, 2000.

[58] Ingolf H. Kruger, Diwaker Gupta, Reena Mathew, Praveen Moorthy, Walter Phillips, Sabine Rittmann, and Jaswinder Ahluwalia. Towards a process and tool-chain for service-oriented automotive software engineering. In *Proc. of ICSE Workshop SEAS*, 2004.

[59] Hillel Kugler, Michael J. Stern, and E. Jane Albert Hubbard. Testing scenario-based models. FASE'07. Springer, 2007.

[60] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *In Proc. of the 4th Intl. Workshop on Formal Approaches to Testing of Software*. Springer, 2004.

[61] Kim Guldstrand Larsen, Shuhao Li, Brian Nielsen, and Saulius Pusinskas. Scenario-based analysis and synthesis of real-time systems using uppaal. In *Proc. of DATE*. IEEE Computer Society, 2010.

[62] CEA LIST. Papyrus uml/open source tool for graphical uml2 modeling. http://www.eclipse.org/modeling/mdt/papyrus/.

[63] Mass Soldal Lund. *Operational analysis of sequence diagram specifications*. PhD thesis, Ph.d. thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2008.

[64] Mass Soldal Lund and Ketil Stølen. Deriving tests from uml 2.0 sequence diagrams with neg and assert. In *Proc. of AST*. ACM, 2006.

[65] Mass Soldal Lund and Ketil Stølen. Fm 2006: Formal methods, 14th international symposium on formal methods, hamilton, canada, august 21-27, 2006, proceedings. In *FM*. Springer, 2006.

[66] Mass Soldal Lund and Ketil Stølen. A fully general operational semantics for uml 2.0 sequence diagrams with potential and mandatory choice. In *Proc. of Int. Conf. FM*. Springer, 2006.

[67] Frédéric Mallet and Charles Andre. On the semantics of uml/marte clock constraints. In *In Proc. of ISORC*. IEEE, 2009.

[68] S. Mauw and M.A. Reniers. High-level message sequence charts, 1997.

[69] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems*. PhD thesis, Ph.d. thesis, University of California, 1974.

[70] Zoltan Micskei and Helene Waeselynck. The many meanings of uml 2 sequence diagrams: a survey. *Software and Systems Modeling*, 2011.

[71] Ashalatha Nayak and Debasis Samanta. Automatic test data synthesis using uml sequence diagrams. *Journal of Object Technology*, 2010.

[72] Dinh-Phuc Nguyen, Chung-Tuyen Luu, Anh-Hoang Truong, and Norbert Radics. Verifying implementation of uml sequence diagrams using java pathfinder. In *Proc. of Int. Conf. KSE*. IEEE, 2010.

[73] OMG. Omg unified modeling languge (omg uml), superstructure version 2.2, 2009. http://www.omg.org/spec/UML/2.2/Superstructure.

[74] Object Management Group (OMG). http://www.omg.org/.

[75] Simon Pickin, Claude Jard, Thierry Jeron, Jean-Marc Jezequel, and Yves Le Traon. Test synthesis from uml models of distributed software. *IEEE Trans. Softw. Eng.*, 2007.

[76] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois. Behavioural unfolding of formal specifications based on communicating automata. In *Proc. of first Workshop on Automated technology for verification and analysis*, 2003.

[77] Atanas Rountev, Scott Kagan, and Jason Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *FASE*, Lecture Notes in Computer Science. Springer, 2005.

[78] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. Symbolic message sequence charts. In *Proc. of European Conf. ESEC-FSE*. ACM, 2007.

[79] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. Symbolic message sequence charts. *Proc. of TOSEM*, 2011.

[80] Vlad Rusu, Lydie Du Bousquet, and Thierry Jeron. An approach to symbolic test generation. In *In Proc. Integrated Formal Methods*. Springer Verlag, 2000.

[81] J. Schmaltz and J. Tretmans. On Conformance Testing for Timed Systems. In *Proc. of FORMATS*. Springer, 2008.

[82] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[83] Sebastian Uchitel. *Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios*. PhD thesis, Ph.d. thesis, University of London, 2003.

[84] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 2004.

[85] Andreas Ulrich, El-Hachemi Alikacem, Hesham H. Hallal, and Sergiy Boroday. From scenarios to test implementations via promela. In *Proc. of Int. Conf. ICTSS*. Springer, 2010.

[86] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*. Springer, 1991.

[87] Sabrina Von Styp, Henrik Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. FORMATS. Springer-Verlag, 2010.

[88] Tao Wang, Abhik Roychoudhury, H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *Proc. of PADL*. Springer, 2004.

[89] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. ACM Press, 2000.

[90] Jeremiah Wittevrongel and Frank Maurer. Using uml to partially automate generation of scenario-based test drivers, 2001.

[91] Fei Xie and James C. Browne. Verified systems by composition from verified components. In *Proc. of ESEC*. ACM, 2003.

[92] Meixia Zhu, Hanpin Wang, Yongzhi Cao, Zizhen Wang, and Wei Jin. The analysis of sequence diagram with time properties in qualitative and quantitative aspects by model transformation. In *Proc. of APSEC*. IEEE Computer Society, 2010.