On the mechanized verification of the meta-theory of contracts and its instantiation to differential dynamic logic

Stéphane Kastenbaum

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES

ECOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Electronique
Spécialité : *Informatique*

Par
Stéphane KASTENBAUM

On the mechanized verification of the meta-theory of contracts and its instantiation to differential dynamic logic

Thèse présentée et soutenue à Rennes, le 5 mai 2023
Unité de recherche : Inria, Centre Inria Rennes-Bretagne Atlantique (Inria-Rennes)

Rapporteurs avant soutenance :

Jérôme Hugues	Directeur de Recherche à Carnegie Mellon Institute
Dominique Merry	Professeur à Télécom Nancy

Composition du Jury :

Président :	Jérôme Hugues	Directeur de Recherche à Carnegie Mellon Institute
Examinateurs :	Dominique Merry	Professeur à Télécom Nancy
	Régine Laleau	Professeure à l'Université Paris-Est Créteil
Dir. de thèse :	Jean-Pierre Talpin	Directeur de Recherche à Inria
Enc. de thèse :	Benoit Boyer	Ingénieur de Recherche à Mitsubishi Electric R&D Centre Europe

Abstract

The rise of complexity and heterogeneity of safe-critical systems that surround us poses a challenge to existing workflows employed to design them. Critical components of such systems need to be designed with precision, and their safety must be ensured before they are made available to the public.

However, reasoning about an entire, possibly large, complex, heterogeneous system based on such precise component specifications is certainly not humanly possible. Moreover, the cost associated with building, testing and possibly rebuilding systems pushes designers to model a system beforehand and to validate models as thoroughly as possible before actually building the system.

Modeling systems is a difficult task prone to errors, formal methods helps the designer to validate their soundness and safety. The complexity of the systems leaves the designer no choice but to have a component-based approach, namely, to separate the task of modeling a system into multiple subtasks (components) and combine them afterward (composition).

An effort have been made to construct formal methods to validate large and complex systems andto allow the combined specification and validation of components of cyber-physical systems. A central difficulty of such validation frameworks, or workflows, is to verify the safety of the global system from the validated specifications of heterogeneous, individual components.

Contract theory addresses this problem. It starts from assume/guarantee (A/G) contracts as specifications of components. An individual contract can be validated against a component by verifying that its assumptions and guarantees over-approximate the pre- and post-conditions that would result from valid evaluations of the component model. Then, contracts of individual components can be composed, by verifying that the assumptions of one match the guarantees of the others.

This first part of this manuscript defines an algebraic formalization of assume/guarantee contracts that is formally proven to validate a meta-theory of contract. It is defined as a type class and parameterized by a logic, in order to maximize the scope of its applicability. To demonstrate this capability, our formalization is instantiated with Platzer's differential dynamic logic, in order to provide a fully verified contract theory for modeling cyber-physical systems.

Our model of assume/guarantee contracts is entirely implemented in the calculus of construction of the proof assistant Coq. It comprises the algebraic operators of composition, conjunction, abstraction, refinement, variables intro-

duction and elimination.

The second part of the manuscript describes a practical use case for the definition of both the meta-theory and parametric A/G contracts models. It proceeds by instantiating the contract model with differential dynamic logic and by considering two instances of the contracts model in this logic: with hybrid and abstract $\mathbf{d}\mathcal{L}$ programs.

Finally, we exercise the use of both instances on a typical case-study to illustrate the power of the theory to model components and their contractual abstractions.

Résumé en français

Les systèmes cyber-physiques (SCP) sont des assemblages complexes de composants matériels et logiciels en réseau, conçus pour détecter, évaluer et réagir aux changements de leur environnement physique. Leur nature hétérogène et leur échelle étendue les rendent incroyablement complexes à modéliser avec précision, ce qui représente un véritable défi pour les ingénieurs et les chercheurs. La variété de technologies impliquées, allant des capteurs aux actionneurs en passant par les systèmes de traitement de données, rend la tâche de compréhension et de modélisation des interactions entre ces différents éléments encore plus ardue. En outre, les SCP sont souvent responsables de missions critiques, comme les systèmes de transport autonomes ou les infrastructures de santé, ce qui ajoute une pression supplémentaire pour garantir leur fiabilité et leur sécurité.

La vérification formelle des SCP est donc d'une importance capitale pour s'assurer de leur bon fonctionnement et de leur sécurité. En utilisant des méthodes formelles, il est possible de fournir des garanties mathématiques sur le comportement du système et de détecter les éventuelles erreurs dès les phases de conception, évitant ainsi des problèmes coûteux et dangereux à l'avenir. Cependant, en raison de la complexité et de la taille des SCP, la vérification formelle devient rapidement difficile, voire impossible, sans une approche adéquate.

C'est ici que l'abstraction des modèles se révèle être une approche cruciale pour rendre la vérification formelle réalisable pour les SCP. En utilisant l'abstraction, les ingénieurs peuvent simplifier la représentation des composants et des interactions, en se concentrant sur les aspects essentiels pour la vérification tout en masquant les détails complexes. Cela permet de réduire la complexité globale du système et de faciliter la vérification modulaire, en traitant chaque composant de manière individuelle avant de les combiner pour former le système complet.

Pour répondre à cette complexité et faciliter la vérification modulaire, les concepteurs de SCP adoptent souvent une approche basée sur les composants. Cette approche consiste à diviser le système en sous-systèmes ou composants individuels, chacun étant responsable d'une partie spécifique du comportement global. En validant chaque composant individuellement, les ingénieurs peuvent s'assurer que les propriétés requises sont respectées avant de les combiner pour former le système complet. Cette approche "diviser pour mieux régner" permet également de gérer la complexité en gérant chaque partie du système séparément,

simplifiant ainsi la vérification et la maintenance à grande échelle.

Des efforts considérables ont été déployés pour développer des méthodes formelles et des méthodologies de conception pour les SCP, en particulier pour la vérification combinée de composants discrets et continus. Cela implique de mettre en place des cadres de validation solides qui permettent de spécifier et de vérifier les comportements individuels des composants, puis de les composer pour former le système global. Les contrats, dans ce contexte, ont émergé comme une approche prometteuse pour concevoir et vérifier des systèmes complexes en utilisant des abstractions. Ils permettent de spécifier les comportements attendus des composants sous forme d'hypothèses et de garanties, abstrayant ainsi leur implémentation interne. Cela facilite la vérification modulaire et la composition des composants pour former le système global. En outre, les contrats offrent une certaine flexibilité en permettant d'utiliser différentes logiques pour définir les comportements des composants, ce qui les rend adaptables à une grande variété de SCP.

Les contrats d'assomption/garantie abstraient les modèles de composants pour fournir un processus de vérification solide, structuré et modulaire : un contrat individuel peut être validé par rapport à un composant en vérifiant que ses hypothèses et ses garanties se rapprochent de la sémantique du composant par les pré-conditions et les post-conditions qui résulteraient d'évaluations valides du modèle du composant. Ensuite, les contrats des composants individuels peuvent être composés en vérifiant que les hypothèses de l'un correspondent aux garanties des autres.

Alors que les abstractions de modèles par des contrats sont généralement prouvées concordant avec les modèles (sémantique) des composants, aucune théories de contrats n'a, à notre connaissance, été formellement prouvé correcte jusqu'à présent. Dans ce but, nous présentons la formalisation d'une théorie générique de contrat assomption/garantie dans l'assistant de preuve Coq. Nous identifions et prouvons les théorèmes qui garantissent son exactitude. Notre théorie est générique, ou paramétrique, en ce sens qu'elle peut être instanciée et utilisée avec n'importe quelle logique donnée, en particulier les logiques hybrides, dans lesquelles des systèmes cyber-physiques très complexes peuvent être décrits de manière uniforme.

La première partie de ce manuscrit définit la formalisation algébrique des contrats d'assomption/garantie qui est formellement prouvée pour valider une méta-théorie des contrats. Ce travail se base sur la théorie des contrats assomption/-garantie et de la méta-théorie des contrats définie par Benveniste et al. [Ben+15b]. Dans ces travaux, la méta théorie des contrats donne un cadre pour définir un d'autres théorie de contrats, dont la théorie des contrats assomption/-garantie. Dans ce cadre la théorie assomption/garantie, définit les contrats comme l'assemblage d'une assomption sans laquelle le composant n'a pas de fonctionnement définit, et une garantie qui est assuré par le composant (sous condition que son assomption soit validée). Formellement, les composants, assomptions et garanties sont définis comme des ensembles de comportements hypothétiques du système. Les relations de compositions, conjonctions, raffinements et autres sont définis comme des opérations sur ces ensembles tels

que l'union, l'intersection ou le complémentaires. Avec cette approche, on peut instancier la théorie avec des logiques différentes en changeant la définitions des comportements.

Notre modèle de contrats assomption/garantie, conçu pour garantir la sûreté et la fiabilité des systèmes cyber-physiques, a été entièrement implémenté dans l'environnement de preuve Coq, qui offre une plateforme puissante pour la vérification formelle. Cette implémentation comprend une gamme complète d'opérateurs algébriques essentiels, tels que la composition, la conjonction, l'implémentation et le raffinement, permettant ainsi de spécifier les contrats de manière formelle et expressive. L'un des avantages clés de notre modèle est la flexibilité qu'il offre grâce à la paramétrisation de la logique comme une classe de type. Cette approche permet une modularité adaptée à la théorie, permettant aux concepteurs de choisir la logique la mieux adaptée à leurs besoins spécifiques. Cela signifie que notre modèle peut être facilement adapté à différentes logiques, y compris les logiques hybrides, qui sont couramment utilisées pour décrire les systèmes cyber-physiques complexes. Cette flexibilité fait du modèle de contrats assomption/garantie un outil polyvalent pour la vérification formelle dans une grande variété de domaines. Après avoir mis en place notre modèle de contrats assomption/garantie, nous avons procédé à une validation rigoureuse en le confrontant à la méta-théorie des contrats. Cette étape cruciale garantit la cohérence et la solidité de notre approche en s'assurant que les propriétés fondamentales des contrats sont respectées et préservées tout au long du processus de spécification et de vérification. Cette validation approfondie renforce la confiance dans l'utilisation de notre modèle pour la vérification formelle des systèmes cyber-physiques. Par souci de transparence et de partage avec la communauté scientifique, nous avons rendu tout le code associé à notre modèle de contrats assomption/garantie disponible en accès libre. Vous pouvez consulter et accéder à l'ensemble du code à l'adresse suivante : `https://gitlab.inria.fr/skastenb/differential-contracts`. Cette initiative de mise à disposition du code source vise à encourager la collaboration, les discussions scientifiques et à faciliter les travaux futurs dans le domaine de la vérification formelle des systèmes cyber-physiques.

L'une des fonctionnalités clés de notre modèle est le mécanisme d'extension de l'alphabet, qui joue un rôle essentiel dans la définition des contrats. Ce mécanisme permet de définir les contrats sur un ensemble spécifique de variables, l'alphabet, qui capture les aspects pertinents du comportement du système. La validation de ce mécanisme garantit la robustesse de notre approche et son adéquation aux contraintes du monde réel, où la spécification des contrats doit souvent être limitée à un sous-ensemble des variables du système pour des raisons de complexité et de performance.

En outre, nous avons également implémenté le processus d'élimination des variables, qui permet d'abstraire les contrats en supprimant certaines de leurs variables. Cette fonctionnalité est précieuse pour simplifier la vérification en se concentrant sur les aspects essentiels des contrats, tout en masquant les détails moins pertinents. L'efficacité de cette abstraction est cruciale pour gérer la complexité croissante des systèmes cyber-physiques tout en maintenant un niveau

élevé de confiance dans leur comportement vérifié.

Enfin, pour faciliter l'utilisation et l'instanciation de notre théorie des contrats assomption/garantie, nous avons créé une interface qui permet aux concepteurs de spécifier la logique souhaitée en tant que paramètre. Cette interface exige que la logique donnée soit basée sur des expressions dont l'alphabet est explicite pour chaque formule et que l'ensemble des expressions soit clos par les relations de négations, de conjonctions et de disjonctions. Cette approche modulaire facilite l'intégration de différentes logiques dans notre modèle de contrats, offrant ainsi une solution flexible et adaptable pour la vérification formelle des systèmes cyber-physiques dans divers domaines d'application.

La deuxième partie du manuscrit décrit un cas pratique d'utilisation de la définition des modèles de méta-théorie et de contrats A/G paramétriques. Il s'agit de l'instanciation des contrats assomption/garantis avec une logique dynamique différentielle comme paramètre. Cela est prouvé en considérant le modèle des programmes hybrides et les programmes abstraits, comme deux instances distinctes pour modéliser les composants concrets et les contrats dans la logique.

La première instance considère les programmes hybrides pour modéliser les composants. Pour définir l'instance, nous devons déterminer la valeur de deux types : `value` et `ident`. Le type `value` correspond au domaine de définitions des variables tandis que `ident` correspond au type des identifiants de ces variables. Dans Coq dL, les variables sont identifiés avec un type ad-hoc : `KAssignable` et ont leur valeur dans les réels `R`. C'est donc avec ces types que l'on définit `ident` et `value`. Dans cette instance, les programmes hybrides définit dans la théorie sont légèrement modifiés pour inclure dans leur type l'alphabet sur lequel ils sont définis. C'est-à-dire, que l'on définit un prédicat qui assure que toutes les variables d'un programme hybride sont incluse dans l'alphabet des variables. Pour cela on utilise les mécanismes de sémantiques statiques pour extraires toutes les variables liées (écrites par le programme) et les variables libres (lus par le programme), pour vérifier qu'elles sont bien présente dans l'alphabet dans l'alphabet. L'instance est validée par la preuve d'un théorème reliant la relation de raffinement définie pour la logique dynamique différentielle à la relation de raffinement pour les composants dans la théorie des contrats d'assomption/garantie. Cette preuve utilise plusieurs lemmes qui ont été développé pour vérifier la sémantique de substitution uniforme et qui ont été implémenté dans Coq dL. Parmis ces lemmes il y a le lemme de la coincidence qui stipule que deux programmes étant égaux sur toutes les variables libres et liées, sont aussi égaux sur les variables non-libres et non-liées. La preuve utilise aussi le lemme d'effet liées, qui exprime qu'un programme ne modifie pas les variables non-liées.

La deuxième instance définit des programmes abstraits : des programmes hybrides réduits pour décrire des pré-conditions et des post-conditions. Elle se traduit en programmes hybrides, c'est-à-dire que l'ensemble des programmes abstraits est un sous-ensemble des programmes hybrides. Les programmes abstraits ont l'avantage d'être fermés par la conjonction et la négation. Nous prouvons que cette traduction ou concrétisation est correct par rapport à la satisfiabilité. L'instanciation de la théorie des contrats avec des programmes

abstraits nous donne des contrats définis avec des programmes abstraits en tant qu'hypothèses et garanties. Les contrats sont accompagnés des opérateurs définis dans la théorie, à savoir la conjonction, le raffinement et la composition. L'implémentation d'un composant de contrat par un contrat est également définie entre les composants définis comme des programmes hybrides et les contrats définis avec des programmes abstraits.

Enfin, nous exerçons l'utilisation des deux instances sur une étude de cas typique afin d'illustrer la puissance de la théorie pour modéliser les composants et leurs abstractions contractuelles. L'exemple choisit est celui de la cuve d'eau dont l'arrivée d'eau est réglé par une valve. La valve ajuste le flot d'arrivée en fonction de la hauteur de l'eau dans la cuve. Le système doit assurer que la cuve ne déborde pas lors de son fonctionnement. Nous définissons les composants et leur spécification dans la théorie de l'assomption/garantie des contrats, et nous montrons les obligations de preuve nécessaires pour prouver la cohérence du modèle résultant. Cette étude de cas démontre l'intérêt pratique de notre approche pour valider formellement des systèmes cyber-physique de large échelle.

Á Léandre et Gwenaëlle

Remerciement

Tout d'abord, je me dois de remercier mon directeur de thèse Jean-Pierre Talpin, merci de m'avoir soutenu, et d'avoir été patient lorsque je me perdais dans les méandres de la recherche. Merci aussi à Benoit Boyer, mon co-encadrant de thèse qui m'a accompagné pendant ces trois années de recherches avec ses conseils avisés et son soutien indéfectible.

J'aimerais aussi remercier Régine Laleau d'avoir accepté de faire partie de mon jury, ainsi que Jérome Hughes et Dominique Mery d'avoir été les rapporteurs. Merci à vous trois d'avoir posé votre regard critiques et bienveillant sur mes travaux de thèse.

Je remercie aussi Patrice Quinton et Khalil Ghorbal, qui forment à eux deux mon comité de soutient individuel. Nos réunion m'ont été précieuses, vous avez su me remettre sur les rails lorsque j'en avais besoin.

Ma thèse s'est faite en garde alterné au centre de R&D Mitsubishi Electric, ainsi je remercie autant ceux qui m'ont permis de faire cette thèse: David, Luc, Magalie et Marie, que mes collègues : David, Denis, Florian, François, Enzo et Emilie.

Merci aussi à mon équipe Inria, Tea, Liang Cong, Lucas, Jean-Joseph, Shenghao et Benjamin. Notre équipe était petite mais soudée, ce fut un plaisir de travailler avec vous et d'échanger nos problèmes au quotidien. En plus de mon équipe, j'aimerais remerceier mes autres collègues de l'IRISA. Je remercie du fond du cœAxel qui m'a fait l'honneur d'emménager dans mon bureau. Merci aussi à ceux qui était au bout du couloir, qui on bien voulu que je les force à faire des pauses régulières Jérémy, Kilian et Rémi. Merci aussi à ceux qui m'ont prêté leurs bureau lorsque j'en pouvais plus du mien : Théo, Thomas, Adèle et Alberto. Merci à ceux qui m'ont accueilli quand je voulais taper l'incruste dans les pauses cafés de leurs équipes : Katja, Salomé, Élodie, Louise, Guéno, Thibault, Hasnaa, Valentin, Benjamin et Sony.

Je ne pourrais jamais remercier assez ma famille. Sans elle je ne serais nulle part, et plus j'avance dans ma vie plus je me rends compte la chance que j'ai. Je remercie mes parents évidemment, mes grands-parents, mes frères et ma soeur, mes tantes et mon oncle, et mes cousins et cousines.

Merci à tous mes amis, que j'ai rencontré tout au long de ma vie. Je remercie particulièrement Pierre, qui m'a offert des beaux premiers pas dans la recherche. Je remercie aussi Jean-Baptiste qui a été cette voix qui donne les bons conseils au bon moment.

L'accueil que j'ai reçu à Rennes a été formidable, je remercie Bertrand de m'avoir accueilli chez lui les premiers mois. Je remercie mes amis de Borderline Games, pour les moments de rire autour du jeu *en* société. Et enfin, merci à Lucie pour les cheveux roses.

Merci à tout le monde.

Contents

Chapter 1

Introduction

1.1 The Rise of Complexity in System Design

Cyber-physical systems (CPS) are assemblies of networked, heterogeneous, hardware, and software components sensing, evaluating, and actuating a physical environment. This heterogeneity and scale of CPSs induces complexity that makes them challenging to even model correctly, let alone verifying them.

Since CPSs are often entrusted critical missions, it is however of utmost importance to formally verify them in order to provide them with the highest guarantees of safety. Faced with CPS complexity, model abstraction becomes paramount to make verification attainable.

The complexity of CPS systems leaves designers no choice but to implement a component-based approach, that is, divide and conquer: to divide the task of modeling a system into that of multiple subtasks (components), to validate them individually (separation of concerns), and to combine them afterward (composition).

Tremendous efforts have been made to develop formal methods, formally defined design methodologies, to validate large and complex systems and allow the combined specification and validation of discrete and continuous components of cyber-physical systems. A central difficulty of such validation frameworks, or workflows, remains that of verifying the safety of the system from the validated specifications of heterogeneous, individual components.

1.2 Contracts for Modular System Design

Contract theory addresses this problem. Contracts help to design and to verify complex systems by abstracting component models, using their assumptions and guarantees in place of their exact, internal specification. A contract being more abstract than the internal specification of a component makes the modular verification of large systems feasible.

Assume/guarantee contracts abstract component models to provide a sound, structured, and modular verification process: an individual contract can be validated against a component by verifying that its assumptions and guarantees approximate the component's semantic by the pre- and post-conditions that would result from valid evaluations of the component model. Then, contracts of individual components can be composed, by verifying that the assumptions of one match the guarantees of the others.

While abstractions of models by contracts are usually proved sound with respect to the model (semantic) of components, none of the related contract frameworks themselves have, to the best of our knowledge, been formally proved correct so far. In this aim, we present the formalization of a generic assume/guarantee contract theory in the proof assistant Coq. We identify and prove theorems that ensure its correctness. Our theory is generic, or parametric, in that it can be instantiated and used with any given logic, in particular hybrid logics, in which highly complex cyber-physical systems can uniformly be described.

1.3 Contributions

This first part of this manuscript defines the algebraic formalization of assume/guarantee contracts that is formally proven to validate a meta-theory of contract. The logic is parameterised as a type class, in order to provide the suited generality of the model.

Our model of assume/guarantee contracts is entirely implemented in the calculus of construction of the proof assistant Coq. It comprises the algebraic operators of composition, conjunction, implementation and refinement. The model of assume/guarantee contracts is then validated with respect to the meta-theory of contracts.

The mechanism to extend the alphabet on which a contract is defined is implemented. This mechanism is also validated with respect to the meta-theory. Then, we implement the variable elimination process, which abstract a contract by removing one of the variable. Finally, we give an interface to facilitate the instantiation of the theory, given a logic as parameter.

The second part of the manuscript describes a practical use case of the definition of both the meta-theory and parametric A/G contracts models. It consists in the instantiation of the contracts' type class with differential dynamic logic as a parameter. This is proven correct by considering the model of hybrid programs and their abstraction, as two separate instances to model concrete components and their abstraction in the logic.

The first instance considers hybrid programs to model components. It is validated by the proof of a theorem relating the refinement relation defined for differential dynamic logic to the refinement relation for components in the assume/guarantee contract theory.

The second instance defines abstract programs: hybrid programs reduced to describe pre-conditions and post-conditions. It translates into hybrid programs, that is, the set of abstract programs is a subset of hybrid programs. We prove

this translation or concretization to be sound with respect to satisfiability.

Finally, we exercise the use of both instances on a typical case-study to illustrate the power of the theory to model components and their contractual abstractions. We define components and their specification in the assume/guarantee theory of contracts, and show the proof obligations needed to prove the consistency of the resulting model.

Chapter 2

State of the art

In this chapter, we review works on the use of contracts and formal methods for the verification of cyber-physical systems. Section 2.1 considers the process of designing and validating cyber-physical systems. Section 2.2 focuses on the different proof-assistant available, categorizing them into generic and domain-specific ones. Finally, Section 2.3 gives an overview of methods to improve modularity in mechanized proofs of cyber-physical systems.

2.1 Designing Cyber-physical Systems

We first highlight the numerous challenges of modeling and validating a cyber-physical system (CPS). We point out the importance of modelling such systems with high-fidelity and the intrinsic difficulties to do so. We explain when, in the "V design cycle", this modeling effort is done, and why it is crucial to do it precisely and exhaustively. Then, we give an overview of different logics that were introduce to model hybrid cybernetic and physical systems.

Ubiquity We call cyber-physical systems, every system which is constituted of a physical plant and a computerized controller [Raj+10]. If we zoom this definition out, we easily observe instances of such systems everywhere in our everyday life: planes, cars and trains are cyber-physical systems with a mechanical transport body, a pilot seat, a driving power-train and a sophisticated computerized controller to help piloting and routing safely.

Embedded Embedded systems are closely related to cyber-physical systems, a computer card plugged in a mechanical system constitutes such an embedded system. When modeling a cyber-physical system, we consider both the embedded card and its surrounding mechanical system. Whereas, to model an embedded system, the physical world is regarded as an abstract input to the system: its behavior is abstracted to the computational logic of the system.

Critical As cyber-physical systems encompass embedded systems, they hence are equally often critical. This implies that some of their behaviors must absolutely be verified. As an example, we expect a train to brake when there is an obstacle on the track, to ensure the safety of the passengers. The properties that must be validated to ensure the safety of the users of the system are called safety-critical properties. Theses properties must be defined by the engineer in charge of the specification of the system, and must be validated at each step of the design of the system. This raises the question of the process of designing of cyber-physical systems, and how theses properties are validated.

2.1.1 Validation Methodology for CPSs

The Cyclic V-Model The V-Model is a representation of the process of developing systems. It gives a good but superficial summary of the different step of a design workflow, as it yet doesn't give the full picture of the whole design

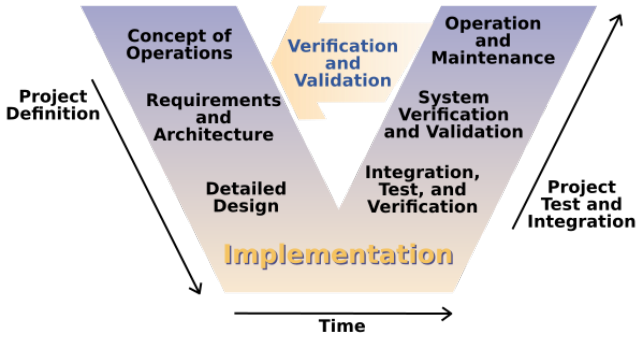


Figure 2.1: The V model [Osb+05].

process. It is an idealized view of design, where the system is fully specified, and modeled before being implemented. Once components are implemented, validated and integrated, the system is fully verified. Of course, every time a validation, integration or verification step doesn't succeed, the process has to backtrack to the origin of the failure to correct the problem and redo every step to the test that failed. This backtracking after implementation and construction of the system can be very costly. This is why modeling a project with precision before implementing it can save funding. The model must be precisely defined and its specifications should be as accurate and exhaustive as possible. By modeling the system according to the specification, one can ensure that the final system is safe, yet only up to the fidelity of that specification.

2.1.2 Industrial Tools for Modeling CPSs

We have seen that modeling is an important step in the design of CPS, now we present tools conventionally used to model cyber-physical systems. We focus on those that have a modular approach to parry the high-complexity of CPS.

Simulink Before introducing formal methods of modeling cyber-physical systems, we consider the subject of industrial tools to model CPS. Simulink, edited by MathWorks, is such a software. It allows modeling and simulating complex systems from multiple fields. Simulink has an easy-to-use graphical interface where components can be added and connected visually. Simulink provides built-in libraries of components simulating electrical or physical components, but can also be extended with custom components. With Stateflow, Simulink is extended with an automata creation tool, it is particularly useful to model the different modes of operation of a system.

Modelica Modelica is an open-source, object-oriented language for modeling, simulating and analyzing complex dynamic systems, particularly in the field of cyber-physical systems [MEO98]. Modelica provides a comprehensive and flexible

toolset for modeling different domains, including mechanical, electrical, thermal, fluid, and control systems. The language is based on ordinary differential equations, making it easy to describe complex physical phenomena and to perform advanced simulations and analyses. The use of Modelica in cyber-physical systems is beneficial as it facilitates the design and evaluation of these systems, and helps engineers to understand the behavior and performance of these systems under different conditions.

Scade Synchronous reactive languages are designed to model real time systems by describing them as reacting system to a flows of inputs. They are an efficient way to design systems that interact with the physical world. They are multiple synchronous reactive languages such as Lustre [Hal+91] and Signal [GLB87]. Scade is an environment to develop safety critical systems based on the Lustre synchronous language. Scade can produce C or Ada code, after the systems is modeled. Even thought Lustre is by nature a discrete language, based on sampling time according to a discrete clock, approximating the continuous evolution of physical systems, it was extended to Hybrid Lustre to model continuous evolution, by introducing continuous clocks that represent dense time [Yua+16].

Modeling Languages UML, is a general purpose modeling language, with an emphasis on a visual representation of the model. It is geared toward software engineering, but has a derivative that extends to CPS design: the System modeling language (SysML) [FMS14].

The Architecture Analysis & Design Language (AADL) is a graphical (and textual) architecture-centric description language [FGH06], specialized to the engineering of embedded systems in avionics. It contains standardized constructs to model the hardware and software architectures of embedded real time systems for the ARINC-653 normative runtime system and formalized semantics. It further contains extensions, called annexes, to model physical elements attached to that architecture.

Interestingly, the AADL has recently been equipped with a mechanized semantic model, Coqarina [Hug+22] and equipped with contracts [HP22] as part of a design language called Gumbo. Similarly, AUTOSAR [Fre10] can be perceived as a derivative of AADL specialized for the design of automotive embedded systems using the equally standard OSEK real-time operating system interface and has similarly been equipped with timing contracts [Yu+15].

2.2 Validation of CPS Models

Since cyber-physical systems are often critical, the use of effective and verified tools to prove their safety properties is paramount. We first give an overview of tools made for model checking of CPS, then we introduce correct-by-construction design methods. Multiple approaches could be adopted: we could use a generic proof-assistant and add libraries to model cyber-physical systems or we could use a proof-assistant specifically designed for the verification of cyber-physical

systems. In this section, we try to give an overview of the existing tools and environments to verify cyber-physical systems. For the interested reader, a recent survey exhaustively reviews a larger spectrum of formal methods used for the verification of cyber-physical systems [RST20].

2.2.1 Checking Automata Models

Starting from the possibly informal or ambiguous specification of a graphical modeler, a first step to verification is to obtain a formalized and possibly abstract model of the specification, traditionally, using automata, Petri nets or message sequences in order to make its state transitions explicit and exhaustively explore the space it defines using a model checker. A model checker takes such a model and a property to be verified by the variables it manipulates during its successive transitions: a temporal logic property.

Reaching a state where the property is not satisfied reveals of the existence of an undesired behavior. We call it a reachable property. On the contrary, if the property is valid in all state and transitions of the model, then it is called an invariant.

The path to a reachable property is revealed as a witness to help modify the model or, if the model is to abstract, refine the location of the counter-example in the model with a more precise specification.

Using this method, one can iteratively refine an initial model of the system to obtain one which satisfies all required invariants and whose undesired states are unreachable.

Hybrid Automata Hybrid Automata are an extension of automata in which a state can be associated with an ordinary differential equation (ODE) [Alu+93; Hen00]. The automata behave according to the usual semantics, switching states according to guards and conditions, the variables also evolve in a state according to the ODE if the state is associated with one.

The advantage of this logic is the availability of automata theory to manipulate the computational parts of the system, which is a well-known theory that adapts well to describe the functional modes of a system. In particular, the product of automata gives a natural definition to the composition of components. It is also visual, see Figure 2.2 to facilitate the design of system with this logic. Yet, it has the major disadvantage of incurring the combinatoric explosion of the product automaton's states.

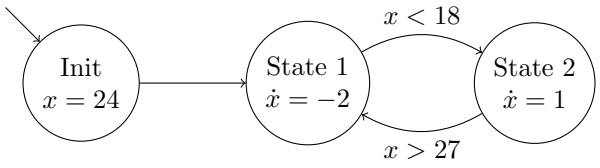


Figure 2.2: An example of a three state hybrid automata

TLA+ The temporal logic of actions, TLA, is a specification language for high-level modeling of concurrent systems. It is intended for the specification of a program that an engineer has in mind prior to its actual implementation. The specification consists of logical propositions subject to modalities expressing the its transitions in time: next, always, until.

TLA+ supports extensions to design cyber-physical systems [Lam93] and is particularly well suited to design concurrent systems, as most cyber-physical systems are. Programs are modeled as state machines that can be executed in order to check invariant properties of the system. It is intended for model checking, but can be used to prove safety properties with the TLA Proof System (TLAPS) [Cou+12]. Typically, invariant properties are verified with TLA+, which is suitable for proving the safety properties of CPS.

Signal Temporal Logic The Signal Temporal Logic (STL) is inspired by Linear Temporal Logic and defined as a language to specify real-time systems manipulating real values. The user creates rules in STL, that reflect the safety property of the system. Then a simulation is run to monitor if it respects its specified rules. [MN04].

2.2.2 Correct-by-construction design

By contrast to model checking, in correct-by-construction design, the model is created in a way that the safety property cannot be violated. In correct-by-construction design, the specification is formalized and the model is designed starting from the specification and validated at each step of the design process. The goal is to detect problem as soon as possible during the modeling process.

By formalizing the specification, one ensures that it is sufficient to validate the safety property, and by integrating it during the modeling process, one ensure the model satisfies it. As a result, correct-by-construction design provides higher guarantee than model checking, while being more time consuming to apply. Indeed, while model checking is focused on finding existing errors and removing them, correct-by-construction design aims at proving that no error may arise.

Unlike with model-checking , correct-by-construction design is usually not automatic, it is said mechanical. The correct-by-construction refinement of a formalized specification into an implementable model is performed using a so-called proof assistant.

A proof assistant is a tool to help prove mathematical and logical properties about programs and systems. Usually, properties will be expressed in a specific language or algebraic model. Then, with the help of *tactics*, properties will be refined until they becomes trivial and admissible. A proof-assistant usually has a trusted *core* to verify the correctness of the proof : its trusted computing base. We assume that the core of the proof assistant is correct, which implies that the proof created by the proofs assistant are correct. In this section, we will be interested in proof assistants which support the verification of cyber-physical systems.

Coq The Coq proof-assistant [Tea18] is based on Coquant's calculus of constructions. Its dependent type system makes its constructive logic very expressive and powerful to manipulate mathematical terms, algebras, infinite and continuous objects. Coq can additionally be used as a verified programming environment, as it allows the extraction of Ocaml functional programs upon successful proofs. A more detailed introduction to Coq is given in Section 3.1.

To verify a category of simple monitors as found in automated controllers for quadcopter drones [Ric+15; Mal+16], Veridrone developed a framework in Coq that supports the specification of a system using Linear-Temporal Logic. The framework is able to prove that the drone remains within a certain domain of evolution and aims using both the reals theory and proved C semantics to produce executable code for verified controllers. Similarly, RosCoq is a framework that uses Coq to verify cyber-physical systems, where the systems are robots. [AK15]. The user writes verified programs for the Robot Operating System and uses the Coq proof system to verify their specification with real analysis. Both frameworks use the extract code mechanism to have a certified executable after the system has been specified and implemented. As the Coq standard library is not suitable for the real analysis needed to deal with CPS, all of the above approaches use different libraries to compute real equations.

HCSP Hybrid Communicating Sequential Processes (HCSP) is an extension of Communicating Sequential Processes (CSP), a logic to describe the behaviour of multiple parallel communicating processes [CJR96]. In HCSP, components can perform discrete operations, like in CSP, but can also evolve continuously according to ordinary differential equation. Hybrid Hoare Logic was created to reasons about HCSP processes, and with the overall goal to verify their safety [Liu+10]. HHL is based on Hoare Logic, which can reason on CSP, and augmented with Duration Calculus to deal with continuous evolution [CHR91; ZH10].

Isabelle Isabelle (or Isabelle/HOL, its most widespread instance) is a generic and versatile proof-assistant [NPW02]. It allows mathematical properties to be written in its formal language, Isar, and proven using of its IDE such as Isabelle/jEdit. It features some very powerful automation tactics compared to Coq, and notably *sledgehammer*, a tool to heuristically find a path toward the complex proof of a sophisticated properties. Hybrid Hoare Logic (HHL) and HCSP have been encoded in Isabelle with both a deep and a shallow embedding [WZZ15; Zou+14].

Mars Modelling Analyzing and veRifying hybrid Systems (Mars) is a tool chain to transform Simulink models into HCSP models and specify them with HHL [Che+17]. Thanks to the embedding of HHL and HCSP in Isabelle, the model can then be formally verified using Isabelle powerful tactics and automation. Simulink is a widely used environment to model systems of all sorts, discrete or continuous. It is very modular and visual, components are like boxes

that connects one another. Yet, the language of Simulink is not formally defined, which makes it difficult to verify the safety of models. This is why, transforming models from Simulink to HCSP is a step toward proving the safety of Simulink models. Moreover, the inherent modularity of Simulink models can be used to create a modular proof of the model. This is a important point because cyber-physical systems are often big and complex which makes it difficult to express their safety as one property.

UTP The Unifying Theory of Programming (UTP) is a logic meant to unify operational semantics, denotational semantics and algebraic semantics into a unifying framework [HJ98; WC04]. It encodes programs as the predicates, precisely, the strongest predicates describing the behavior of a program. Healthiness properties can be added to encode a particular programming paradigm. Foster et al. mechanized UTP in Isabelle/HOL as Isabelle/UTP [FZW15], and applied it to describing a discrete semantics and contracts theory for reactive systems design [FW17]. Later, Xu et al. [Xu+23] extended the UTP to define a denotational semantic model of hybrid systems (HUTP) capturing earlier models of HCSP and of Simulink/Stateflow in the Mars environment.

2.2.3 Dedicated Proof-Assistant for Modeling CPS

We observe that using generic proof assistants to design and verify cyber-physical systems requires to address the issue of extending them with constructive models of concurrency and functions over reals, which the Mars platform does for instance by the introduction of a hybrid extension to Hoare logic (HHL) in the Isabelle proof assistant. Instead, others have developed models and implementations of specific proof assistant for modeling cyber-physical systems.

Differential Dynamic Logic Differential dynamic logic ($\mathbf{d}\mathcal{L}$), is a logic and a proof calculus designed to model and verify cyber-physical systems. In $\mathbf{d}\mathcal{L}$, two expressions are defined recursively : hybrid programs and hybrid formulas [Pla08]. A hybrid program represents an abstraction of the behaviour of a system. Hybrid programs are equipped with the usual programming operator (e.g. sequence) to describe discrete computations. They are also equipped with ordinal differential equations to describe the continuous evolution of physical components. Hybrid formulas are propositional formulas that can express conditions on hybrid programs. Hybrid formulas can express two kinds of modality, invariance: the program must satisfy a property for all runs, or reachability: the program accepts a run that satisfy the property. Instead of resolving the differential equations, differential dynamic logic proposes to compute a differential invariant which suffices to prove the desired safety property. A differential invariant is the invariant of a differential equation: a domain that the variables it defines cannot cross at any time. By proving that the invariant is satisfied during the evolution of the differential equation, and by proving that it implies the safety property, we prove that the safety property is satisfied during

the evolution of the differential equation. A more exhaustive introduction to differential dynamic logic is given in Section 4.1.

Keymaera X To prove $\mathbf{d}\mathcal{L}$ properties, Platzer et al. developed a proof-assistant, Keymaera X [PQ08; Ful+15]. Keymaera X implements a language of tactics, like Coq or Isabelle, to direct the derivation of a proof tree of discrete computational properties and formulas according to the proof system of its logic, differential dynamic logic. To resolves numerical questions the tool discharges them into SMT solvers such as Z3 [dMB08].

Rodin Event-B The B-method [Abr10] is a framework for refinement based design of complex systems. The generic approach of Event-B is to first specify model abstraction of a complex system by enumerating its requirements. The design workflow of Event-B then consists in the iterative, correctness-preserving, refinement of these requirements toward consistent and concrete implementations in C or Ada. An important line of work in the Event-B community is its extension to the model of cyber-physical systems [Dup+18] and [CM20] as well as [MAL22], which also regards it from the perspective of dynamic differential logic. Examples uses of Method B in industry-scale system design are for instance: Meteor, the design of the automated control system for line 14 in Paris' subway network [Beh+99].

2.3 The Issue of Compositionality

Validation of cyber-physical systems with proof assistants is difficult due to their complexity and heterogeneity. Methods to reduce this complexity by using abstraction, modularity and compositionality is an active field of research [Gra+18; SDP12]. To this end, the notion of contracts as been introduced in order to provide modularity to the verification of cyber-physical systems. First, we introduce notions of contracts specifically designed for differential dynamic logic and hybrid hoare logic, and then we outline a generic framework of contract theory with the aim of unifying these different theories.

2.3.1 Logical Contracts for Hybrid Systems

Lunel's Component Model Lunel's work explores the idea to contain the definition of components in differential dynamic logic to follow a certain pattern in order to support interleaving, hence parallelism and commutative composition, by design [Lun19; LBT17; Lun+19]. This pattern is also designed to precisely model computer-controlled systems. A computer-controlled systems is constituted of an analog plant and a digital computerized controller 2.3. Theses two parts are composed to form the system, or a subsystem. Lunel shows how the contracts of each such subsystems compose consistently with the composition of their corresponding components.

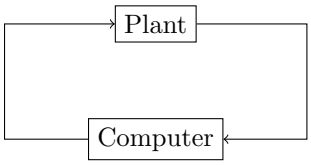


Figure 2.3: The schematic of a computer controlled system

Müller's Contracts In an effort to bring modularity to differential dynamic logic, an algebra of contracts has been defined specifically for it [Mül+18; Mül+17; Mül+16]. In this work, the components consist of three parts: computation, plant, and communication ports. The symbolic variables called ports are used to communicate between the components. Components can be specified with interfaces that describe how they interact with the environment, their assumption, and their guarantee. In an effort to specify different aspects of the system, different types of contracts were created. For example, some contracts specify the extent by which a value can change. Others specify the time period over which a change can occur.

Wang's Contracts Wang et al. extended the hybrid Hoare logic calculus to handle compositionality [WZG12]. Indeed, because processes can communicate and execute in parallel, specifying the composition of processes is very challenging. By specifying processes with an assume/guarantee pattern, HHL constructs a calculus suitable for handling modular specifications of complex system. In this development, the components are free to be of any form. This is in contrast to the work of Lunel and Muller, where components were need to be structured in a particular form.

Foster's Contracts Foster et al. have proposed a mechanized theory of contracts in Isabelle and the Unified Theory of Programming (UTP) using pre-, peri- and post-conditions to model discretely timed reactive systems. [Fos+20]. This framework would be a good candidate to extend the theory of hybrid systems in the UTP [Xu+23] with a theory of contracts.

2.3.2 Generic Theories of Contracts

Foundations of Contracts for System Design Design by contracts was first proposed by Meyer for software programming [Mey92]. Specification by contracts traditionally consists of pre- and post-conditions, leaving the continuous timed variables extraneous of the specification. This is not practical when designing cyber-physical systems, where time is intrinsically linked to the continuous behavior of the system. Contracts for cyber-physical systems replace pre-conditions by assumptions and post-condition by guarantees, the main difference being that the assumption and guarantee can express continuous properties: invariants of or hypothesis on the environment's behavior; invariants

or guarantees on the component's behavior. Two paper lay the foundations of the theory of specifications and contracts adapted to system design [AL93; AL95]. They address the problem of composing specifications from a generic and semantic point of view that on the logic used to express properties. They define the basic operations and relations of contracts such as compositions, conjunction, refinement and saturation.

Contract from Specifications Bauer et al. build a framework to construct a theory of contract from a specification theory [Bau+12]. A specification theory, is constituted of a class of objects called specifications, equipped with a parallel composition operator and a refinement relation. In this framework, contracts are pairs of specifications, one specification for the assumption and one specification for the guarantee.

The Meta-Theory In an effort to unify contract theories, Benveniste defined a meta-theory of contracts [BNH14; Ben+15a; Ben+08]. It can be instantiated as various classes of contract theories and defines an algebra component algebra. A more detailed introduction is available in Section 3.2.

More recently, contracts have been extended to hypercontracts [Inc+22]. Hypercontracts are similar to assume/guarantee contract, but the assume/guarantee properties are replaced by hyperproperties which allows to specify more systems. For example, information flow cannot be specified with properties on traces, but could be specified with hyperproperties.

Conclusion

While slightly different from one another, all these definitions of contracts retain the same core ideas from the seminal works of Meyer, Abadi and Lamport. Namely, a contract abstracts the specification of a component in a logic. These definitions also support the same usual operators on contracts such as composition or refinement.

A meta-theory of contracts has been defined, aiming at unifying all theories of contracts [Ben+15b]. The theory of assume/guarantee contract instantiates the meta-theory and is generic enough to bridge the gap between related definitions of contracts [AL93; Ben+08]. It was used, for instance, to define contracts in heterogeneous logics and relate them together [Nuz15].

In this chapter, we gave an overview of the different works related to the proof assisted verification of cyber-physical systems. First, we showed the different methods to validate cyber-physical systems. Then, the different proof assistant were described and their uses to verify cyber-physical systems were given. Finally, we introduced the different methods of adding modularity to the methods. This state of the art leads us to contribute a formalized contract theory and to parameterize it by a logic, to serve as a theoretical foundation and to practical applications by instantiating it to a particular logic: differential dynamic logic.

The meta-theory of contracts and the theory assume/guarantee of contracts have, to the best of our knowledge, no formalized proof of correctness. Nor does, by extension, its possible instantiation to specific logics for hybrid and cyber-physical systems. In the following chapters, we propose a formalization of both of them in the proof assistant Coq with the goal to instantiate it with a hybrid logic that could be defined using that theorem prover.

Chapter 3

A Verified Contract Theory

This chapter introduces our mechanized formalization of the theory of assumption/guarantee contracts. First, we give an introduction to Coq, the proof-assistant used throughout the thesis. Then, in Section 3.2 an overview and a related to the meta-theory of contract is given. Section 3.3 presents the mechanization of the theory of assumption/guarantee contracts. Section 3.4 proves that the theory of assumption/guarantee contracts is a specialization of the meta-theory of contracts. Sections 3.5, 3.6 and 3.7 extend this mechanization: Section 3.5 first presents the logical tools to deal with contracts and components of different alphabets. Section 3.6 models the elimination of unnecessary variables from a specification. Section 3.7, introduces an interface for the mechanization. Section 3.8 concludes this chapter. The code presented in this section is located in the file `VCT/Contracts.v` of the development available at this address: `https://gitlab.inria.fr/skastenb/differential-contracts`.

3.1 Introduction to Coq

In this section, we introduce Coq, the proof-assistant we use to mechanize the formalization of contracts. We give a very short introduction to the basic syntax of Coq, and some key principles we use in the development. For a more in-depth exploration of Coq, the interested reader is directed towards the multiples introduction made for Coq [BC13].

3.1.1 Basic Syntax

Function Functions are first-class citizen, they are defined with the keyword `Definition`. In the example below, `double`, is the name of the function, (`n : nat`) is its parameter, `: nat` is the type of its return value, and `n + n` is the body of the function. A function can have any number of parameters, including zero, hence, non-inductive constant object or type can be defined using `Definition`. Below we define `set`, an abstract type to encode sets in Coq. If the type of the parameter can be deduce from the type checker it is not mandatory to write them, we chose to write them or not for a better human-readability and to help the type checker.

```
Definition double (n : nat) : nat := n + n.

Definition set (Γ : Type) : Type := Γ → Prop.
```

Parameters of can be of any `Type`, `Set` or `Prop`, and can be dependent. Concretely this means a type or a function can be parameterized by another value of any type, including `Prop` or `Type`. In this example we code the function `In`, which takes a type Γ, a object `x` of type Γ, and a set of object of this type `s`, and return if this object is in the set. Here, the types of `s` and `x` are dependent of Γ.

```
Definition In {Γ : Type} (x: Γ ) (s: set Γ ) : Prop := s x.
```

Theorem Properties can be stated with `Lemma`, `Theorem` or `Corollary`, they will be of type `Prop`. They are proved by modifying the hypotheses or conclusions until the proof becomes trivial. Tactics are the instructions which modify the hypothesis or conclusion. In reality, Coq construct a proof term, an object that is the proof of the statement, using the Curry-Howard equivalence. In this thesis, we do not show the process of proving and the tactics used, hence we don't introduce tactics and the process of proving the properties in Coq. Indeed, most of our challenges lie on the definition of relevant lemmas, and finding the correct proof scheme.

```
Lemma double_greater (n : nat) :
  double n >= n.
```

Induction New inductive types can be defined in Coq using the keyword `Inductive`. For every type defined this way, Coq will generate useful lemmas, such as the induction principles. In the example, we defined `Tree` a simple binary tree structure. `Leaf` is a constructor from a natural number, `Leaf 4` is a tree with one leaf with number 4. `Branch` construct a tree from two other trees. Coq produces the lemma `Tree_ind` to prove properties about tree inductively, it is best used with the `induction` tactic.

```
Inductive Tree : Type :=
  | Leaf : nat → Tree
  | Branch : Tree → Tree → Tree.
```

Record We can define record types with the `Record` keyword, as in programming languages, they aggregate data in a single entity with multiple fields. The user must name the record type `Student`, define name of the constructor `MkStudent`, and label the records and their types `id : nat`. Similarly to other Coq definitions, the fields can be parameterized by values of any Type, particularly, a field can be the parameter of another field.

```
Record Student : Type := MkStudent
  { id : nat ;
   age : nat}.
```

Type Class Similarly, we can define type classes, with `Class`. Under the hood, they are parameterized records. Type classes are used to define functions parameterized by a group of properties. The interested reader can find the full definition in [SO08]. In the example below, we define the type class `Equivalence` with the `Class` keyword. `Equivalence` is the class of relation `eq` that respect the three properties of transitivity (`eq_trans`), symmetry (`eq_sym`) and reflexivity (`eq_refl`). Then we define an instance of `Equivalence` for natural numbers `nat_eq`. We prove that the properties hold for this relation (`omega` is a tactic for automatically proving arithmetic properties), and instantiate the type class with the keyword `Instance`. Thanks to type class mechanism, every function defined for `Equivalence` can be used with `nat_Equivalence`.

```
Class Equivalence (A : Type) := {
  eq :  A → A → Prop ;
  eq_trans : ∀ (a b c : A), eq a b → eq b c → eq a c ;
  eq_sym : forall a b : A, eq a b → eq b a ;
  eq_refl : forall a : A, eq a a ;
  }.

Definition nat_eq (a b : nat) := a <= b ∧ a >= b.

Lemma nat_eq_trans (a b c : nat) : nat_eq a b → nat_eq b c → nat_eq a c.
Proof.
  unfold nat_eq ; omega.
Qed.

Lemma nat_eq_sym : forall a b : nat, nat_eq a b → nat_eq b a.
Proof.
  intros a b ; unfold nat_eq ; omega.
Qed.

Lemma nat_eq_refl : forall a : nat, nat_eq a a.
Proof.
  intros a ; unfold nat_eq ; omega.
Qed.

Instance nat_Equivalence : Equivalence nat := {
  eq := nat_eq ;
  eq_trans := nat_eq_trans ;
  eq_sym := nat_eq_sym ;
  eq_refl := nat_eq_refl ;
  }.
```

3.1.2 Set Library

In this section, we introduce the library we use to encode set in our development. It will lead us to introduce the principle of extensionality and classical logic in our library.

Set Definitions We define a `set` type, parameterized by a type Γ. In that way, the type `set Nat`, is the type of natural set. This way, we define a `set` theory, independent from the type of elements. We define three basic operations on sets, `In` the belonging relation, `SubsetEq` the inclusion of sets and `Eq` the equality of sets. We also define the union `union` and intersection `inter` relations for sets. In this encoding of sets, an element `x` is in the set `s` if and only if `s x` ↔ `True`.

```
Definition set (Γ : Type) : Type := Γ → Prop.
```

```
Definition In {Γ : Type} (x: Γ ) (s: set Γ ) : Prop := s x.
Notation "x ∈ s" := (@In _ x s) (at level 70, no associativity).
Notation "x ∉ s" := (¬ @In _ x s) (at level 70, no associativity).

Definition SubsetEq {Γ : Type} (s1 s2: set Γ ) : Prop :=
    forall x: Γ , x ∈ s1 → x ∈ s2.
Notation "u ⊆ v" := (@SubsetEq _ u v) (at level 70, no associativity).

Definition Eq {Γ : Type} (s1 s2: set Γ ) : Prop :=
    forall x: Γ , x ∈ s1 ↔ x ∈ s2.
Notation "u == v" := (@Eq _ u v) (at level 70, no associativity).

Definition union {Γ : Type} (s1 s2: set Γ ) : set Γ :=
    fun x: Γ ⇒ x ∈ s1 ∨ x ∈ s2.
Notation "u ∪ v" := (@union _ u v) (at level 61, left associativity).

Definition inter {Γ : Type} (s1 s2: set Γ ) : set Γ :=
    fun x: Γ ⇒ x ∈ s1 ∧ x ∈ s2.
Notation "u ∩ v" := (@inter _ u v) (at level 51, left associativity).
```

3.1.3 The Issue of Extensionality

Since we defined an equality of set `Eq` that is not the default equality of Coq, we want to prove that `Eq` implies the default equality, namely the theorem `Eq_extensionality`. Two difficulties will come from this, one the default Coq equality is intensional, meaning objects are equal if they are defined equaly. We will elaborate on that point and functional extensionality in the following. Two, we will need to prove that `s1 x ↔ s2 x → s1 x = s2 x`, for `s1` and `s2` two sets and `x` an element of those sets. This is false by default in Coq, because proof of the same properties are not necessarly equal. We need to assume propositional extensionality, we talk about this later.

```
Theorem Eq_extensionality: forall {Γ : Type} (s1 s2 : set Γ ),
  s1 == s2 → s1 = s2.
```

Functional extensionality By default, Coq uses the principle of intensionality, meaning that two terms are equal if they are equal syntactically. Extensionality, by contrast, means that two object are the same if they have the same external properties. In usual mathematics and notably in set theory, set is an abstract concept which relies on the elements of each set. As such, two sets are equal if they contain the same elements exactly. Since we uses set theory in our development, the axiom of extensionality, which states that two sets are equal if they have the same elements, has to be added to our context. For this reason, we include the axiom of functional extensionality. The standard library of Coq has a module for this: `FunctionalExtensionality`[1].

[1]The documentation of the module can be found at: `https://coq.inria.fr/distrib/current/stdlib/Coq.Logic.FunctionalExtensionality.html`

```
Axiom functional_extensionality_dep : ∀ {A} {B : A → Type},
  ∀ (f g : ∀ x : A, B x),
  (∀ x, f x = g x) → f = g.
```

Propositional Extensionality We also need the propositional extensionality[2], it states that two properties that are equivalent in the mathematical sense are replaceable. Indeed, in constructive logic, this is false, two proofs of the the same statement are not equal if their proof terms are not syntactically equal. With propositional extensionality, we consider that two equivalent statements are equal.

```
Axiom propositional_extensionality :
  forall (P Q : Prop), (P ↔ Q) → P = Q.
```

Set Extensionality With both axioms, we can prove the following extensionality axiom `Eq_extensionality`, it states that two sets with the same elements are equal.

```
Theorem Eq_extensionality: forall {Γ : Type} (s1 s2 : set Γ ),
    s1 == s2 → s1 = s2.
Proof.
    unfold set, Eq, In.
    intros.
    apply functional_extensionality.
    intro x.
    apply propositional_extensionality.
    apply H.
Qed.
```

3.1.4 De Morgan's Laws

For our set library we would like to prove both De Morgan's laws (Equation 3.1 and 3.2). They are set manipulation results that we are going to need for our development. Yet, they are not directly provable in Coq.

$$\overline{A \cup B} = \overline{A} \cap \overline{B} \tag{3.1}$$

$$\overline{A \cap B} = \overline{A} \cup \overline{B} \tag{3.2}$$

Classical Logic Indeed, we usually reason with classical logic. In classical logic, we have an axiom call excluded middle, it states that every proposition is either true or its negation holds. In Coq we use constructive logic, this axiom is not admitted, thus we must always give a proof term to construct an object. In particular to prove something exists, one must construct the object, proving

[2] https://coq.inria.fr/distrib/current/stdlib/Coq.Logic.PropExtensionality.html

that the object cannot exist is insufficient. This makes it possible for Coq to extract executable code from the definitions.

Yet, we need classical reasoning to prove De Morgan's laws. Indeed, equation 3.2 is only true if we accept $\forall x,\ x \in A \vee x \in \overline{A}$, a consequence of the excluded middle. As such, we assume the `classic` axiom to prove De Morgan's laws[3]. In our thesis we use constructive logic, with this set library as exception. We aim to remove the uses of the excluded middle axiom, but it is easier to accept it for now.

```
Axiom classic : forall P : Prop, P ∨ ¬ P.

Lemma morgan {Γ : Type} : forall s1 s2 : set Γ , ¬s1 ∩ ¬ s2 = ¬ (s1 ∪ s2).
Lemma morgan2 {Γ : Type}: forall s1 s2 : set Γ , ¬(s1 ∩ s2) = ¬ s1 ∪ ¬ s2.
```

3.2 Overview of the Theories of Contracts

This section gives an overview of the contracts theories used in the mechanization of Section 3.3. We start with an intuitive introduction to component-based design and assumption/guarantee contracts, which we progressively formalize with mathematical definitions. We continue with an introduction to the meta-theory of contracts. Finally, we show that the theory of assumption/guarantee contracts is an instance or specialisation of the meta-theory of contracts.

3.2.1 Introduction to Component-Based Design

Component-based design is a paradigm of system design, where the system is modeled as a combination of components. In this paradigm, the design process consists of combining the right components the right way, to ensure the functional correctness of the system and its safety.

Component Components are functionally independent parts of a system, they can have inputs and outputs, they compute the outputs from the inputs. In component-based design, we consider some atomic components, they are the simplest component in the system. Two different meanings can be given to the notion of component. In the first case, the components are blueprints of the actual parts that will be replicated to build the system. In the second case, components are functionally unique, they are the actual building block of the systems. In our development, we will use the second definition. Components are interconnected with each other, meaning that the output of one components can be connected to the input of another component. For example, a water tank and a valve controlling the inflow of the tank are combined into a system. This system has two components, the valve and the tank.

[3] `https://coq.inria.fr/distrib/current/stdlib/Coq.Logic.Classical_Prop.html`

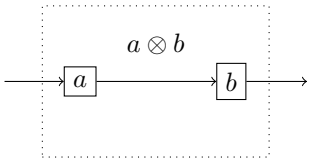


Figure 3.1: Composition of a and b

Composition of Components Composition allows to regard two components as one. We define a composition operator, that constructs a component from two components. The component constructed should have the correct inputs and outputs. For example, the input of a, is also an input of $a \otimes b$ if it's not an output of b, see Figure 3.1.

Contract We also want to create specifications for those components. We choose to specify components with contracts, because they are written in a formal language as opposed to documentation which are informal descriptions in natural language. Contracts are equipped with an algebra which allow us to compose them with each other. The goal of a contract is to abstract the behaviour of the components, and to focus on the connections it has with the other components. We hence won't describe how an individual contract is constructed, since it is not defined by construction but by intent: what we can do or prove with it. For example, the contract of a valve could express in a formal language that the inflow is always positive ($v > 0$).

Composition of Contracts To construct the contract that specifies the composition of components we have an operator that composes contracts. Composed components and contracts should match: composed contracts should define assumptions and guarantees on the corresponding composition of components. For example, if a contract is the result of the composition of the contracts of the valve and the water tank, it should specify the system.

Conjunction Sometime a specification can target different and independent features of a component. For example, one specification may pertain to the correctness of an output, and the other relate to the speed at which this output can be computed. These two specifications are independent but relate to the same component. We can conjoin these contracts, to have a single contract encompassing the two aspects of its specification: correctness and timing.

Refinement The specification of a component can adopt different levels of abstraction. This induces an order relation, the refinement relation, to express how a contract abstracts or refine another. If a contract specifies a subset of the properties (e.g., $x \in \mathbb{N}$) another contract specifies (e.g., $x > 10$), we say that it refines it, and conversely, abstracts it. Indeed, it is sometimes easier to

manipulate an abstract contract when unnecessary details are irrelevant (e.g., to a verification goal).

3.2.2 A/G Contracts for Component Specification

We now introduce the assumption/guarantee contracts theory and give the mathematical definitions of the objects involved.

Variables and Alphabets We start by the definition of a variable, they are the inputs and outputs of the components. $\mathcal{V}$ is the set of all variables identifier. Each part of the system, is defined using only a subset of $\mathcal{V}$, we call this set of variables its alphabet. For simplification purpose, we first suppose that all components and contracts are defined on the same alphabet $d \subseteq \mathcal{V}$. Below, we define operations to change the alphabet an object is defined on, by eliminating or by adding a variable to its alphabet. The alphabet of the valve is $d = \{\mathtt{h}, \mathtt{v}\}$.

Behavior Variables are containers for values, for simplification purposes we consider that all variables hold value in the same domain. In the examples we use $\mathbb{R}$ as the domain of value. A behavior is a valuation of all the variables in the alphabet, that is to say, it is a function from d to $\mathbb{R}$. The behavior space $\mathcal{B}_d = d \to \mathbb{R}$ is the set of all behavior on alphabet d. For example the behavior s is defined on d by $s(\mathtt{h}) = 1$ and $s(\mathtt{v}) = 3.5$. For the sake of readability we note (h, v) the behavior that is defined by $(h, v)(\mathtt{h}) = h$ and $(h, v)(\mathtt{v})(v)$.

Assertion An assertion is a logical statement on the variables of an alphabet d. By using the duality of sets, it can be seen either as a collection of elements or as the property that every element satisfies. Here, we consider an assertion to be the set of all behaviors that satisfies the property it denotes. We define an assertion a defined on d as a subset of the behavior space, noted $a \subseteq \mathcal{B}_d$. Here is an example of assertion: $\{(h, v) \mid v > 0 \wedge h < 20\}$

Definition 1 (Assertion satisfaction)**.** *For an assertion a and a behavior s, we say s satisfies a iff $s \in a$.*

Component We model a component by the assertion it guarantees on the system's behaviors. There is no clear differentiation between the inputs and outputs of a component (as long as causality is not the matter); they are both variables in d the alphabet. A component is not only viewed as a logical property relating inputs and outputs, but as every behavior satisfying that property. This defines a component by the set of behaviors it can follow. So the formal definition of a component is a subset of $\mathcal{B}_d$, though it is more convenient to think of it as the logical statement relating inputs and outputs. For example a model of the valve could be modeled as the component that exists only positive inflow: $\{(h, v) \mid v > 0\}$

Composition of Components Composing components construct a new component. We can compose any two components together. The composition of two components of same alphabet is simply the intersection of their assertions. If such a composition yields the empty set (of behaviors), the component is defined but cannot be used since it is a unimplementable component. For example, the composition of the valve keeping the flow positive, and the water tank keeping the water-level low is $\{(h, v) \mid v > 0\} \cap \{(h, v) \mid h < 20\} = \{(h, v) \mid v > 0 \wedge h < 20\}$. Later on, we discuss the composition of components with different alphabets.

Environment As in the assumption/guarantee contract theory, when composing $\alpha \cap \beta$, we say that β is an environment for α (and reciprocally). When considering a component we call the rest of the system the environment, yet formally, the environment is just another name for a component. For example, when considering the motor of a car, the other components interacting with the motor form a sub-system: the environment of the motor. Conversely, when considering the gearbox, the other components of the car, including the motor, form its environment.

Assumption/Guarantee Contract A contract is a possibly abstract specification of a component. In assumption/guarantee contract theory, contracts are pairs assertions: an assumption and a guarantee. Informally, the assumption is the property required from the environment for the component to function and the guarantee is the property ensured by the component. The goal is to define rules which restrain both the implementation of the component and the environment it needs to embed. Here, the assumption is the restriction on the environment whereas the guarantee is the restriction on the component. An example of contract for the tank could be : $(\{(h, v) \mid v < 5\}, \{(h, v) \mid h < 20\})$, meaning that, if the flow is lower than 5, the water-level is kept below 20.

Definition 2 (Assumption/guarantee contract)**.** *An assumption/guarantee contract is the combination of two assertions, one for the assumption (A) one for the guarantee (G). For a contract $c = (A, G)$ we define projections to get the assumption and the guarantee*

$$A(c) = A \ ; \ G(c) = G$$

.

Implementation Now, we define what it means for a behavior to *satisfy* a contract. We lift this definition to components that *implement* a contract. Informally, a component implements a contract if it it specified by the contract. So, for example, the tank $\{(h, v) \mid v < 5 \rightarrow h = 10\}$ is an implementation of the contract: $(\{(h, v) \mid v < 5\}, \{(h, v) \mid h < 20\})$ (or a more refined contract).

Definition 3 (Satisfies)**.** *A behavior s satisfies a contract if either it's a behavior excluded from the contract's assumption, or the contract's guarantee holds for the behavior.*

$$s \vdash c \equiv s \in \overline{A(c)} \cup G(c)$$

Definition 4 (Implements)**.** *A component σ implements a contract if every behavior in σ satisfies the contract.*

$$\sigma \vdash c \equiv \forall s \in \sigma,\ s \vdash c$$

An environment e provides a contract if every behavior of e is included in the assumption of the contract. So the component $\{(h, v) \mid v = 2\}$ provides the contract $(\{(h, v) \mid v < 5\}, \{(h, v) \mid h < 20\})$.

Definition 5 (Provides)**.**

$$e \vdash^P c \equiv \forall s \in e,\ s \in A(c)$$

Saturation This leads to a particular point in assumption/guarantee contracts. With the above definition, we notice that multiple contracts can be implemented by the same components and provided by the same environments. So we have a class of contracts which are all equivalent, as they specify the same set of components and environment. In order to normalize contracts, we define *saturation* an idempotent operation which doesn't change the set of components satisfying the contract, nor the set of environment providing the contract. The saturation operation is idempotent, namely $\text{saturate}(\text{saturate}(c)) = \text{saturate}(c)$. Since it doesn't change the components which implement the contract, nor the environment that provides it, we always use the saturated version of a contract. In the remainder, we assume contracts to be saturated, the definitions of conjunction, refinement and composition would be different if the contracts were not saturated:

For example, the contract c_{sat} is the saturated version of the unsaturated c.

$$\begin{aligned} c = &\quad (\{(h, v) \mid v < 5\}, && \{(h, v) \mid h < 20\}) \\ c_{sat} = &\quad (\{(h, v) \mid v < 5\}, && \{(h, v) \mid v < 5 \rightarrow h < 20\}) \end{aligned}$$

Definition 6 (Saturation)**.**

$$\text{saturate}(c) \equiv (A(c),\ \overline{A(c)} \cup G(c))$$

Lemma 1 (Saturation soundness)**.** *Saturation doesn't change the set of implementing component, and providing environment.*

$$\sigma \vdash c \iff \sigma \vdash \text{saturate}(c)$$

$$e \vdash^P c \iff e \vdash^P \text{saturate}(e)$$

Refinement Next, we want to define the refinement relation between two contracts. In this definition and the two following ones (conjunction and composition), we consider contracts to be defined on the same alphabet d and to be saturated. The most refined contract is the one which has the strongest guarantee and the weakest assumption. The guarantee is stronger because refinement

allows a more precise guarantee. The assumption is weaker because the refined contract can accept a more abstract environments. For instance, the contract $a \preceq$-refines b.

$$\begin{aligned} a &= (\{(h,v) \mid v \in \mathbb{R}\}, && \{(h,v) \mid v \in \mathbb{R} \rightarrow 0 < h < 10\}) \\ b &= (\{(h,v) \mid v < 5\}, && \{(h,v) \mid v < 5 \rightarrow h < 20\}) \end{aligned}$$

Definition 7 (A/G refinement)**.**

$$c1 \preceq c2 \;\equiv\; A(c1) \supseteq A(c2) \;\wedge\; G(c1) \subseteq G(c2)$$

Conjunction The conjunction of contracts, which constructs a contract that combines two contracts specifying different aspects of a component, is defined by the formula below.

Definition 8 (A/G conjunction)**.**

$$c1 \sqcap c2 \;\equiv\; (A(c1) \cup A(c2),\; G(c1) \cap G(c2))$$

For instance, the conjunction of
$(\{(x,y) \mid x \neq 0\}, \{(x,y) \mid x \neq 0 \rightarrow y = x^{-1}\})$ and
$(\{(s_x,s_y) \mid s_x < 2\}, \{(s_x,s_y) \mid s_x < 2 \rightarrow s_y < 5\})$ is
$\left(\{(x,y,s_x,s_y) \mid x \neq 0 \vee s_x < 2\}, \left\{(x,y,s_x,s_y) \;\middle|\; \begin{array}{ll} & (x \neq 0 \rightarrow y = x^{-1}) \\ \wedge & (s_x < 2 \rightarrow s_y < 5) \end{array} \right\}\right)$.

Composition of Contracts The guarantee of composed contracts is the conjunction of the contracts' guarantees, because it is provided by both of the components it denotes. The assumption of composed contracts is the conjunction of the contracts' assumptions, relaxed from the guarantees. The idea, is that the component implementing the contract resulting from the composition can accept any behavior that is outside the scope of both guarantees. We give an example of composition of contracts below.

$$\begin{array}{ll} & (\{(h,v) \mid v < 5\}, \{(h,v) \mid v < 5 \rightarrow h < 20\}) \\ \otimes & (\{(h,v) \mid v > 0\}, \{(h,v) \mid v > 0 \rightarrow v < 5\}) \\ = & (\{(h,v) \mid v > 0\}, \{(h,v) \mid v > 0 \rightarrow (h < 20 \wedge v < 5)\}) \end{array}$$

Definition 9 (A/G composition)**.**

$$c1 \otimes c2 \equiv (A', G')$$

With

$$A' \equiv A(c1) \cap A(c2) \cup \overline{G(c1) \cap G(c2)}$$

$$G' \equiv G(c1) \cap G(c2)$$

Quotient of Contracts Until recently, there was no definition of quotient in A/G contracts, for example it was not defined in the theory defined by Benveniste [Ben+08]. The quotient was defined for assumption/guarantee contracts, but it is not defined in our formalization [Ínc+18]. The quotient is the inverse of composition. It is characterized by the fact that given contracts c and c_1, for every contract c' : $c' \preceq c \setminus c_1 \iff c' \otimes c_1 \preceq c$.

Definition 10 (A/G Quotient)**.**

$$c \setminus c_1 \equiv (A(c) \cap G(c_1),\ A(c_1) \cap G(c) \cup \neg(A(c) \cap G(c_1)))$$

Compatibility and Consistency We add two useful definitions to characterize the non-emptiness of the assertions in contracts. Indeed, contracts with empty guarantee or empty assumption should be treated differently as they are not implementable by a component or providable by an environment.

Definition 11 (Compatibility)**.** *A contract $c = (A, G)$ is compatible if A is non-empty.*

Definition 12 (Consistency)**.** *A contract $c = (A, G)$ is consistent if $A \to G$ is non-empty.*

Alphabet Equalization We are now interested in operations on contracts from different alphabets. Since we have only defined the operators on contracts with same alphabet, we want to modify the contracts so they have the same alphabets. The meta-theory names this process "Alphabet Equalization". In the following we define the tools to equalize alphabets of contracts. First, we define two projections operators on assertion. The first is the projection of the assertion on a smaller alphabet. The second one is the inverse projection, or extension, of an assertion on a bigger alphabet. Thus, we can update contracts by extending their alphabet to any larger alphabet. To equalize the alphabets $d1$ and $d2$ of two contracts $c1$ and $c2$, we extend both contracts' alphabets to the alphabet $d = d1 \cup d2$.

Definition 13 (Projection)**.** *For a an assertion defined on d and $d1 \subseteq d$.* $\mathbf{pr}_{d1}(a)$ *is the set of all behaviors of a, restricted to $d1$.*

As an example let us consider the assertion $a = \{(h, v) \mid h > 0 \wedge v > 0\}$. Its projection on $\{\mathtt{h}\}$ is : $\mathbf{pr}_{\{\mathtt{h}\}}(a) = \{h \mid h > 0\}$.

Definition 14 (Inverse Projection)**.** *For P defined on $d1$ and $d1 \subseteq d$.* $\mathbf{pr}_d^{-1}(P)$ *is the set of all behaviors defined on d that project in P when restricted on $d1$.*

For example, the inverse projection of the assertion $a = \{h \mid h > 0\}$ on $\{\mathtt{h}, \mathtt{v}\}$ is $\{(h, v) \mid h > 0\}$.

Contract Extension We lift the definition of inverse projection to contracts by using the operator on both the assumption and guarantee. For contracts and components, we call this operation "extension", as it is the extension of a contract's alphabet to a larger alphabet.

Definition 15. *For $c = (A, G)$ a contract defined on $d1$ with $d1 \subseteq d$, $\mathbf{pr}_d^{-1}(c) = (\mathbf{pr}_d^{-1}(A), \mathbf{pr}_d^{-1}(G))$ is the extension of c to alphabet d.*

Component Variable Elimination If a variable is only internal to a component, we can choose to remove it. This allows us to abstract the component while still retaining its external characteristics. To eliminate a variable of a component we project its assertion to the alphabet with the variable removed.

Definition 16 (Component variable elimination)**.** *For σ a component defined on an alphabet V and a variable $a \in V$, $\mathbf{pr}_{V \setminus a}(\sigma)$ is the component with a eliminated.*

Contract variable elimination Similarly, we can choose to remove a variable from a contract. This is useful if the variable is internal to the component that is specified by the contract, and we don't want to specify it through the contract. In order to eliminate a variable of a contract, first we project the guarantee to the alphabet from which the variable has been removed. Then, we project the assumption on the same alphabet, but only keeping the behaviors in which the variable was not relevant. Namely, we take the set of behavior, which every value of the new variable appears in the original set. This is done to only accept behaviors where the eliminated variable could have any value. We call this second kind of projection a projection-$\forall$, we give its formal definition below. The name projection-$\forall$ comes from an analogy we make from sets to statement. For a set $s = \{(x, y) \mid P(x, y)\}$ the projection-$\forall$ is the set $\mathbf{pr}^{\forall}_{\{x\}}(s) = \{x \mid \forall y,\, P(x, y)\}$. We also give an example in Figure 3.2.

Definition 17 (Projection $\forall$)**.** *For an assertion P defined on V, and $V' \subseteq V$, $\mathbf{pr}^{\forall}_{V'}(P)$ is the set of all behaviors whose extensions (to V) are in P.*

Definition 18 (Contract variable elimination)**.** *For $C = (A, G)$ a contract defined on alphabet V and a variable $a \in V$, $C' = (\mathbf{pr}^{\forall}_{V \setminus a}(A), \mathbf{pr}_{V \setminus a}(G))$ is the contract with a eliminated.*

3.2.3 The Meta-Theory of Contracts

This section introduces the meta-theory of contracts [Ben+15b] which is a more general theory, encompassing assumption/guarantee contracts as an instance or a specialization, but also encompassing other specializations such as the interface theory. This implies that the definitions of this section will be similar to the definitions above Yet, they are more abstract and typically not constructive. The main difference is that in the meta-theory the components are an abstract parmeter, whereas in the assumption/guarantee contracts, the components are

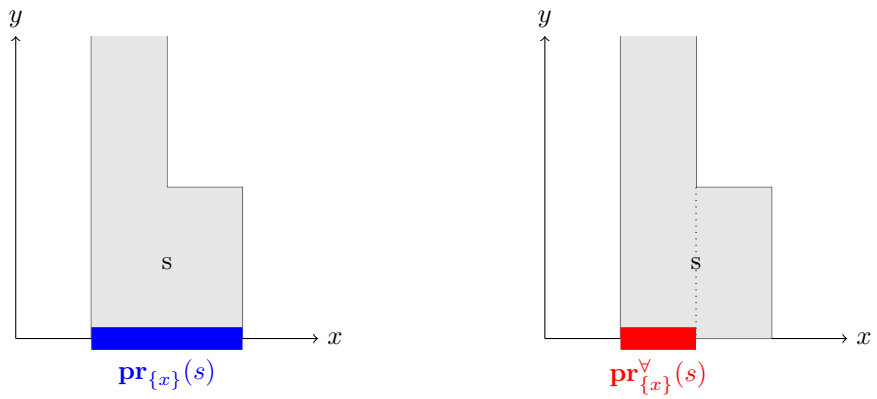


Figure 3.2: An example of projection on the left and projection-$\forall$ on the right

defined as sets of behaviors. After introducing the notion of components as a parameter in the Meta-Theory, we show how contracts and algebraic operators are defined.

Components as Parameters of the Meta-Theory

Component In the meta-theory of contracts, there is only one parameter, that of component. It is deeply related to component-based design, components are combined and can be composed to create new components. Components are kept abstract, their construction is not defined as opposed to previous definitions in the assumption/guarantee theory of contracts. In practice, a meta-theoretic or abstract component is meant to represent an element of a system performing a specific task. In the remainder, such components are noted by the letter σ.

Composition of Components We note $\sigma_1 \times \sigma_2$ the composition of σ_1 and σ_2. We assume this operator to be total and commutative. Again, we don't give the definition of the composition of components as the definition of components is abstract (a parameter).

Environment As in the assumption/guarantee contract theory, when composing $\sigma_1 \times \sigma_2$, we call σ_2 an environment for σ_1. When considering a component we call the rest of the system the environment, yet formally, environment is just another name for components.

Component Specification

Contract A contract, in the meta-theory, is the specification of a part of a system performing a certain task, similarly to the definition in component-based design. It is also a specification of the environment required for the task to be correctly performed. Formally, a contract is a set of components capable of

performing the task, and the set of possible environments in which it can be performed. This definition is a generalization of Definition 2.

Definition 19 (Contract Meta-Theory)**.** *A contract is a pair* $C = (\mathcal{E}, \mathcal{M})$ *with* $\mathcal{M}$ *a set of components and* $\mathcal{E}$ *a set of environments for every component in* $\mathcal{M}$*.*

Implements and Provides We define the relation of implementation for the Meta-Theory. This is similar to the definitions in assumption/guarantee contracts, Definition 4 and Definition 5. Since the theories are so close, we use the same name for the operators, to avoid confusion we note $\vdash_{MT}$ (respectively $\vdash^{P}_{MT}$) for the implements (respectively provides) relation in the meta-theory.

Definition 20 (Implements Meta-Theory)**.** *For a contract* $C = (\mathcal{E}, \mathcal{M})$ *and a component* σ*,*

$$\sigma \vdash_{MT} \equiv \sigma \in \mathcal{M}.$$

.

Definition 21 (Provides Meta-Theory)**.** *For a contract* $C = (\mathcal{E}, \mathcal{M})$ *and an environment* e*,*

$$e \vdash^{P}_{MT} C \equiv e \in \mathcal{E}.$$

Refinement The goal of a specification is to be an abstraction of actual components. Multiple contracts can be specifications of the same component on different levels of abstraction. We introduce the *refinement* relation to describe that a contract is the abstraction of another. Here C_1 is the refined version of C_2. This means that any implementations of C_1 can be used in place of an implementation of C_2.

Definition 22 (Refinement Meta-Theory)**.** $C_1 \preceq_{MT} C_2 \equiv \mathcal{M}_1 \subseteq \mathcal{M}_2 \wedge \mathcal{E}_2 \subseteq \mathcal{E}_1$

Conjunction Sometimes, multiple specifications can be applied to the same component. For example, we want a component to be fast (efficient) and safe (effective). These are specifications regarding the runtime speed of the component and its functional correctness. We would like to regroup both specifications in the same contract. Any contract which refines both contracts can be used, though it is here more desirable to use the most abstract one. With this in mind, we define the conjunction of contracts as the greatest lower bound of refinement.

Definition 23 (Conjunction)**.** $C_1 \sqcap_{MT} C_2$ *is the greatest lower bound* C_1 *and* C_2 *regarding the refinement.*

Composition For now, we only considered one component in its environment. Yet, most of the complexity of system design comes from the composition of multiple components. The problem can be formulated as *"If we have the specification of two components, can we determine the specification for their composition?"*. The subtlety is that each component is part of the environment of the other, hence the definition:

Definition 24 (Composition)**.**

$$C_1 \otimes_{MT} C_2 \equiv \min_{\preceq_{MT}} \left\{ C \;\middle|\; \begin{array}{ll} \forall M_1 \vdash_{MT} C_1 & \\ \forall M_2 \vdash_{MT} C_2 & \implies \\ \forall E \vdash^{P}_{MT} C & \end{array} \begin{array}{l} M_1 \times M_2 \vdash_{MT} C \\ E \times M_2 \vdash^{P}_{MT} C_1 \\ E \times M_1 \vdash^{P}_{MT} C_2 \end{array} \right\}$$

Quotient The quotient is the inverse of the composition. C_1/C_2 combined with C_1 refines C_2. It is a really powerful tool for system engineering.

Definition 25 (Quotient)**.** *C_1/C_2 is the contract such that*

$$\forall C, C \preceq_{MT} C_1 \otimes_{MT} C_2 \iff C \preceq_{MT} C_1/C_2$$

Compatibility and Consistency The definition of compatibility and consistency is more natural in the meta-theory. A contract is consistent if a component implements it and compatible if an environment provides it. Lifting compatibility and consistency on pairs of contracts gives a more intuitive vista on the definition. We say that two contracts are compatible (respectively consistent) if their composition is compatible (respectively consistent).

Definition 26 (Consistency)**.** *A contract C is consistent if $\exists M,\ M \vdash_{MT} C$.*

Definition 27 (Compatibility)**.** *A contract C is compatible if $\exists E,\ E \vdash^{P}_{MT} C$.*

3.2.4 Specialization

We now have two theories of contracts that differ in their definitions of implementation, refinement, conjunction, and composition. The meta-theory has an intuitive definition, while the assumption/guarantee theory has more concrete and constructive properties. In this section, we define the properties needed to prove that assumption/guarantee contracts are an instance or a specialization of the meta-theory of contracts. The idea is to replace the notion of implementation in the meta-theory with the implementation defined in the assumption/guarantee contract theory. By doing that, we can check that the characterisation of the definitions in the meta-theory holds for the definitions in the assumption/guarantee contract theory.

Refinement To show that the refinement in assumption/guarantee contract is correct, we must prove that it is an order relation regarding the implementation (Theorem 1). Also, if two contracts refines each other by Definition 7, they also refines each other by Definition 22, with the meta-theory implementation relation replaced by the A/G implementation (Theorem 2).

Theorem 1 (A/G Refinement Order)**.** *$\preceq$ is reflexive, anti-symmetric and transitive.*

let a := (c1'.A ∩ c2'.A) ∪ ¬ g in

Theorem 2 (A/G Refinement Sound)**.** *If* $C_1 \preceq C_2$ *then*

$$\forall M, \quad M \vdash C_1 \implies M \vdash C_2$$

and

$$\forall E, \quad E \vdash^P C_2 \implies E \vdash^P C_1$$

Conjunction We must verify that the definition of the conjunction in the A/G contract theory is the greatest lower bound of two contracts, as it is defined in the Meta-Theory in Definition 23.

Theorem 3 (A/G Conjunction Correction)**.** $C_1 \sqcap C_2$ *is the greatest contract that refines* C_1 *and* C_2

Composition of Contracts To verify that the composition of contract in the assumption/guarantee theory (Definition 9) corresponds to the definition in the meta-theory, we must check that the constructed contract is the minimum of the set described in Definition 24, with the implementation relation instantiated by the implementation of assumption/guarantee contract. First, we check that the it is in the set (Theorem 4), then that it refines every contract in the set (Theorem 5).

Theorem 4 (A/G Composition in set)**.**

$$C_1 \otimes C_2 \in \left\{ C \;\middle|\; \begin{array}{lcl} \forall M_1 \vdash C_1 & & M_1 \times M_2 \vdash C \\ \forall M_2 \vdash C_2 & \implies & E \times M_2 \vdash^P C_1 \\ \forall E \vdash^P C & & E \times M_1 \vdash^P C_2 \end{array} \right\}$$

Theorem 5 (A/G Composition minor set)**.**

$$\forall C \in \left\{ C \;\middle|\; \begin{array}{lcl} \forall M_1 \vdash C_1 & & M_1 \times M_2 \vdash C \\ \forall M_2 \vdash C_2 & \implies & E \times M_2 \vdash^P C_1 \\ \forall E \vdash^P C & & E \times M_1 \vdash^P C_2 \end{array} \right\}, \qquad C_1 \otimes C_2 \preceq C$$

Consistency and Compatibility Verifying the definitions of consistency and compatibility in A/G contracts consists in proving Theorem 6 and Theorem 7.

Theorem 6 (A/G Consistency Correct)**.** *For a contract* $C = (A, G)$*,*

$$\exists s,\, s \in \overline{A} \cup G \iff \exists \sigma, \sigma \vdash_{MT} C$$

Theorem 7 (A/G Compatibility Correct)**.** *For a contract* $C = (A, G)$*,*

$$\exists s,\, s \in A \iff \exists e, e \vdash^P C$$

3.3 Mechanization of A/G Contracts in Coq

The meta-theory intends to provide a generic contract theory that can be instantiated with several logics. Each logic presents some features to enable or facilitate the verification of domain-specific properties in a system. Proving that several logics implement the same meta-theory is a way to unify them. Here, we formalize the assume/guarantee contract theory and prove that it corresponds to the definitions given in the meta-theory. The code of this section is available in the file `VCT/Contracts.v` of the library available at: `https://gitlab.inria.fr/skastenb/differential-contracts`.

The contract theory relies on set-theoretic definitions. At this stage, we assume the abstract type `set : Type → Type` which is constructed, from any type (`T`), a set type (`set T : Type` equipped with the usual set operators as $\cup$, $\cap$, $\neg$. We also assume the relations $\in$ and $\subseteq$ as well as the set equivalence `s1 == s2` which is extended to the standard equality `s1 = s2` by extensionality.

3.3.1 The Types for Variables

We use two abstract types to parameterize our theory, `ident` and `value` We will instantiate them in Chapter 4 to use the theory with differential dynamic logic.

Identifiers The `ident` is the identifier used for the variables. It is usually a string of characters, but can also be a natural number, or anything else. In Section 3.2, `ident` were the elements of set $\mathcal{V}$.

Values The type `value` code the values held by variables. It typically is real, naturals or booleans. Here, we only consider one type of value, for the sake of simplicity in our reasoning, which is however not realistic, as a real component may manipulate different type of variables.

3.3.2 Requirements

We have some requirements on the base types. All theses requirements are summarized in the type class `NonEmptyDecidableVariable`. Indeed, we need to be able to determine if a identifier belongs to a set. This is the `in_dec_ident` function. We also need a default value `any_value` for the extension of alphabet of behaviors. Both of the requirements should be computed when the theory is instantiated.

```
Class NonEmptyDecidableVariable (value ident : Type) := {
  any_value : value ;
  in_dec_ident : ∀ (v : ident) (d : set ident), {v ∈ d} + {v ∉ d}
}.
```

3.3.3 Objects Definitions

Alphabet We define an alphabet as a set of identifiers, as in Section 3.2.

```
Definition alphabet : Type := set ident.
```

This section considers `d` as the unique set of identifiers used in the system. Most of the definitions we give below are parameterized by it. It is an implicit parameter for every definition using `d`. We see in Section 3.5, how to manipulate components who are defined on different alphabets.

Variable The type `var` is the type of a variable, namely an identifier with the proof that it inhabits `d`. It is different of an identifier in the sense that we know it is relevant in the context. This is a particular point in the mechanization that doesn't appear in mathematical definitions. The notation { `x : A | P x`} denote the sigma type of an element `x` with property `P x`.

```
Definition var := { v : ident | v ∈ d }.
```

Behavior A behavior is a function that associates a value to each variable. As in the mathematical definition, we only define the behavior on variables that are in the alphabet. This is why we have to define the `var` type as it the way to define a function on a subset of `ident`.

```
Definition behavior : Type := var → value.
```

Assertion We define assertions as behavior predicates using the duality of sets, namely, `s ∈ q` denotes that `s` satisfies the assertion `q`. This is exactly the same as in Section 3.2.

```
Definition assertion : Type := set behavior.
```

Components At this stage of the development, we assimilate the concept of a component with its behavior as in the previous section. The same is true for environments, which are another name for components. The name environment is used to refer to components that act as the external environment of a component.

```
Definition component := assertion.
Definition environment := component.
```

Contracts Contracts are directly defined as pairs of assertions relating the behavior expected from the environment (assumption) with the behavior of the component (guarantee). The syntax `c.A` (and `c.G`) denotes the assumption (respectively guarantee) of the contract `c` in the rest of the paper[4]. As in Definition 2:

```
Record contract : Type := mkContract {A : assertion ; G : assertion}.
```

[4]Coq original syntax is `c.( A)` but we replaced it for the sake of readability.

3.3.4 Relations

Behavior satisfaction The semantics of the contract relies on the implementation of a contract by a component. In order to define it, we first introduce the satisfiability of the contract by a single behavior and the saturation principle.

```
Definition satisfies (s : behavior) (c : contract) : Prop :=
    s ∈ ¬ c.A ∪ c.G.
```

Basically, a behavior satisfies a contract either if the behavior is discarded by the assumption then nothing is guaranteed by the contract, or the behavior satisfies both assumption and guarantee of the contract.

Saturation In the following code, we saturate contracts when necessary: it is easier to saturate a contract than to check if it is already saturated. By contrast, in the mathematical definition, it is easier to consider every contract to be saturated rather than to saturate them all the time. The `saturate` function is the implementation of the Definition 6.

```
Definition saturate (c : contract) : contract :=
    {| A := c.A ;
       G := ¬ c.A ∪ c.G |}.
```

We verify that the contract saturation is sound, indeed the same behavior are characterized before and after the saturation of any contract.

```
Theorem saturate_sound : ∀ (s : behavior) (c : contract),
    satisfies s c ↔ satisfies s (saturate c).
```

Implementation We extend the contract satisfiability from behaviors to components to define the implementation relation. Additionally, we also need to characterize the relationship between contracts and environments. This corresponds to Definitions 4 and 5.

```
Definition implements (σ : component) (c : contract) : Prop :=
    ∀ s, s ∈ σ → satisfies s c.
Notation "σ ⊢ c" := (implements σ c).
Definition provides (e : environment) (c : contract) : Prop :=
    e ⊆ c.A .
```

Refinement Then, we define the refinement relation on contracts. Here, it is important to note that we are implementing Definition 7 of the assume/guarantee theory of contracts. The refinement and composition relations are defined differently in the meta-theory and in the assume/guarantee theory. We show in Section 3.4 that this definition is equivalent to the definition in the meta-theory. Note that, in line with the saturation principle discussed above, we saturate the contracts before calculating the refinement relation.

```
Definition refines (c1 c2 : contract) : Prop :=
    let (c1' , c2') := (saturate c1 , saturate c2) in
    c2'. A ⊆ c1'.A ∧ c1'.G ⊆ c2'.G.
Notation "c1 ⪯ c2" := (refines c1 c2).
```

Conjunction The conjunction of contracts corresponds to multiple views one can have on the same component. This is the exact translation of Definition 8.

```
Definition glb (c1 : contract) (c2 : contract) : contract :=
    let c1' := saturate c1 in let c2' := saturate c2 in
    {| A := (c1'. A ∪ c2'.A) ;
       G := (c1'. G ∩ c2'.G) |}.
Notation "c1 ⊓ c2" := (glb c1 c2).
```

Composition The central operator in the contract algebra is composition. Two contracts can be composed if they are defined on the same variables. The composition of components aims at constructing a contract specifying the composition of components. We implement the A/G contract definition as in Definition 9 and then show we will that it corresponds to the meta-theory Definition 24.

```
Definition compose (c1 c2 : contract) : contract :=
    let c1' := saturate c1 in
    let c2' := saturate c2 in
    {| A := (c1'. A ∩ c2'.A) ∪ ¬ g ;
       G := c1'. G ∩ c2'.G |}.

Notation "c1 ⊗ c2" := (compose c1 c2).
```

Compatibility and Consistency To verify that our contract is appropriate, we have two tools: compatibility and consistency. Compatibility represents the ability of the contract to be used in an environment. Whereas consistency represents the possibility of implementing the contract by a component. We can easily define the compatibility and the consistency as a direct translation of Definition 12 and 11. Since Coq is based on constructive logic, the only way to express that a set is not empty is to exhibit an element that belongs to the set.

```
Definition is_compatible (c : contract) : Prop := exists s : behavior, s ∈ (c.A).

Definition is_consistent (c : contract) : Prop := exists s : behavior, satisfies s c.
```

3.3.5 Outlook

We presented the mechanization of the A/G contract theory. We pointed out the differences between the mathematical definitions and the Coq implementation.

- In the mechanization we saturate contract when needed, whereas in the mathematical definition we consider the contract to be saturated beforehand.
- Variables are defined with the proof that they're in the alphabet.

In the rest of the chapter, we present four different axes of improvement of the mechanization:

Equivalence with the meta-theory In the next section, we show how to prove that the above mechanization is a correct specialization of the meta-theory of contracts. We focus on each relation and show the connection between its definition in the mechanization and its definition in the meta-theory of contracts.

Alphabet Equalization In Section 3.5, we present how to handle the alphabets, and contracts that are defined on different alphabets.

Elimination Variable In Section 3.6, we introduce the mechanisation of the process to abstract a contract by eliminating one of the variables.

Formula interface In Section 3.7, we introduce an interface for implementing A/G contracts more easily. Indeed, the implementation and uses of contracts would be a bit cumbersome without the interface. The interface permits to define and use contracts more easily at the cost of generality, namely not every implementation of the theory can be done using the interface.

3.4 Consistency of A/G Contracts with the Meta-Theory

In this section, we detail the structure and mechanization of the proof that assumption/guarantee contract theory is a specialization of the meta-theory of contracts. First, since it is necessary to prove the equality of two contracts, we use functional extensionality to define the extensionality of contracts. Indeed, proving that assumption/guarantee contract theory is a specialization of the meta-theory of contracts requires proving that some contracts are equal.

Contract extensionality Adding functional extensionality makes it possible to prove behavior extensionality and contract extensionality, i.e., to prove that the contracts are equal. We have admitted functional and propositional extensionality already for the library of sets in Section 3.1. This means we can prove that when two contracts refine each other, their saturated versions are equal. In the following, we won't bother showing that two contracts are equal but we merely show that they refine each other. The equality of their saturated version follows directly, from theorem `contract_extensionality`. To prove the contract

extensionality, we need to prove the extensionality on behavior first, which directly derives from the functional extensionality.

```
Definition behavior_equiv (s1 : behavior) (s2 : behavior) : Prop :=
  ∀ x : {v : ident | v ∈ d}, s1 x = s2 x.

Theorem behavior_extensionality : ∀ (s1 s2 : behavior),
  behavior_equiv s1 s2 → s1 = s2.

Definition equiv (c1 : contract) (c2 : contract) : Prop :=
    refines c1 c2 ∧ refines c2 c1.

Notation "c1 ≍ c2" := (equiv c1 c2) (at level 70 , no associativity).

Theorem contract_extensionality :
    ∀ (c1 c2 : contract), c1 ≍ c2 → saturate c1 = saturate c2.
```

Refinement Since `refines` is an order, we prove the usual properties: reflexivity, transitivity, and anti-symmetry. We also demonstrate that the definition of refinement for A/G contracts is equivalent to the more standard and meaningful Definition 22 of the refinement given by the meta-theory.

```
Theorem refines_correct : ∀ (c1 c2 : contract),
    c1 ⪯ c2 ↔
    (∀ σ: component, σ ⊢ c1 → σ ⊢ c2) ∧
    (∀ e: environment, provides e c2 → provides e c1).
```

To prove the correctness of $\preceq$ (namely, Theorems 1 and 2), it suffices to see that the definitions are similar. By unfolding the definitions of `refines`, `implements` and `provides`, the theorem `refines_correct` can be rewritten as Equation 3.3. The lemma "$\forall S, S \subseteq A \implies S \subseteq B \iff A \subseteq B$" finishes the proof of `refines_correct`.

$$c2.A \subseteq c1.A \wedge c1.G \subseteq c2.G \iff \forall e \subseteq c2.A \to e \subseteq c1.A \wedge \forall \sigma \subseteq c1.G \to \sigma \subseteq c2.g \tag{3.3}$$

Conjunction In the meta-theory, the conjunction is defined by the greatest lower bound of refinement. So, we prove that our set definition is equivalent to the meta-theoretical Definition 23 as in Theorem 3. The proof is a direct consequence of the definition of $\sqcap$. It suffices to unfold the definitions of `refines` and $\sqcap$ to finish the proof.

```
Theorem glb_correct : ∀ c1 c2 : contract,
    (c1 ⊓ c2) ⪯ c1 ∧ (c1 ⊓ c2) ⪯ c2 ∧
    (∀ c, c ⪯ c1 → c ⪯ c2 → c ⪯ (c1 ⊓ c2)).
```

Composition Here, we give the proof that the definition of composition corresponds to Definition 24 given in the meta-theory. Precisely `compose_correct` is the mechanization of Theorem 4, and `compose_lowset` is Theorem 5.

```
Theorem compose_correct :
    ∀ (c1 c2 : contract) (σ1 σ2 : component) (e : environment),
    σ1 ⊢ c1 → σ2 ⊢ c2 → provides e (c1 ⊗ c2) →
    (σ1 ∩ σ2 ⊢ c1 ⊗ c2 ∧ provides (e ∩ σ2) c1 ∧ provides (e ∩ σ1) c2).

Theorem compose_lowest : ∀ (c1 c2 c : contract),
    (∀ (σ1 σ2 : component) (e : environment),
    σ1 ⊢ c1 → σ2 ⊢ c2 → provides e c →
    (σ1 ∩ σ2 ⊢ c ∧ provides (e ∩ σ2) c1 ∧ provides (e ∩ σ1) c2)) →
    c1 ⊗ c2 ⪯ c.
```

Compatibility and Consistency We verify that the definitions of compatibility and consistency are correct. Unfolding the definitions of consistency gives us Equation 3.4 which is trivially true. The important point is to always define non-emptiness as the existence of an element belonging to the set.

$$\exists x, x \in A \qquad \Longleftrightarrow \qquad \exists E, (\exists x, x \in E) \wedge E \subseteq A \tag{3.4}$$

```
Theorem consistent_correct : ∀ c : contract,
  is_consistent c ↔ exists σ, is_not_empty σ∧ implements σc.

Theorem compatible_correct : ∀ c : contract,
  is_compatible c ↔ exists e, is_not_empty e ∧ provides e c.
```

3.5 Alphabet Equalization

This section considers the case when contracts or components are defined over different alphabets of variables. This poses a problem when we need to compose them, or do any operation on two contracts or components that are not defined on the same variables. In that case, we need a way to extend the alphabet of the object on the union of their variables. With both contracts or components extended, we can use the operators defined previously.

We give a set-theoretic definition in Coq of the extension of alphabet for components and contracts. We assume the existence of `D`, the set of all variables used in the system. We also assume other alphabets `d1`, `d2` as subsets of `D`.. We also assume `H1 : d1 ⊆ D` and `H2 : d2 ⊆ D`, the proofs that `d1` and `d2` are a subsets of `D`.

3.5.1 Definition of Extension

Changing Alphabet of Var Since our variables are typed with the alphabet on which they are defined, we need to do some work to transform a variable defined

on alphabet `d1` into a variable defined on `D`. First, we define `H'1 : var d1 → var D`, which takes a variable in `d1` and shows that it is also a variable in D.

```
Definition H'1 (v1 : var d1) : var d2 := let (i,P) := v1 in exist _ i (H1 i P).
```

Here, we use `H1` to show that the ident `i` in `d1` is also in `d2`. Indeed, the type of `H1` is `∀ v : ident, v ∈ d1 → v ∈ d2`. Hence, `H1 i H` is of type `i ∈ d2`, and `exist _ v (H1 i H)` is of type `var d2`.

Projection of a Behavior We define the projection of a single behavior defined on alphabet `D` on a smaller alphabet `d1`.

```
Definition project (s : behavior D) : behavior d1 :=
  fun v1 ⇒ s (H'1 v1).
```

Projection of an Assertion We define the projections of assertions by extending the projection behaviors. Indeed, the projection of an assertion on alphabet `d1`, is the projection of every behavior in the assertion to alphabet `d1`. If we consider the assertion as the property it holds on variables, projection is similar to an existential quantification. For example, the assertion $P(x, y)$ projected on the variable $\{y\}$ is $\exists x, P(x, y)$.

```
Definition project_assertion (a : assertion D) : assertion d1 :=
  fun s1 ⇒ exists s, s ∈ a ∧ project s = s1.
```

Extension of the Alphabet on Behavior Then, we can define the inverse of the projection which, for a behavior, gives the set of behaviors that project to it. Notice that the extension of a behavior gives an assertion. Indeed, multiple behaviors defined on `D` have the same projection on `d1`.

```
Definition extend_behavior (s1 : behavior d1) : assertion D :=
  fun s ⇒ project s = s1.
```

Extension of the Alphabet on Assertion As for the extension of behaviors, extending an assertion `a1` is done by taking every behavior that projects to a behavior in `a1`.

```
Definition extend_assertion (a1 : assertion d1) : assertion D :=
  fun s ⇒ project s ∈ a1.
```

Extension of the Alphabet on Contract The extension of the alphabet on which a contract is defined is the extension of both the assumption and the guarantee.

```
Definition extend_contract (c1 : contract d1) : contract d2 :=
  let c1' := saturate _ c1 in
  {| A := (extend_assertion (c1'.A)) ;
     G := (extend_assertion (c1'.G)) |}.
```

3.5.2 Definition of Extended Operators

In this section, we define operators to deal with a contract defined on different alphabets. We define the operators of the theory of contracts as in Section 3.2 and Section 3.3.

Implementation and Provides Extended In this definition, we consider a component σ to be defined on `D` and a contract `c1` defined on `d1`. We define the implementation of the contract by the component. Similarly, we define the extended definition of `provides` with an environment `e` defined on `D`.

```
Definition implements_ext (H1 : d1 ⊆ D) (σ : component D) (c1 : contract d1) : Prop :=
  implements D σ(extend_contract H1 c1).

Definition provides_ext (H1 : d1 ⊆ D) (e : environment D) (c1 : contract d1) : Prop :=
  provides _ e (extend_contract H1 c1).
```

We introduce notations for `implements_ext` and `provides_ext`. Since we will need to use them to evaluate implementation and provides on contract defined on `d1` and `d2`, we introduce two notations for both.

```
Notation "σ ⊢1 c" := (implements_ext H1 σc) (at level 70, no associativity).
Notation "σ ⊢2 c" := (implements_ext H2 σc) (at level 70, no associativity).

Notation "e ⊢p1 c" := (provides_ext H1 e c) (at level 70, no associativity).
Notation "e ⊢p2 c" := (provides_ext H2 e c) (at level 70, no associativity).
```

Refinement of Contracts on Different Alphabet The extended refines is defined by extending the alphabet of both contracts on `D`. This is useful if, for example, one wants to show that a contract without a variable is an abstraction of a contract with that variable. To validate that the definition corresponds to the meta-theory of contracts, we can use the same validation theorem we used in Section 3.5. The proof is direct by reusing proofs `refines_correct` on `extend_contract c1` and `extend_contract c2`.

```
Definition refines_extended (c1 : contract d1) (c2: contract d2) : Prop :=
  refines _ (extend_contract H1 c1) (extend_contract H2 c2).
Notation "c1 ⪯e c2" := (@refines_ext _ _ D H1 H2 c1 c2) (at level 70, no associativity).

Theorem refines_ext_correct : forall (c1 : contract d1) (c2 : contract d2),
  c1 ⪯e c2 ↔
  (∀ σ: component D, σ ⊢1 c1 → σ⊢2 c2) ∧
  (∀ e : environment D, e ⊢p2 c2 → e ⊢p1 c1).
```

Composition from different alphabet We can define the composition on two contracts defined on different variables with the same process we use to defined the extended refine relation. The validation is the same as for the refinement, we use the proof of `compose_correct` on `extend_contract c1` and `extend_contract c2`.

```
Definition compose_ext (H1 : d1 ⊆ D) (H2 : d2 ⊆ D)
  (c1 : contract d1) (c2 : contract d2) : contract D :=
  compose _ (extend_contract H1 c1) (extend_contract H2 c2).

Notation "c1 ⊗e c2" := (compose_ext H1 H2 c1 c2) (at level 61, left associativity).

Theorem compose_ext_correct : forall (c1 : contract d1) (c2 : contract d2)
  (σ1 : component D) (σ2 : component D) (e : environment D),
  σ1 ⊢1 c1 → σ2 ⊢2 c2 →
  e ⊢p (c1 ⊗e c2) →
  ((σ1 ∩ σ2) ⊢(c1 ⊗e c2) ∧ (e ∩ σ2) ⊢ p1 c1 ∧ (e ∩ σ1) ⊢ p2 c2).
```

Conjunction extended Similarly, to prove that the conjunction extended is correct, we prove the same properties that we proved for the conjunction in Section 3.4.

```
Definition glb_ext (c1 : contract d1) (c2 : contract d2) : contract D :=
  glb _ (extend_contract H1 c1) (extend_contract H2 c2).

Notation "c1 ⊓e c2" := (glb_ext H1 H2 c1 c2) (at level 61, left associativity).

Theorem glb_ext_correct : forall (c1 : contract d1) (c2 : contract d2),
       (c1 ⊓e c2) ≼(extend_contract H1 c1) ∧
       (c1 ⊓e c2) ≼(extend_contract H2 c2) ∧
       (forall c : contract D,
        c ≼(extend_contract H1 c1) →
        c ≼(extend_contract H2 c2) →
        c ≼(c1 ⊓e c2)).
```

3.6 Elimination of Variables

In certain situations, it may be useless to keep some variables specified. For example, if a component provides the input for another component, then the composition of their contracts may not need that variable to be specified anymore. In that case, we want to eliminate the spurious variable from the contract. In this section, we provide a concrete definition of elimination of variable and give a validation of the definition.

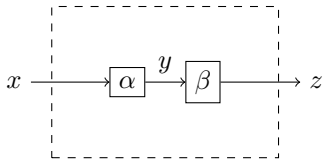


Figure 3.3: The composition of α β, the variable y can be eliminated since it is not visible by the outside

Mathematical insight In [Ben+08], the elimination of variables is defined in the following way.

Definition 28. *For a contract $c = (A, G)$, and a variable v in the contract. The contract without v is :*

$$[c]_v \equiv (\forall v\ A\ ;\ \exists v\ G)$$

It gives a good intuition of the elimination of variables. A and G are both sets, the quantifier has no meaning for sets. To define this elimination of variable, the authors consider assertions as logic formulas and bind free variables with quantifiers. However, this shortcut cannot be taken in our mechanization of the assumption/guarantee theory of contract, because the sets are not equals to logic formulas. Hence, we introduce the necessary functions to define eliminations of variable in an abstract set theoretic contract theory.

Example The contract in Equation 3.5 is an example of contract where y could be eliminated as in Figure 3.3. To eliminate y we use the Definition 28, this give us Equation 3.6 By simplifying we obtain Equation 3.7.

$$(\{(x,y) \mid y^2x \geq 0\}, \qquad \{(x,y) \mid y^2x \geq 0 \to z \geq 0 \wedge z = y^2x\}) \tag{3.5}$$

$$(\{x \mid \forall y,\ y^2x \geq 0\}, \qquad \{x \mid \exists y,\ y^2x \geq 0 \to z \geq 0 \wedge z = y^2x\}) \tag{3.6}$$

$$(\{x \mid x \geq 0\}, \qquad \{x \mid x \geq 0 \to z \geq 0\}) \tag{3.7}$$

3.6.1 Definition

Elimination of Variable for Components To eliminate a variable in a component we can use the projection we defined in the previous section. This follows the Definition 13 given in Section 3.2.

```
Definition project_assertion (a : assertion D) : assertion d1 :=
  fun s1 ⇒ exists s, s ∈ a ∧ project s = s1.

Definition project_component (σ : component D) : component d1 :=
  project_assertion σ.
```

Projection-∀ To define the elimination of variable from a contract, we need to define the projection-∀ of an assertion. The projection-∀ of an assertion is the set of behaviors whose every extensions are in the assertion. The equivalent of the projection-∀ when viewing assertion as property is the ∀ quantifier. With the mathematical notation: the projection-∀ of $P(x, y)$ on $\{y\}$ is $\forall x,\ P(x, y)$. We define this projection in Coq, with a set theoretic definition.

```
Definition project_assertion_forall (a : assertion d2) : assertion d1 :=
    fun e1 ⇒ extend_behavior e1 ⊆ a.
```

Elimination of Variable for Contracts We can now define the elimination of variables in a contract. We use the projection-∀ on the assumption and the projection of the guarantee. This is the mechanization of Definition 18 with the assumption/guarantee contract definition of projection.

```
Definition project_contract (c2 : contract d2) : contract d1 :=
  let c2' := saturate _ c2 in
  {| A := project_assertion_forall c2'.A ;
     G := project_assertion c2'.G |}.
```

3.6.2 Validation of the Definition

Difficulties of Validation To the best of our knowledge, there is no clear way to validate the definition of variable elimination, other than by showing that the original contract is a refinement of that with an eliminated variable. We will show some properties about the projections, in relation to the extension, that should convince us this definition is correct. However, to fully prove the correctness of the definition, we believe a stronger argument should be made.

Projection is Smallest Abstracting By verifying that the projected contract is an abstraction of the initial contract, we validate that the variable elimination doesn't create restriction on the contract. We also show that the projection of contract is the most refined abstraction of the original contract. Indeed, since the projection is an abstraction, we want it to abstract as little as possible. By checking that no contract is refined by the projection and refines the initial contract, we ensure that the projection does not "over-abstract" the initial contract.

```
Theorem project_abstract : ∀ (c : contract D),
  refines D c (extend_contract H1 (project_contract H1 c)).
```

```
Theorem project_smallest_abstract : forall (c : contract D) (c1 : contract d1),
  refines D c (extend_contract H1 c1) →
  refines d1 (project_contract H1 c) c1.
```

3.7 Formula Interface

In this section, we define an interface for assumption/guarantee contracts. This interface is a practical tool to facilitate the process of instantiating the theory. It expects a language to express logical properties, along with basic operators and a satisfaction function. It provides a theory of contracts with operators defined in the supplied logical language. The theory is parameterized by a type class representing the logic.

3.7.1 Purpose of the instance

The Theory is Cumbersome to Use When instantiating the theory as described in Section 3.3, we face a practical problem. A lot of work lies in the definition of assertions, and Coq definitions do not make this easy to do. In the usual instantiation of the theory, the logic has an expression language with a deeply-embedded type, and operators to define an expression. To construct an assertion from an expression, we must use the satisfaction function, which builds a proposition from an expression and a state. This is usable, but when using operators on the assertion, the operations done don't reflect on the expression.

Factorise the Code This is why we want to factorise the code dedicated to transform expression into assertion. The idea of the interface is to abstract the type of expressions and the operators on it, and define a new contract theory, where the assertions are replaced by expressions. Then, we define the transformation of contracts with expressions into contracts with assertion by using the satisfaction function. In this section, we call contracts defined with expressions instead of assertions *expression contracts*, and contract with assertions: *assertion contracts*.

Relation between sat, expressions and assertions To abstract the expressions, we use a type class. The type class is parameterized by the type of the identifiers `ident`, the type of values `value` and the type of expressions `expr`. The user must give the satisfaction function `sat`, and the operators on expressions that correspond to the typical set operators `e_and`, `e_or` and `e_not`. They must also give the proof that this operators corresponds to the set of operators in regards to the satisfaction function `sat_e_and`, `sat_e_or` and `sat_e_not`. These proof terms are needed to prove the correctness of the interface.

Decidability of `sat` The last requirement is the decidability of the satisfaction function `sat_dec`. Indeed, this is needed to prove the equivalence between the satisfaction by a behavior of an expression contract and an assertion contract.

```
Class AlphabetizedExpression (value ident : Type) (expr : set ident → Type) := {
  e_and : ∀ d : set ident, expr d → expr d → expr d ;
  e_or : ∀ d : set ident, expr d → expr d → expr d ;
  e_not : ∀ d : set ident, expr d → expr d ;
  sat : ∀ d : alphabet, (@behavior value ident d) → expr d → Prop ;

  sat_e_not : ∀ (d : set ident) (s : behavior d) (e : expr d),
    ¬ sat d s e ↔ sat d s (e_not d e) ;
  sat_e_and : ∀ (d : set ident) (s : behavior d) (e1 e2 : expr d),
    sat d s e1 ∧ sat d s e2 ↔ sat d s (e_and d e1 e2) ;
  sat_e_or : ∀ (d : set ident) (s : behavior d) (e1 e2 : expr d),
    sat d s e1 ∨ sat d s e2 ↔ sat d s (e_or d e1 e2) ;
  sat_dec : ∀ (d : set ident) (s : behavior d) (e : expr d),
    sat d s e ∨ ¬ sat d s e ;
}.
```

3.7.2 Operations Definitions

Definition of a Contract The definition of an expression contract is pretty straightforward, we have an expression for the assumption and one expression for the guarantee. As in Section 3.3, the expression are parameterized by the alphabet they are defined on.

```
Record contractF := ContractF {A : expr d ; G : expr d}.
```

Saturation We define the saturation with respect to the operators of the parameter logic. This is similar to Definition 6 and to the Coq formalization in Section 3.3, but we use the operators of the logic instead of the usual operators.

```
Definition saturateF (cf1 : contractF) : contractF :=
  ContractF cf1.A (e_or d cf1.G (e_not d cf1.A)).
```

Composition The definition of the composition supposes the contract to be saturated, so we have to saturate them using the function we just defined. We follow the Definition 9, and use the expression operators.

```
Definition composeF (cf1 cf2 : contractF) : contractF :=
  let cf1' := saturateF cf1 in
  let cf2' := saturateF cf2 in
  let g := e_and d cf1'.G cf2'.G in
  let a := e_or d (e_and d cf1'.A cf2'.A) (e_not d g) in
    ContractF a g.
```

Conjunction Similarly to the composition definition, we use the saturation operator and follow Definition 9 to define the conjunction of expression contracts.

```
Definition glbF (cf1 cf2 : contractF) : contractF :=
  let cf1' := saturateF cf1 in
  let cf2' := saturateF cf2 in
  let a := e_or d cf1'.A cf2'.A in
  let g := e_and d cf1'.G cf2'.G in
  ContractF a g.
```

3.7.3 Equivalence with A/G Contracts

From ContractF to Contract To transform an expression contract (`contractF`) into an assertion contract (`contract`), we need the function `assertion_of`. `assertion_of` uses the satisfaction function to create the set of all behavior satisfying an expression. This set is, in the sense of the assume/guarantee theory of contract, an assertion.

```
Definition assertion_of (formula : expr d) : assertion d :=
  fun e ⇒ sat d e formula.
```

Then creating the assertion contract from the expression contract is only a matter of creating both assertion using `formula_to_assert`. Notice that we use the constructor for assertion contract `mkContract` and not the constructor for expression contract `ContractF`.

```
Definition contract_of (cf : contractF d) : contract d :=
  mkContract d (assertion_of cf.A) (assertion_of cf.G).
```

Correctness of Operators In the following, we prove that the operators we defined for expression contracts `composeF`, `saturateF` and `glbF` are equivalent to their equivalent for assertion contracts `compose`, `saturate` and `glb`. For example, we want to prove that the composition operators of assertion contracts `composeF` and expression contracts `compose` result in the same contracts. To prove that two contracts are equal, we first prove that they are equivalent. Then, thanks to the contract extensionality proved in section 3.4, we can conclude that they are equal.

The proofs of correctness of `saturateF`, `composeF` and `glbF`, are similar. We unfold the definitions, and uses the equivalences `sat_e_or`, `sat_e_and` and `sat_e_and` to transform the expression operators into the set operators, then the proof becomes trivial.

```
Theorem saturateF_correct : ∀ (cf : contractF d) ,
  (c2c (saturateF d cf)) == (saturate d (c2c cf)).

Theorem composeF_correct : ∀ (cf1 cf2 : contractF d),
  c2c (composeF d cf1 cf2) == compose d (c2c cf1) (c2c cf2).

Theorem glbF_correct : ∀ (cf1 cf2 : contractF d),
  c2c (glbF d cf1 cf2) == glb _ (c2c cf1) (c2c cf2).
```

3.8 Outlook

In this chapter, we presented our mechanization of the theory of assumption/guarantee contracts. We first gave an overview of the theory as it is described by Benveniste [Ben+15b] and its relation to the meta-theory of contracts. Then, we showed our Coq development that defined contracts, components and environments. We also showed our definition of saturation, implementation, composition. We then used the Coq proof assistant to validate the relation between the assumption/guarantee contracts and the meta-theory of contracts. By proving this relation, we validated our definitions of the assumption/guarantee theory of contracts. Then, we introduced extensions of the theory, with alphabet equalization, elimination of variables, and an interface. Alphabet equalization is the process used to manipulate objects defined on different alphabets, by extending their alphabet of definition. We gave the definitions of the operators, and we suggested a proof of their correction. We also introduced the elimination of variables, which is used to abstract a contract by removing variables. We checked

some properties on the elimination of variables on contract, but we are missing an argument to fully confirm its definition. Finally, we introduced an interface for the definition of assumption/guarantee contracts. This interface helps to instantiate the theory with a propositional logic, with code that would need to be rewritten for every instance.

The next step is to instantiate the assumption/guarantee contracts with a logic, in order to practically validate the formalization. We do this in Chapter 4, with differential dynamic logic, which is a good fit to describe cyber-physical systems.

Chapter 4

Contracts for Differential Dynamic Logic

In this chapter, we give two instances of our parametric contract theory by instantiating it with differential dynamic logic ($\mathbf{d}\mathcal{L}$), and show they can be used together to define components and contracts to model a cyber-physical system. The code presented in this chapter can found at the root of the library available here : `https://gitlab.inria.fr/skastenb/differential-contracts`. The first instance is in file `DifferentialContracts.v`, and the definition of abstract programs is in file `AbstractPrograms.v`.

First, in Section 4.1, we introduce differential dynamic logic and its deep embedding in Coq : Coq dL which we use to instantiate the theory. In Section 4.2, we introduce the running example of this chapter, which we will use to illustrate the instantiation of our contract theory to $\mathbf{d}\mathcal{L}$. Section 4.3 presents the first instantiation of our generic contract theory to define concrete $\mathbf{d}\mathcal{L}$ components. Section 4.4 shows the nature of refined components that arise from this process and the relation between that refinement relation and the differential refinement. Section 4.5 introduces the $\mathbf{d}\mathcal{L}$ notion of abstract programs as a mean to represent ($\mathbf{d}\mathcal{L}$) contracts by instantiating our generic contract theory. Finally we instantiate the verified contract theory with abstract contract and address the relations between the two instances in Section 4.6.

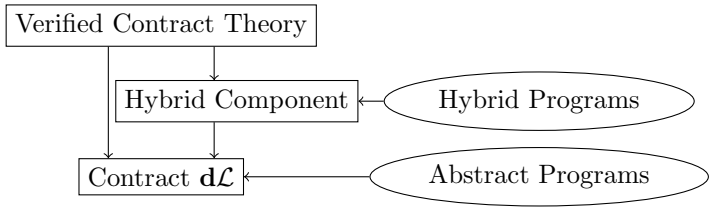


Figure 4.1: Architecture of the chapter

4.1 Introduction to Differential Dynamic Logic

Differential dynamic logic is a logic to model hybrid systems as well as a proof system on the logic. The logic has two main parts, hybrid programs which are sequences of instructions, and hybrid formulas. With hybrid programs, one can model both the discrete, computational part of a system and its continuous evolution. Hybrid formulas can be used as tests in hybrid programs or, with the help of modalities, they can be used to express properties about hybrid programs.

In this section, we introduce differential dynamic logic, its syntactic definitions and its semantics. We also provide graphical representation, to facilitate the understanding of hybrid programs. We base ourselves on the uniform substitution calculus for differential dynamic logic [Pla17] for the introduction to $\mathbf{d}\mathcal{L}$, and we also reproduce the Coq deep-embedding that has been made to verify the calculus: Coq dL [Boh+17]. We point out the differences between the mathematical definitions and their Coq mechanizations.

Mechanization of the Reals The semantics of an ordinary differential equation relies on real mathematics definition. While the standard library of Coq has an axiomatization of reals, it is a bit cumbersome to use for complex results. Thus Coq dL uses the library Coquelicot [BLM15], a more user-friendly extension of Coq for reals.

4.1.1 Syntactic Definitions

We first give the syntactic definitions of terms, formulas and programs. We introduce both the paper definition from [Pla17] and the formalization in Coq dL, we show their relations and their differences. We also give an informal semantics of the constructors.

Variables We define the set of variables $\mathcal{V}$. For each variable $x \in \mathcal{V}$, its differential is also in the set: $\dot{x} \in \mathcal{V}$. In the mechanization, the type of variable symbols is `KVariable`, and the type of variables is `KAssignable`. A `KAssignable` can either be non-differential with `KAssignVar` or the differential of a variable with `KAssignDiff`.

```
Inductive KAssignable : Set :=
    KAssignVar : KVariable → KAssignable
  | KAssignDiff : KAssignable → KAssignable
```

Example Here we show how to define the variables $h, \dot{h}$ and $\ddot{h}$.

```
Definition h : KAssignable := KAssignVar (variable "h").
Definition h' : KAssignable := KAssignDiff h.
Definition h'' : KAssignable := KAssignDiff h'.
```

State A state is a function mapping variables from $\mathcal{V}$ to real $\mathbb{R}$. We call $\mathcal{S}$ the set of states $\mathcal{S} = \mathcal{V} \to \mathbb{R}$. Notice, that since the function is total on $\mathcal{V}$, a state is defined on all possible variables. Since the systems are defined only using a few variables, we usually ignore the value of the variables not used. In the following, we give the definition of states in Coq, as well as a example of a state ν that evaluate to 1 for h and 0 for every other variable.

```
Definition state := KAssignable → R.
Definition nu : state :=
    fun x : KAssignable ⇒ if is_equal_KAssignable x h then 1 else 0.
```

Definition 29 (Terms)**.** *We define terms with the following grammar, $\theta_1 \dots \theta_n$ being terms, x a variable, r any number and f a function symbol.*

$$\theta_1, \dots, \theta_n ::= \quad x \mid r \mid \theta_1 + \theta_2 \mid \theta_1 - \theta_2 \mid \theta_1 \times \theta_2 \mid -\theta \mid \dot{\theta} \mid f(\theta_1, \dots, \theta_n)$$

The definition in Coq follows the same pattern with a few differences. The constructor `KTdot` is reserved for uniform substitution calculus, we won't use it

as we are not interested in the verification of the axiomatization of $\mathbf{d}\mathcal{L}$. The constructor `KTnumber` takes a `KTnum` which can either be a real or a natural number. The `KTfuncOf` takes a `FunctionSymbol`, a natural number `n` and a vector of size `n`. The `FunctionSymbol` will be evaluated with the interpretation function `I` which we will elaborate on later.

```
Inductive FunctionSymbol : Set := function_symbol : string → FunctionSymbol

Inductive Term : Type :=
| KTfuncOf      (f : FunctionSymbol)
                (n : nat)
                (a : Vector.t Term n) : Term      (* application of function symbol *)
| KTnumber      (r : KTnum)           : Term      (* number constant *)
| KTread        (var : KAssignable)   : Term      (* read variable x or diff. symbol x' *)
| KTneg         (child : Term)        : Term      (* negation      -x  *)
| KTplus        (left right : Term)   : Term      (* addition      x+y *)
| KTminus       (left right : Term)   : Term      (* subtraction   x-y *)
| KTtimes       (left right : Term)   : Term      (* multiplication x*y *)
| KTdifferential (child : Term)       : Term      (* differential x' *)
| KTdot         (n : nat)             : Term      (* dot symbol for terms *)
.
```

Example of Terms As an example we give the Coq definition of a term $H_{limit} = 15$, the term reading variable h and the cosines function of h. Although the cosines function is not defined here, we merely named the function, it has to be defined in the interpretation function `I` to be evaluated.

```
Definition HLimit : Term := KTnumber (KTNreal 15).
Definition h_term : Term := KTread h.
Definition cos : FunctionSymbol := function_symbol "cos".
Definition cos_h : Term := KTfuncOf cos 1 [ h_term ].
```

Definition 30 (Hybrid Formulas)**.** *Hybrid formulas are mutually defined with hybrid programs, we define hybrid programs below. Here φ and ψ are hybrid formulas, α is an hybrid program, θ_i are terms, C is a quantifier symbol and p is a predicate symbol.*

$$\begin{aligned}\varphi, \psi ::= \quad & \top \mid \bot \mid \theta_1 \sim \theta_2 \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \mid \varphi \leftrightarrow \psi \mid \\ & \forall x,\ \varphi \mid \exists x,\ \varphi \mid p(\theta_1, \ldots, \theta_n) \mid C(\varphi) \mid [\alpha]\varphi \mid \langle\alpha\rangle\varphi\end{aligned}$$

Where the symbol $\sim$ can be one of the relation symbols $\{<, >, \leq, \geq, =, \neq\}$.

The definition of hybrid formulas and hybrid programs is done by mutual induction. We first introduce formulas but since they are used in hybrid programs to express tests, both definitions must be done simultaneously. The formula $\top$ holds for every state, $\bot$ holds for none. The negation $\neg\varphi$ holds iff φ doesn't hold. We also have connectors for multiple formulas: $\varphi \wedge \psi$ holds iff both φ and ψ hold ; $\varphi \vee \psi$ holds iff either φ or ψ hold ; $\varphi \to \psi$ holds iff either φ doesn't hold

or ψ holds ; $\varphi \leftrightarrow \psi$ holds iff either both φ and ψ hold or they both doesn't hold. To quantify on variable x we have $\forall x,\ \varphi$ that holds iff φ holds for any valuation of x. We have the existential quantifier $\exists x,\ \varphi$ that holds iff it exists a valuation of x with which φ holds. The modality $[\alpha]\varphi$ expresses that φ holds after *all* runs of α. Whereas the modality of $\langle\alpha\rangle$ expresses that it exists a run of α after which φ holds. The formula $p(\theta_1, \ldots, \theta_n)$ holds iff the predicate p holds for the terms $\theta_1, \ldots, \theta_n$. The formula $C(\varphi)$ holds iff the quantifier C holds for the formula φ. A quantifier is an arbitrary function on formulas, it is a generalization the universal quantifier, the existential quantifier and the modalities.

Coq dL Definitions of Formulas The definition in the mechanization is similar to the Definition 30. We use `PredicateSymbol` and `QuantifierSymbol` to denote constant predicate and quantifiers, the symbols will be evaluated by the interpretation function `I`. The constructor `KFdot` is used internally for uniform substitution calculus, and is not used to construct terms.

```
Inductive PredicateSymbol : Set := predicate_symbol : string → PredicateSymbol
Inductive QuantifierSymbol : Set := quantifier_symbol : string → QuantifierSymbol
```

```
Inductive Formula : Type :=
| KFtrue                          : Formula          (* true  *)
| KFfalse                         : Formula          (* false *)
| KFequal      (left right : Term) : Formula          (* x == y *)
| KFnotequal   (left right : Term) : Formula          (* x != y *)
| KFgreaterEqual(left right : Term) : Formula         (* x >= y *)
| KFgreater    (left right : Term) : Formula          (* x > y *)
| KFlessEqual  (left right : Term) : Formula          (* x =< y *)
| KFless       (left right : Term) : Formula          (* x < y *)
| KFpredOf     (f : PredicateSymbol)
               (n : nat)
               (a : Vector.t Term n)                 : Formula
(* application of predicate symbol *)
(* predicational or quantifier symbol applied to argument formula child *)
| KFquantifier (f : QuantifierSymbol) (a : Formula) : Formula
| KFnot        (child : Formula)                     : Formula          (* ¬p *)
| KFand        (left right : Formula)                : Formula          (* p ∧ q *)
| KFor         (left right : Formula)                : Formula          (* p ∨ q *)
| KFimply      (left right : Formula)                : Formula          (* p → q *)
| KFequiv      (left right : Formula)                : Formula          (* p ↔ q *)
(* quantifiers *)
| KFforallVars (vars : list KVariable) (child : Formula)  : Formula          (* Forall x,y,z. p *)
| KFexistsVars (vars : list KVariable) (child : Formula)  : Formula          (* Exists x,y,z. p *)
(* modal formulas *)
| KFbox        (prog : Program) (child : Formula)         : Formula          (* [alpha] p *)
| KFdiamond    (prog : Program) (child : Formula)         : Formula          (* <alpha> p *)
| KFdot                           : Formula          (* dot symbol for formulas *)
```

Example of Formulas Here we give the Coq definition of the formula: $\varphi = h < H_{limit}$ and the formula `h_even` checking if h is a multiple of 2. The symbol `divides` needs to be evaluated by the interpretation function. The formula `cos_le_one` states that $\forall h,\ cos(h) \leq 1$.

```
Definition phi : Formula := KFless (KTread h) HLimit.

Definition divides : PredicateSymbol := predicate_symbol "divides".
Definition h_even : Formula := KFpredOf divides 2 [h_term ; KTnumber (KTNnat 2)].

Definition cos_le_one : Formula :=
  KFforallVars [variable "h"] (KFlessEqual cos_h (KTnumber (KTNnat 1))).
```

Definition 31 (Hybrid Programs)**.** *Hybrid programs are mutually defined with hybrid formulas with the following grammar. Here α and β are hybrid programs, φ is an hybrid formula, θ is a term and a is a program constant symbol.*

$$\alpha, \beta ::= \quad x := \theta \mid x := * \mid ?\varphi \mid \alpha; \beta \mid \alpha \cup \beta \mid \alpha^* \mid \dot{x} = \theta \,\&\, \varphi \mid a$$

The assignment $x := \theta$ assigns the evaluation of the term θ to the variable x. The undetermined assignment $x := *$ assigns to x an undetermined real value. The test $?\varphi$ passes, meaning it doesn't change the state, if and only if the formula φ holds for the starting state. The composition $\alpha; \beta$ is the sequence of α and β. The non-deterministic choice of α or β is $\alpha \cup \beta$. The Kleene star, α^* is the iteration of α an arbitrary by finite (possibly zero) number of times. The continuous evolution is encoded with ordinary differential equations (ODE), $\dot{x} = \theta \& \varphi$ is the ODE where the state evolves satisfying the equation $\dot{x} = \theta$ and satisfying the domain constraint φ. The evolution can end at any time, and must end if the formula φ is not satisfied anymore. The symbol a is a program constant, that will be evaluated by the interpretation function.

Coq dL Definition of Programs We now present the mechanization of hybrid programs. Most of the constructs directly mirror the syntactical Definition 31. The notable difference is the `KPodeSystem` that we elaborate on further.

```
with Program : Type :=
| KPassign      (x : KAssignable) (e : Term)            : Program      (* x := e or x' := e *)
| KPassignAny   (x : KAssignable)                       : Program      (* x := * or x' := * *)
| KPtest        (cond : Formula)                        : Program      (* ?cond *)
| KPchoice      (left : Program)(right : Program)       : Program      (* alpha u beta *)
| KPcompose     (left : Program)(right : Program)       : Program      (* alpha ; beta *)
| KPloop        (child : Program)                       : Program      (* alpha* *)
| KPodeSystem (ode : ODE) (constraint : Formula) : Program
| KPconstant    (name : ProgramConstName)               : Program
    (* program constant e.g., alpha *)
.
```

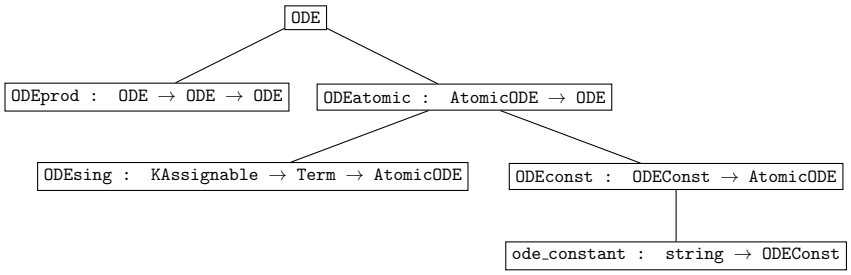


Figure 4.2: A summary of the types used to encode an ordinary differential equation in Coq dL.

Ordinary Differential Equation The definition of ODE requires some inductive types that we summarise on Figure 4.2. The top-level type `ODE` has two constructors `ODEatomic` and `ODEprod`. While `ODEatomic` is used to create an ordinary equation either from a constant `ODEconst` that will be evaluated by the interpretation function or explicitly with `ODEsing`, `ODEprod` takes two `ODE` and create the system with both ODE. Notice that `ODEconst` and `ODEprod` have no equivalent in the mathematical definition of ODE in $\mathbf{d}\mathcal{L}$. Actually, `ODEconst` can be defined using a program constant and `ODEprod` is the equivalent of making the conjunction of two ODE with the $\wedge$ operator.

Example of Program To illustrate the use of Coq dL, we give an example of two hybrid programs $\alpha = (h := *; (?\varphi \cup v := 0))$. and $\beta = (\dot{h} = v \,\&\, \varphi)$.

```
Definition alpha : Program := KPcompose
                              (KPassignAny h)
                              (KPchoice
                                (KPtest phi)
                                (KPassign v (KTnumber (KTNreal 0)))).

Definition beta : Program :=
  KPodeSystem
    (ODEatomic (ODEsing (KAssignDiff h) (KTread v)))
    phi
```

4.1.2 Visual Representation of Hybrid Programs

In this section, we give visual representations of the constructor of hybrid programs in order to help the reader understanding them. For this section $\eta, \nu_1, \ldots, \nu_n$ and $\omega_1, \ldots, \omega_n$ are states, x and y are variables, and φ is an hybrid formula.

Figure 4.3 is an assignment of x. The state ω is the same as ν, except for x which is equal to 2 in ω.

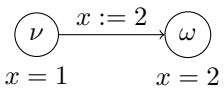


Figure 4.3: Assignment

In the Figure 4.4, we represent an undetermined assignment of x. The reachable states are all states that are equal to ν on all variables others than x, which can hold any real value in ω.

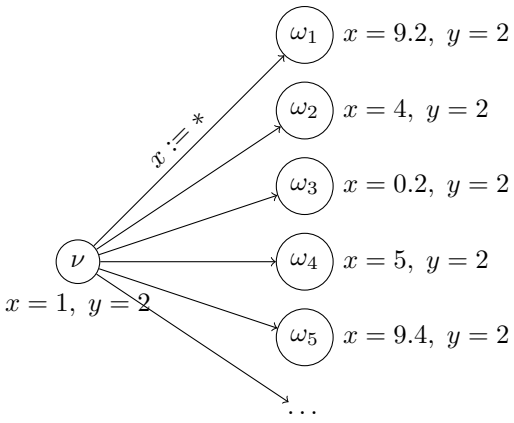


Figure 4.4: Assignment to any value

The only transitions accepted by tests are the ones that are from and to the same state. The state must also to respect the condition formula. In Figure 4.5, the state ν satisfies the formula $x = 1$ so the transition from ν to itself is accepted by the test program $?x = 1$.

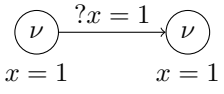
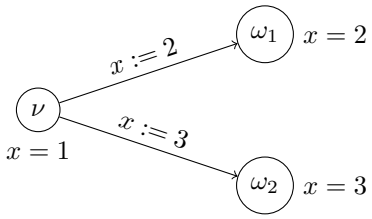


Figure 4.5: Test

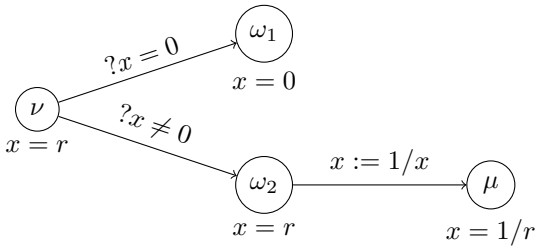
The Figure 4.6 represents the non-deterministic choice of two hybrid programs. The first possibility goes from ν to ω_1, the second goes from ν to ω_2. Thus, the states reachable from ν are ω_1 and ω_2.



$$x := 2 \cup x := 3$$

Figure 4.6: Choice

While we can use this operator to encode the indeterminate behavior of a system. We can also use it with the test constructor to encode an if-then-else behavior as shown on Figure 4.7. In this example, we test to see if the variable x is equal to 0 before dividing by it. The point is to determine the choice operator by putting tests in each branch. By doing so we ensure that the branch from ν to ω_2 will be taken only if $x \neq 0$.



$$?x = 0 \cup (?x \neq 0; x := 1/x)$$

Figure 4.7: If-then-else encoding in Coq dL

We represent the sequential composition in Figure 4.8. The state μ is the intermediate state from ν to ω.

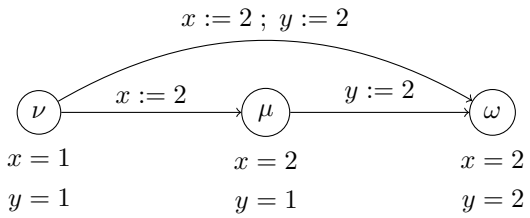
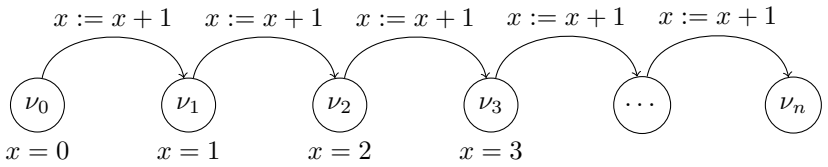


Figure 4.8: Sequential composition

Next, we represent the Kleene star iterator in Figure 4.9. Every state $\nu_0, \nu_1, \ldots, \nu_n$ are reachable.



$$(x := x + 1)^*$$

Figure 4.9: Kleene star iterator

We represent the continuous evolution with another kind of graphic (Figure 4.10). For an ODE: $\dot{x} = \theta_1, \dot{y} = \theta_2 \,\&\, \varphi$. From the state ν the state ω is reachable if there exists a function $f : \mathbb{R} \to \mathcal{S}$ such as (i) $f(0) = \nu$ and $\exists k,\ f(k) = \omega$, (ii) f is a solution of $\dot{x} = \theta_1 \wedge \dot{y} = \theta_2$ and (iii) for all t between 0 and k, $f(t)$ satisfies φ. We say that f is a solution of $\dot{x} = \theta_1 \wedge \dot{y} = \theta_2$ iff $f(t)$. $\forall t, 0 < t < k, f(t)(\dot{x}) = \theta_1 \wedge f(t)(\dot{y}) = \theta_2$.

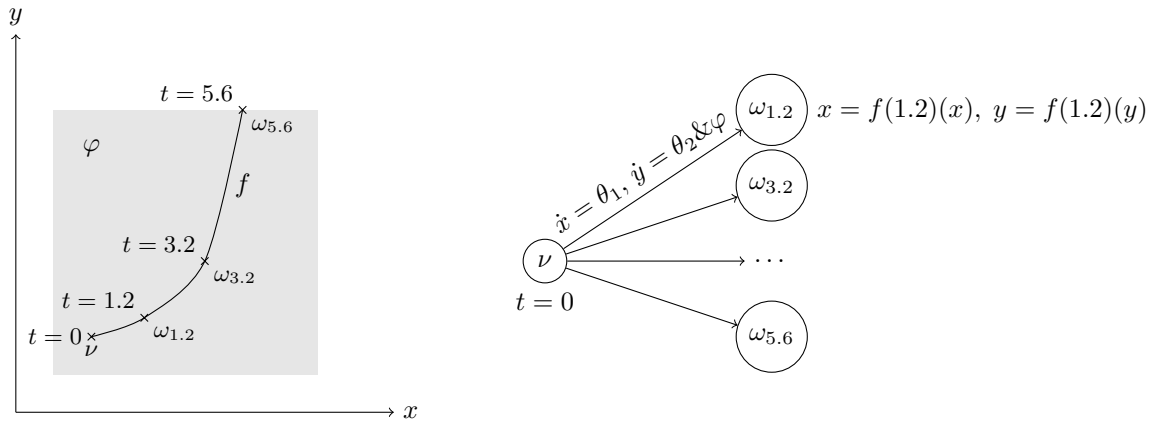


Figure 4.10: Continuous Evolution of an ODE according to its solution f.

4.1.3 Formal Semantics

In this section we introduce the semantics of differential dynamic logic defined for the uniform substitution calculus [Pla17]. We also introduce the implementation done in Coq dL [Boh+17]

Interpretation Function The interpretation I is a function that gives a semantic to function symbols, predicate symbols, quantifier symbols, program constant symbols and ODE symbols. The type of the interpretation function depends on the nature of symbol we interpret. For a function symbol f of arity n, $I(f)$ is a function $\mathbb{R}^n \to \mathbb{R}$. For a predicate symbol p of arity n, the interpretation is a relation $I(p) \subseteq \mathbb{R}^n$. For a quantifier symbol C, the interpretation is a function that maps any set of states $M \subseteq \mathcal{S}$ to a set of states $I(C)(M) \subseteq \mathcal{S}$. For a program constant symbol a, the interpretation is a set of transitions $I(a) \subseteq (\mathcal{S} \times \mathcal{S})$, like the semantic of a program. Coq dL defines the `interpretation` type, the type of the interpretation function. Notice that Coq dL also gives an interpretation for `SymbolODE` that is not in the mathematical definition.

```
Definition interpretation := forall f : Symbol,
  match f return Type with
    | SymbolFunction _ n ⇒ interpretation_function n
    | SymbolDotTerm n ⇒ R
    | SymbolPredicate _ n ⇒ Vector.t R n → Prop
    | SymbolQuantifier _ ⇒ interpretation_quantifier
    | SymbolDotForm ⇒ FormulaSem
    | SymbolODE _ ⇒ interpretation_ode
    | SymbolProgramConst _ ⇒ ProgramSem
end.
```

Semantics in d$\mathcal{L}$ The semantic of a term, formula or program is given by the function $[\![\ldots]\!]$, the function will have a different type according to the nature of the expression. Precisely, the semantic of a term is the evaluation of the term for a state ν as a real $I\nu[\![\theta]\!] \in R$, the semantic of a formula is a set of states satisfying the formula $I[\![\varphi]\!] \subseteq \mathcal{S}$, and the semantic of a program is a set of transitions that are a possible run of the program $I[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$. In Coq dL the semantics of a term, formula or program is given by the functions `dynamic_semantics_term`, `dynamic_semantics_formula` and `dynamic_semantics_program`. The semantics are defined inductively on the constructors of terms, formula and programs. Since both formulas and programs uses terms on their definitions, we first define the semantic of terms, then since programs and formulas uses each other we define their semantics mutually.

Definition 32 (Semantics of Terms)**.** *The semantic of a term θ with interpretation I and ν is its value in $\mathbb{R}$ and is noted $I\nu[\![\theta]\!]$.*

- $I\nu[\![x]\!] = \nu(x)$ *for variable* $x \in \mathcal{V}$
- $I\nu[\![f(\theta_1, \ldots, \theta_n)]\!] = I(f)(I\nu[\![\theta_1]\!]), \ldots, I\nu[\![\theta_n]\!])$
- $I\nu[\![r]\!] = r$
- $I\nu[\![\theta_1 + \theta_2]\!] = I\nu[\![\theta_1]\!] + I\nu[\![\theta_2]\!]$
- $I\nu[\![\theta_1 - \theta_2]\!] = I\nu[\![\theta_1]\!] - I\nu[\![\theta_2]\!]$

- $I\nu[\![\theta_1 \times \theta_2]\!] = I\nu[\![\theta_1]\!] \times I\nu[\![\theta_2]\!]$
- $I\nu[\![-\theta]\!] = -I\nu[\![\theta]\!]$
- $I\nu[\![\dot{\theta}]\!] = \sum_{x \in \mathcal{V}} \nu(\dot{x}) \frac{\partial I\nu[\![\theta]\!]}{\partial x}$

To evaluate the semantic of a term in Coq dL we have `dynamic_semantics_term`. Below, we only show an extract of the full definition. For the `KTfuncOf`, we see the use of `interp_fun_f` and `I` to evaluate the symbol `f`.

```
Fixpoint dynamic_semantics_term (I : interpretation) (s : state) (t : Term)
    {struct t} : R :=
    match t with
    | KTnumber r ⇒ KTnum2R r
    | KTread x ⇒ s x
    | KTplus l r ⇒
      dynamic_semantics_term I s l + dynamic_semantics_term I s r
    | KTneg l ⇒ − dynamic_semantics_term I s l
    | KTfuncOf f n args ⇒ interp_fun_f _ I (SymbolFunction f _))
        (Vector.map (dynamic_semantics_term I s) args)
    | KTminus l r ⇒ ...
    | KTtimes l r ⇒ ...
    | KTdifferential theta ⇒ ...
    ...
    end.
```

Definition 33 (Semantics of Formulas)**.** *We define the semantic of a formula inductively mutually with the semantics of programs.*

- $I[\![\top]\!] = \mathcal{S}$
- $I[\![\bot]\!] = \emptyset$
- $I[\![\theta_1 \sim \theta_2]\!] = \{\nu \in \mathcal{S} \mid I\nu[\![\theta_1]\!] \sim I\nu[\![\theta_2]\!]\}$
- $I[\![\neg\varphi]\!] = \overline{I[\![\varphi]\!]}$
- $I[\![\varphi \wedge \psi]\!] = I[\![\varphi]\!] \cap I[\![\psi]\!]$
- $I[\![\varphi \vee \psi]\!] = I[\![\varphi]\!] \cup I[\![\psi]\!]$
- $I[\![\varphi \to \psi]\!] = \overline{I[\![\varphi]\!]} \cup I[\![\psi]\!]$
- $I[\![\varphi \leftrightarrow \psi]\!] = (I[\![\varphi]\!] \cap I[\![\psi]\!]) \cup (\overline{I[\![\varphi]\!]} \cap \overline{I[\![\psi]\!]})$
- $I[\![\forall x,\ \varphi]\!] = \{\nu \in \mathcal{S} \mid \forall r \in \mathbb{R},\ \nu_x^r \in I[\![\varphi]\!]\}$
- $I[\![\exists x,\ \varphi]\!] = \{\nu \in \mathcal{S} \mid \exists r \in \mathbb{R},\ \nu_x^r \in I[\![\varphi]\!]\}$
- $I[\![p(\theta_1, \ldots, \theta_n)]\!] = \{\nu \in \mathcal{S} \mid (I\nu[\![\theta_1]\!], \ldots, I\nu[\![\theta_n]\!]) \in I(p)\}$
- $I[\![C(\varphi)]\!] = I(C)(I[\![\varphi]\!])$

- $I[\![[\alpha]\varphi]\!] = \{\nu \in \mathcal{S} \mid \forall \omega \in \mathcal{S},\ (\nu, \omega) \in I[\![\alpha]\!] \to \omega \in \varphi\}$
- $I[\![\langle\alpha\rangle\varphi]\!] = \{\nu \in \mathcal{S} \mid \exists \omega \in \mathcal{S},\ (\nu, \omega) \in I[\![\alpha]\!] \wedge \omega \in I[\![\varphi]\!]\}$

To get the semantics of hybrid formula, we use `dynamic_semantics_formula`. As it uses the semantics of hybrid programs, it is mutually defined with `dynamic_semantics_program` which we will introduce later. We only show an extract of the definition to improve readability.

```
Fixpoint dynamic_semantics_formula (I : interpretation) (fi : Formula)
    : FormulaSem :=
match fi with
| KFtrue ⇒ fun _ : state ⇒ True
| KFfalse ⇒ fun _ : state ⇒ False
| KFgreaterEqual l r ⇒
  fun S ⇒ Rge (dynamic_semantics_term I S l) (dynamic_semantics_term I S r)
| KFgreater l r ⇒ ...
| KFlessEqual l r ⇒ ...
...
| KFpredOf f n args ⇒
  fun S ⇒ I (SymbolPredicate f n) (Vector.map (dynamic_semantics_term I S) args)
| KFquantifier f a ⇒ ...

| KFnot l ⇒
    fun S ⇒ not (dynamic_semantics_formula I l S)
| KFand l r ⇒ ...
...

| KFforallVars vars F ⇒
   fun S ⇒ forall rs, List.length rs = List.length vars
       → dynamic_semantics_formula I F (upd_list_state S (combine vars rs))
| KFexistsVars vars F ⇒ ...

| KFdiamond alpha F ⇒
    fun S ⇒
     forall w,
       dynamic_semantics_program I alpha S w
       → dynamic_semantics_formula I F w
| KFbox alpha F ⇒ ...

end.
```

Definition 34 (Semantics of Programs)**.** *We introduce the semantics of a program $I[\![\alpha]\!]$ as a set of state pairs (transitions).*

- $I[\![a]\!] = I(a)$
- $I[\![x := \theta]\!] = \{(\nu, \omega) \in \mathcal{S} \times \mathcal{S} \mid \omega(x) = I\nu[\![\theta]\!] \ \wedge\ \forall y \in \mathcal{V} \setminus \{x\},\ \nu(y) = \omega(y)\}$
- $I[\![x := *]\!] = \{(\nu, \omega) \in \mathcal{S} \times \mathcal{S} \mid \forall y \in \mathcal{V} \setminus \{x\},\ \nu(y) = \omega(y)\}$

- $I[\![?\varphi]\!] = \{(\nu, \nu) \in \mathcal{S} \times \mathcal{S} \mid \nu \in I[\![\varphi]\!]\}$
- $I[\![\alpha; \beta]\!] = \{(\nu, \omega) \in \mathcal{S} \times \mathcal{S} \mid \exists \mu \in \mathcal{S},\ (\nu, \mu) \in I[\![\alpha]\!] \wedge (\mu, \omega) \in I[\![\beta]\!]\}$
- $I[\![\alpha \cup \beta]\!] = I[\![\alpha]\!] \cup I[\![\beta]\!]$
- $I[\![\alpha^*]\!] = \bigcup_{n \in \mathbb{N}} I[\![\alpha^n]\!]$ *with* $\alpha^{n+1} \equiv \alpha^n; \alpha$ *and* $\alpha^0 \equiv ?\top$
- $$\begin{aligned} &I[\![\dot{x} = \theta \,\&\, \varphi]\!] = \{(\nu, \omega) \in \mathcal{S} \times \mathcal{S} \mid \exists r \in \mathbb{R},\ \exists f : [0, r] \to \mathcal{S}, \\ &(\forall y \in \mathcal{V} \setminus \{\dot{x}\},\ \nu(y) = f(0)(y)) \wedge && \textit{initially } f \textit{ is equal to } \nu \textit{ except on } \dot{x} \\ &(\forall y \in \mathcal{V},\ \omega(y) = f(r)(y)) \wedge && f \textit{ equals } \omega \textit{ at time } r \\ &(\forall \epsilon \in [0, r],\ f(\epsilon)(\dot{x}) = \frac{df(t)(x)}{dt}(\epsilon) \wedge f(\epsilon) \in I[\![\dot{x} = \theta \wedge \varphi]\!])\} && f \textit{ is solution of the ODE} \\ & && \textit{and } f(\epsilon) \textit{ satisfies } \varphi \end{aligned}$$

Finally, we have the Coq function `dynamic_semantics_program` to evaluate hybrid programs. We only show a portion of the definition for the sake of readability. The function `differ_state_except v w a r`, used for `KPassign` and `KPassignAny`, holds if the state `w` is equal to the state `v` on all variable except on `a`, on which it must be equal to `r`.

```
with dynamic_semantics_program (I : interpretation) (p : Program)
    : ProgramSem :=
      match p with
      | KPconstant a ⇒ I (SymbolProgramConst a)

      | KPassign a theta ⇒
        fun v w ⇒ differ_state_except v w a (dynamic_semantics_term I v theta)

      | KPassignAny a ⇒ fun v w ⇒ exists r, differ_state_except v w a r

      | KPtest fi ⇒ fun v w ⇒ v = w ∧ dynamic_semantics_formula I fi v
      | KPchoice alpha beta ⇒ ...

      | KPcompose alpha beta ⇒
        fun v w ⇒
          exists s,
            dynamic_semantics_program I alpha v s
            ∧ dynamic_semantics_program I beta s w

      | KPloop p ⇒ ...
      | KPodeSystem ode psi ⇒ ...
      end.
```

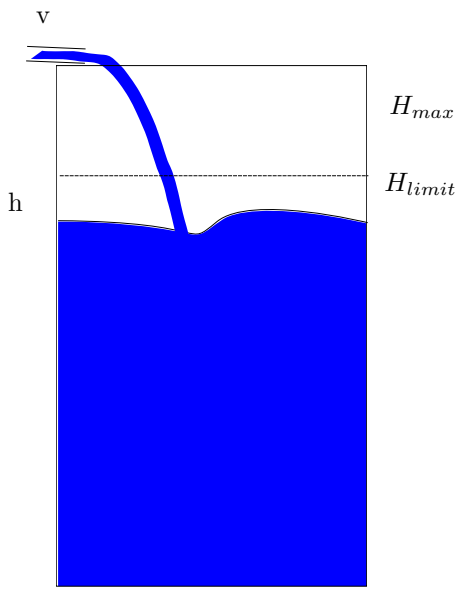


Figure 4.11: A water tank controlled by a valve.

4.2 The Water Tank example

4.2.1 The Model

We introduce a simple model to be used as a running example. The model consists of a water tank whose inflow is controlled by a valve as depicted in Figure 4.11. The water tank and valve is a classic example for this kind of demonstration, used in many papers such as [CJR96]. Its organisation as components and the interaction of continuous evolution and discrete computation makes it a good candidate to express all the features of our formalism. The goal is to formalize the specification and the behavior of the two components, and to show that this formalization is sound. Namely, that this behavior correctly implements the specification.

The valve can detect when the water level is above a certain limit and modulate its flow accordingly. The variable h describes the water level in the tank, $\dot{h}$ is the differential of h, and v is the flow rate of water in the tank. The parameter H_{max} is the level that the water must not exceed, H_{limit} is the level at which we have chosen to close the inflow.

Expected Behavior We describe the valve and tank as two components of the system using hybrid programs. The valve behavior is described by the hybrid program α in Equation 4.1. In this program, first, the value of h is assigned to any value. This is made to represent the change of the water level. Indeed, since the valve doesn't have control over h, we suppose it can take any value.

Then the valve has two options. If $h \geq H_{limit}$, the inflow must be stopped. This behavior is guaranteed, because when $?h < H_{limit}$ is false, the hybrid program can be reduced to $v := 0$. Otherwise, the valve can remain in the previous state or can be stopped before the water level reaches to H_{limit}. The behavior is depicted in Figure 4.12

$$\alpha = (h := *; (?h < H_{limit} \cup v := 0)) \tag{4.1}$$

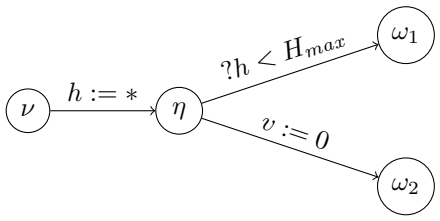


Figure 4.12: The behavior of the valve as an hybrid program.

The water tank is described by the hybrid program β in Equation 4.2 and in Figure 4.13. The water tank will progressively fill. Because the tank has no control over v, it is assigned to a undetermined value. Then, the water level will increase by the inflow rate v. There is no domain constraint, which means the continuous evolution ruled by the differential equation can stop at any time. We give an example of run in Figure 4.14.

$$\beta = (v := *; (\dot{h} = v \,\&\, \top)) \tag{4.2}$$

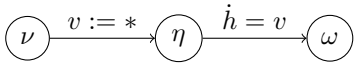


Figure 4.13: The behavior of the water level as an hybrid program.

4.2.2 Specification

Informally, we want the water tank to never overflow, regardless of how long the system is running. This would be the guarantee of the behavior of the system. As for the assumption, it is necessary to assume that the water tank is not overflown initially, as we cannot guarantee anything before the execution of the system. It is also necessary that the valve closes the inflow when the water level is above a certain level.

Our goal is to formally express this specification and verify that our model implements this specification. Namely, that the behavior of the hybrid program modeling the system satisfies the specification.

Here we give a summary of the informal contracts for the water tank:

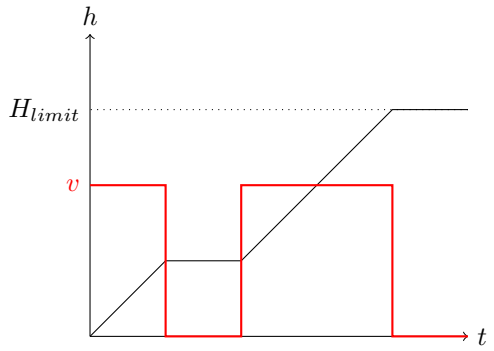


Figure 4.14: Example of a behavior of h and v

- Assume the water level start below H_{max}.
- Assume the valve closes if the water level is above H_{limit}.
- Guarantee the water level will always be below H_{max}.

And for the valve:

- Guarantee the valve closes if the water level is above H_{limit}.

In the following sections, we explore the modelling and formal specification of the system. It is important to note that we won't show that the hybrid programs are the correct model of a real implementation of the water tank. We merely show that the model is sound and implement its specification, not that it is a realistic model.

4.3 Instantiating Hybrid Programs

In this section, we give the first instantiation of the verified contract theory. We first explain what a behavior is, in the context of differential dynamic logic, and how this is used to define the base types needed for the instantiation. Then we give the necessary information for the instantiation of the type class `NonEmptyDecidableVariable` defined in Section 3.3. Finally, we use the instantiation to define components for the water tanks, our running example.

4.3.1 Base Types

To instantiate the theory the first step is to define what the type of a behavior is. Concretely this means choosing the types `value` and `ident`. We first introduce how states and transitions are defined in Coq dL, then we explain our choice for the types `value` and `ident`.

States in Coq dL In Coq dL, the state[1] of a component is represented by the valuation of every variable. Notice that variables are valued in `R`, which is the set of mathematical reals $\mathbb{R}$, and they are identified with `KAssignable`, on which we give more explanation below.

```
Definition KState := KAssignable → R.
```

`KAssignable` as Identifiers In Coq dL, to define a new variable we use the constructor `KVariable`, which is defined using a string as a parameter of construction. These are the non-differential variables. Yet, in differential dynamic logic, a variable x and its differential $\dot{x}$ are both considered variables. Thus we use `KAssignable` to type the variables, which are `KVariable` and their differentials.

```
Inductive KVariable : Set := variable : string → KVariable
Inductive KAssignable : Set :=
  | KAssignVar : KVariable → KAssignable
  | KAssignDiff : KAssignable → KAssignable

 Definition ident : Type := KAssignable.
```

Encoding Behaviors with Transitions A *transition* is a pair of an origin and a target state. In Coq dL, the semantic of an hybrid program is a set of transitions. We want to instantiate the types `ident` and `value`, to have the behavior defined in the verified contract theory being as close as possible as the semantic of an hybrid program. At a first glance, we would hence expect a behavior to be a transition like below.

```
Definition transition := (KState * KState).
Definition behavior := transition.
```

This definition is not compatible with the definition of behavior from the assumption/guarantee theory of contract:

```
Definition behavior (d : alphabet) := {v : ident | v ∈ d} → value.
```

There is no satisfactory implementation of `ident` and `value` that transcribes this formula into a transition. Thus, we define behaviors which are defined with the formula and can be transformed into transitions. The solution is to define behavior as evaluation of each variable in the alphabet of the component to a couple of initial and final value. We give an example of a behavior coding a transition in Figure 4.15.

```
Definition value : Type := (R * R).
```

This is enough to have most definition in our theory, like alphabets, contracts and components. For example we can define an alphabet `d`, made of four `KAssignable`. And create a component as a set of behaviors, although without any new constructors it will be cumbersome to define interesting components.

[1] In the Coq dL library it is defined as `states` but we rename it to `KState` to remove ambiguity

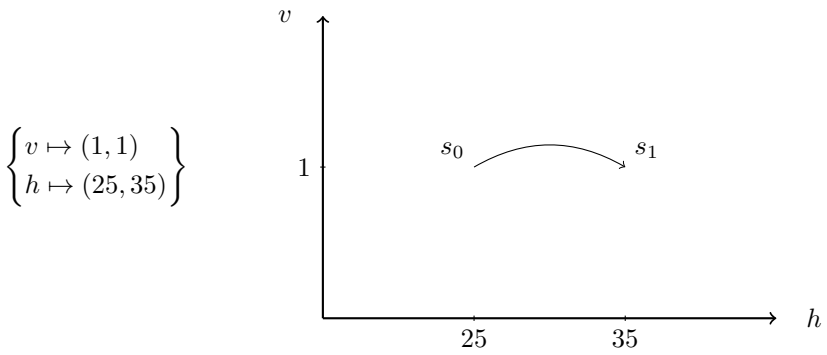


Figure 4.15: A behavior (left) coding a transition (right)

```
Definition d : @alphabet KAssignable :=
    add v ( add h'' (add h' (add h (@emptyset KAssignable)))).

Definition behavior_example : behavior d := fun x : var d ⇒ (0, 0).
Definition component_example : component d :=
    fun s : behavior d ⇒ s = behavior_example.
```

4.3.2 Instantiating the Type Class

To have the complete facts proved for our verified contract theory, we now need to instantiate the `NonEmptyDecidableVariable` type class. The theory of contracts is almost ready to be instantiated, we miss two things:

- A default value needed for the extension of assertion.
- The decidability principle of membership for `KAssignable` sets (alphabet).

As default value, we chose (0, 0) for no particular reasons, it could have been any other `R * R` value.

The "in" decidability is admitted for now, as it is very difficult to prove with our definition of sets. Indeed, since the set are abstract, this is nearly equivalent to general decidability which we don't accept. We hope to be able to remove this axiom by changing the definition of set in the future.

```
Axiom in_dec_KAssignable : ∀ (x : KAssignable) (d : alphabet), {x ∈ d} + { x ∉ d}.
```

This finishes the instantiation of `NonEmptyDecidableVariable`. We use the keyword `Instance` to instantiate the class `NonEmptyDecidableVariable`. We name the the instance we create `KAssignable_Reals` to highlight the types of the identifiers and values.

```
Instance KAssignable_Reals : NonEmptyDecidableVariable (R * R) KAssignable := {
  any_value := (0 , 0)%R ;
  in_dec_ident := in_dec_KAssignable ;
}.
```

4.3.3 Transforming Programs into Components

Since we don't use transitions in the instantiation, the semantics of hybrid programs and components differ. Indeed, the semantic of an hybrid program is a set of transition `Kstate → Kstate → Prop` whereas the semantic of a component is a set of behavior `behavior → Prop`. We show how to transform an hybrid program into a component.

Evaluate Behavior The first building block of this transformation is to evaluate behaviors on `KAssignable`. Since behavior are defined on a alphabet `d`, a restriction of `KAssignable`, they don't hold values for variables outside of `d`. Here, we chose to resolve the problem by returning a default value `any_value`, defined earlier. The function `exist : forall x : A, P x → {x : A | P x}`, creates the variables with the proof they are in the alphabet. And the type `behavior` unfolds to `{v : KAssignable | d v} → value`. So `t (exist _ x x_in_d)` is the evaluation of the behavior `t` on variable `x`.

```
Definition eval {d : alphabet} (t : behavior d) (x : KAssignable) : value :=
  match in_dec_ident x d with
                  | left x_in_d ⇒ t (exist _ x x_in_d)
                  | right _ ⇒ any_value
  end.
```

Behavior Conversion Then we can define `to_transition`, which transforms a behavior to a transition, and `to_component` which transforms a program to a component. Though, we don't want to transform a component with variables outside of the alphabet `d`. Thus we ask for the predicate `prog_in_alphabet d` to hold for the program, we explain below the definition of `prog_in_alphabet`.

```
Definition to_transition {d : alphabet} (t : behavior d) : KState * KState:=
  (fun (x : KAssignable) ⇒ fst (@eval d t x),
  fun (x : KAssignable) ⇒ snd (@eval d t x)).
```

Program Conversion To define the `to_component` function, we use the function `dynamic_semantics_program` which holds if a transition satisfies a program. The function `dynamic_semantics_program` also uses an interpretation function for constant symbols, we abstract the interpretation by admitting we have an unspecified interpretation function `I`.

```
Definition to_component {d : alphabet} (I : interpretation) (p : Program)
    (p_in_alphabet: prog_in_alphabet d p) : component d :=
    fun (t : behavior d) ⇒ let (prestate, poststate) := @to_transition d t in
        dynamic_semantics_program I p prestate poststate.
```

Program's Alphabet Since hybrid programs are defined on every variable, and not a fixed alphabet, we want to create a predicate, which can be understood as "This hybrid program is defined on this alphabet". In Coq dL, the function

`all_vars_program` returns all the free and bound variables appearing in an hybrid program. Though, the set it returns is either finite or cofinite (namely, its complement is finite), and is defined with type `FCset`, which we can not use. With `to_alphabet` we transform a `FCset` to a set of `KAssignable`. Finally, with `prog_in_alphabet`, we can check that a program is defined on a alphabet, by checking that all variables used in it are in the alphabet `d`.

```
Definition to_alphabet (f : @FCset KAssignable) : (@alphabet KAssignable):=
  match f with
  | FCS_finite l ⇒ fun (x : KAssignable) ⇒ List.In x l
  | FCS_infinite l ⇒ fun (x : KAssignable) ⇒ ¬List.In x l
  end.


Definition prog_in_alphabet (d : alphabet) (p : Program) :=
  to_alphabet (all_vars_program p) ⊆ d.
```

4.3.4 Example with the Water Tank

We can define components for the hybrid programs of our example α and β. First we must create the alphabet of our system. Since the three variables used in the programs are h, $\dot{h}$ and v. The alphabet is defined as follows :

```
Definition h : KAssignable := variable "h".
Definition h' : KAssignable := KAssignDiff h.
Definition h'' : KAssignable := KAssignDiff h'.
Definition v : KAssignable := variable "v".
Definition d : alphabet := add v ( add h'' (add h' (add h (@emptyset KAssignable)))).
```

Then, we define the components α and β according to their differential dynamic logic definition. This is the same programs as in Section 4.2, except we use the Coq dL notation introduced in Section 4.1.

```
Definition alpha : Program :=
    KPcompose
        (KPassignAny h)
        (KPchoice
            (KPtest (KFless (KTread h) Hlimit))
            (KPassign v (KTnumber (KTNreal 0)))).

Definition beta : Program :=
    KPcompose
        (KPassignAny v)
        (KPodeSystem
            (ODEatomic (ODEsing (KAssignDiff h) (KTread v)))
            KFtrue
  ).
```

To define the components for α and β, we use the `to_component` function, but we first need to verify that all variables of α and β are in the alphabet `d`.

```
Lemma alpha_in_dom : prog_in_alphabet d alpha.
Lemma beta_in_dom : prog_in_alphabet d beta.

Definition alpha_component := to_component I alpha alpha_in_dom.
Definition beta_component := to_component I beta beta_in_dom.
```

4.3.5 Limitations of the Instance

Contracts as Hybrid Programs In the verified contract theory, components and contracts are very similar: they are defined with assertions. In a way, contracts are made of two components, one expressing the assumption and one expressing the guarantee. We could follow this idea and define contracts as a pair of hybrid programs. But this is not satisfying: it implies that the implementation relation between components and contracts would not be expressible as an hybrid formula. Indeed, the implementation relation is defined as follows :

$$\alpha \preceq \neg A \vee G \tag{4.3}$$

With A and G being the assumption and guarantee of the contract. Yet in $\mathbf{d}\mathcal{L}$, there is no definition of the negation of a hybrid program. This means we can't represent this relation in $\mathbf{d}\mathcal{L}$. Hence we won't be able to use the proof assistant Keymaera X to verify the implementation of a contract by a component.

Abstract Programs for Contract Our idea is to define a subset of hybrid programs, which is as close as possible as hybrid program and close by negation. The subset must also be closed by disjunction and conjunction. It would be translatable into hybrid programs, since we need to write them in Keymaera X. We call such a subset that of *Abstract Programs* and give its definition in the Section 4.5.

Differential Refinement In order to write Equation 4.3 in differential dynamic logic, we also need a way to write refinement in $\mathbf{d}\mathcal{L}$. Luckily there is a refinement relation defined for $\mathbf{d}\mathcal{L}$ called differential refinement. In Section 4.4 we show that the refinement from the assumption/guarantee of contracts can be expressed with differential refinement.

4.4 Refinement in Differential Dynamic Logic

In Chapter 3, we introduced the refinement relation defined in the assumption/guarantee contract theory and the meta-theory [Ben+15b]. Yet, another refinement relation has been defined for differential dynamic logic [LP16]. This definition of refinement is directly defined in the logic and can be used in the proof calculus to facilitate reasoning. It seems natural to investigate the link between the two refinement definitions. In this section, we show that the refinement of hybrid programs from differential dynamic logic implies the refinement of component from the verified contract theory.

4.4.1 Definitions of the Refinement Relations

Refinement from A/G Contract Theory A refinement relation is defined by instance of the contract theory. We gave the general definition of refinement in Chapter 3. The instance of Section 4.3 specializes this definition for hybrid programs which we give in Equation 4.4. We assume an interpretation function I is defined.

$$a \preceq b \quad \equiv \quad I[\![a]\!] \subseteq I[\![b]\!] \tag{4.4}$$

Differential refinement A refinement relation is also defined in the reference paper on differential refinement [LP16], see Equation 4.5 We use *differential refinement*($\preceq_{dL}$) to denote this relation, in order to avoid confusing it with *refinement* ($\preceq$).

$$a \preceq_{dL} b \quad \equiv \quad \vdash \forall \boldsymbol{x'},\ \langle a \rangle\ \boldsymbol{x} = \boldsymbol{x'}\ \rightarrow\ \langle b \rangle\ \boldsymbol{x} = \boldsymbol{x'} \tag{4.5}$$

In this formula, $\boldsymbol{x}$ is a vector containing at least every variables of a and b. $\boldsymbol{x'}$ is a vector of fresh variables with the same size as $\boldsymbol{x}$. The operator is $\boldsymbol{x} = \boldsymbol{x'}$ is the equality of vectors, in other words: $x_1 = x'_1 \ \wedge\ \ldots\ \wedge\ x_n = x'_n$ with $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{x'} = (x'_1, \ldots, x'_n)$.

This formula can be understood as follows: for any run starting from any initial state, if there exists a transition in a that ends with the values $\boldsymbol{x'}$, then one run that also ends with the same values $\boldsymbol{x'}$ must exists in b. Indeed, since $\boldsymbol{x'}$ is a vector of *fresh* variables, its values are not modified in a nor b. And since $\boldsymbol{x}$ contains every bound variables of a and b, knowing the values of every variable on $\boldsymbol{x}$ is sufficient to conclude that both states at the end of a and b are equivalent.

Alphabet In the following we assume a set d of every variable read (free variables) and written (bound variable) in either program a or b. We assume the set to be finite, indeed it seems reasonable to accept that a real component have a finite number of variables. We define $\boldsymbol{d}$ the vector of variables containing once every variable in d. We also assume a set d' of fresh variables with the same size as d. We define $\boldsymbol{d'}$ the vector containing once every variable in d'.

4.4.2 Definition of Differential Refinement in Coq

Before proving that the differential refinement implies the refinement, we first need to define the differential refinement in Coq. This definition of this formula in Coq poses some problems: We need to define the alphabet as a finite set and create a list containing every variable in the alphabet. Then, we need to introduce a fresh variable $\boldsymbol{d}'_i$ for each variable in the alphabet $\boldsymbol{d}_i$.

Finite Alphabet Having a finite alphabet seems to be a reasonable assumption since real-world components cannot manipulate an infinite number of variables. Because we miss a proper definition of finite set, we declare a `Finite` property

that denote a set as finite, an `elements_of` function, which returns every element of a finite set in a list, and `Enum`, the correction lemma of `element_of`. These are introduced as axioms.

```
Section Finite.
  Axiom Finite : ∀ {Γ  : Type} (s: set Γ ), Prop.
  Axiom  elements_of : ∀ {Γ  : Type} (s : set Γ ) (fs : Finite s),
    list Γ .
  Axiom Enum : ∀ (Γ  : Type) (s :  set Γ ) (fs : Finite s) (x :  Γ ),
    x ∈ s ↔ List.In x (elements_of s fs).
End Finite.
```

We assume the property `Finite` to hold for the alphabet `d`. And create the list `dl` which contains once every variable of `d`.

```
Variable d_finite : Finite d.

Definition dl := elements_of d d_finite.
```

Creation of Fresh Variables Coq dL includes a function `fresh_kassignable`, which given a list of variables, returns a new fresh variable and a proof that the fresh variable is different from every variable in the list. The notation {`x : T $ P x`} for the constructor `sigT` which is the sigma type of an element `x` with the property `P x`.

```
Definition fresh_kassignable :
    ∀ l :  list KAssignable, {x : KAssignable $ ¬List.In x l} := ...
```

The function `fresh_kassignable_list n l` returns a list of `n KAssignable`, which are all different from every `KAssignable` and from each other. It works by calling `freqh_kassignable` on `l` to have a fresh variable `x` then recurring and adding `x` to the list of already used variables.

```
Fixpoint fresh_kassignable_list (n : nat) (l : list KAssignable) : list KAssignable :=
  match n with
  | 0 ⇒ nil
  | S m ⇒ let (x, _):= (fresh_kassignable l) in
           x ::  (fresh_kassignable_list m ( x :: l))
  end.
```

Finally, we use this function to introduce `dl'` a list with the same size as `dl` filled with fresh variables.

```
Definition dl' : list KAssignable :=
   fresh_kassignable_list (length dl) dl.
```

Non-differential Variables In order to use the operator `KFforallvars` to quantify over the fresh variabels `dl'`, we need a list of fresh `KVariable`. Yet, the list `dl'` is of type `list KAssignable`. As we introduce in Section 4.1, `KVariable` is the type of non-differential variables $(x, y, \dots)$ and `KAssignable` is the type of variables, differential or not $(x, y, \dot{x}, \ddot{x}, \dots)$. We use Coq dL `KAssignable2variable`

function that returns the variable non-differential variable from any variable. For example, `KAssignable2variable (KAssignDiff x)` = `x` if `x` is non-differential. And `KAssignable2variable (KAssignVar y)` = `y` if `y` is non-differential. We define `dl'_Variable` a list of the `KVariable` with every variable in `dl`.

```
Definition dl'_Variable : list KVariable :=
  map KAssignable2variable dl'.
```

Equality of Variables to Fresh Variables In the definition of differential refinement, we need to express that the vector $\boldsymbol{d}$ and $\boldsymbol{d}'$ are equal as an hybrid formula. To do so we define `KFeq_list l l'` an hybrid formula which express that the variables in `l` have the same value as the variables `l'`.

```
Fixpoint KFeq_list (la : list KAssignable) ( lb : list KAssignable) : Formula :=
  match (la, lb) with
  | (nil, _:: _) ⇒ KFfalse
  | (_:: _, nil) ⇒ KFfalse
  | (nil, nil) ⇒ KFtrue
  | (h1:: t1, h2:: t2) ⇒ KFand (KFequal h1 h2) (KFeq_list t1 t2)
  end.
```

Mechanization of Differential Refinement With all this auxiliary definitions we can define the differential refinement (Equation 4.5) in Coq. We use `dl` as the alphabet list (equivalent of $\boldsymbol{d}$), and `dl'` the fresh variables (equivalent of $\boldsymbol{d}'$).

```
Definition KFrefine(a b : Program) : Formula :=
  KFforallVars dl'_Variable
               (KFimply (KFdiamond a (KFeq_list dl dl'))
                        (KFdiamond b (KFeq_list dl dl'))).
```

4.4.3 Preliminary Lemmas

We need three lemmas before proving that the differential refinement implies the refinement relation. We will introduce the three lemmas in mathematical terms as well as their Coq mechanization. In the following, we will use s, f to name states, and a, b to name programs. This is different for the Section 4.1 where we used ν, ω, α and β, indeed our goal is to have a uniform notation between the mathematical demonstration and the mechanization. We also introduce a notation $f(\boldsymbol{x}) = (f(\boldsymbol{x_1}), \ldots, f(\boldsymbol{x_n}))$ for $\boldsymbol{x} = (x_1, \ldots, x_n)$, notably $f(\boldsymbol{x}) = f(\boldsymbol{y})$ expresses that $f(x_1) = f(y_1) \wedge \ldots \wedge f(x_n) = f(y_n)$. For the sake of readability we also introduce the notation $s \xrightarrow{a} f \equiv (s, f) \in I[\![a]\!]$. This notation removes the interpretation function I, but this has no consequences on our reasoning.

Differential Refinement The differential refinement states that if we have a transition $s \xrightarrow{a} f$ and $f(\boldsymbol{d}) = f(\boldsymbol{d}')$, then there is a state f' such that $(s, f') \in I[\![b]\!]$

and $f'(\boldsymbol{d}) = f'(\boldsymbol{d'})$. It is a direct consequence of differential refinement, but is more usable in our proof. This is summed up as Lemma 2 and Figure 4.16.

Lemma 2 (Differential Refinement)**.** *With $a \preceq_{dL} b$ and s, f two states,*

$$s \xrightarrow{a} f \ \wedge \ f(\boldsymbol{d}) = f(\boldsymbol{d'}) \implies \exists f', \ s \xrightarrow{a} f' \ \wedge \ f'(\boldsymbol{d}) = f'(\boldsymbol{d'})$$

```
Lemma KFrefine_rewrite (a b : Program) (a_in_d : prog_in_d a) (b_in_d : prog_in_d b) :
  ∀ s f : KState,
  dynamic_semantics_formula I (KFrefine a b) s →
  dynamic_semantics_program I a s f →
  map f (elements_of d fd) = map f (fresh_alphabet_a) →
  exists f, dynamic_semantics_program I b s f ∧
    map f (elements_of d fd) = map f (fresh_alphabet_a).
```

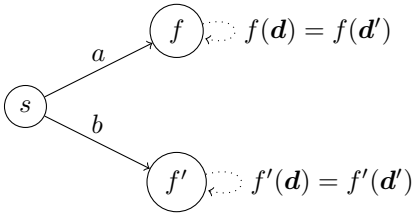


Figure 4.16: Differential Refinement

Bound Effect Lemma The second lemma we need is the bound effect lemma on alphabet. It expresses that a program does not affect variables that are not in its alphabet. In differential dynamic logic, by default, an hybrid program can modify every variable, yet it only modify a finite set of variables. We call bound variables, every variable modified by a program. The bound effect lemma [Pla17, lemma 9] states that every variable that is not a bound variable is equal in the initial and the final state. Since in our construction of components, a component can only modify the variables in the alphabet, every variable that is not in the alphabet is not a bound variable. We deduce the bound effect lemma on alphabet: Lemma 3 and give Figure 4.17 to illustrate.

Lemma 3 (Bound Effect on Alphabet)**.** *If $s \xrightarrow{a} f$,*

$$\forall x \notin d, \ s(x) = f(x)$$

```
Lemma bound_effect_on_d (a : Program) (a_in_d : prog_in_d a)
  (s f : KState) (x : KAssignable) :
  ¬(x ∈ d) → dynamic_semantics_program I a s f → s x = f x.
```

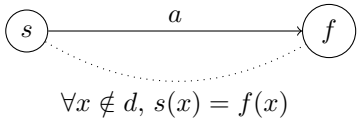


Figure 4.17: Bound effect lemma

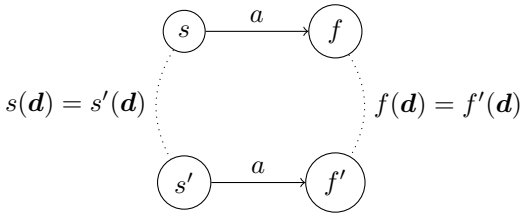


Figure 4.18: Coincidence lemma on alphabet d

Coincidence Lemma The last lemma we need is the coincidence lemma on alphabet. The coincidence lemma used for the uniform substitution calculus [Pla17, lemma 12], expresses that if a transition (s, f) satisfies a, then for every state s' that is equal to the initial state s on at least the free variables, there exists a state f' such that (s', f') satisfies a and f' is equal to f on those variables. A free variable of a program is a variable that change the evaluation of an the program. In our construction only the variables in the alphabet d can change the behavior of a program. Which means that every free variable is in the alphabet, thus we can deduce the coincidence lemma on alphabet. The lemma4 is summarised on Figure 4.18

Lemma 4 (Coincidence Lemma on alphabet)**.** *For states s, f and s' such as $s \xrightarrow{a} f$ and $s'(\boldsymbol{d}) = s(\boldsymbol{d})$ then*

$$\exists f',\ s' \xrightarrow{a} f' \ \wedge\ f'(\boldsymbol{d}) = f(\boldsymbol{d})$$

```
Theorem coincidence_program_on_d (P : Program) (p_in_d : prog_in_d P) (s s' f: KState) :
  (map s dl = map s' dl) →
  dynamic_semantics_program I P s f →
  exists f',  dynamic_semantics_program I P s' f' ∧
  map f dl = map f' dl.
```

4.4.4 Proof that Differential Refinement Implies Refinement

We now prove that the refinement defined from the refinement calculus implies the refinement from the contract theory. We give the mathematical proof, and the mechanized proof is available in the file `DifferentialContracts.v` of the library.

```
Theorem refine_KFrefine
  (a b : Program) (a_in_d : prog_in_d a) (b_in_d : prog_in_d b) :
  (∀ preS : KState, dynamic_semantics_formula I (KFrefine a b) preS) →
  to_component I a a_in_d ⊆ to_component I b b_in_d.
```

Theorem 8 (Differential Refinement Implies Refinement)**.** *For a and b two hybrid programs with variables in d, if $a \preceq_{dL} b$ then $a \preceq b$.*

Proof. We consider two programs a and b such as $a \preceq_{dL} b$. We assume $d \subseteq \mathcal{V}$ the set of all variables used in a or b. Since the set is finite we can create a vector $\boldsymbol{d} = (d_1, \ldots, d_n)$ of all variables used in a or b. We assume $d' \subseteq \mathcal{V}$ a set of fresh variables with the same size as d. Similarly, we have $\boldsymbol{d'} = (d'_1, \ldots, d'_n)$ a vector of fresh variables.

We consider two states $s, f : \mathcal{V} \to \mathbb{R}$, such as $s \xrightarrow{a} f$. We will demonstrate that $s \xrightarrow{b} f$ which, by definition of the refinement, will prove that $a \preceq b$.

We create a new state, s', equal to s on every variables except that $\forall i \in [1; n]$, $s'(d'_i) = f(d_i)$, namely:

$$s'(\boldsymbol{d'}) = f(\boldsymbol{d}) \tag{4.6}$$

Since $s \xrightarrow{a} f$ and $s(\boldsymbol{d}) = s'(\boldsymbol{d})$, we can use the Coincidence Lemma 4. Which expresses that it exists a state f' such as $s' \xrightarrow{a} f'$ and

$$f(\boldsymbol{d}) = f'(\boldsymbol{d}). \tag{4.7}$$

This is illustrated in Figure 4.19.

From the bound lemma 3, we can conclude that $\forall x \notin d$, $s'(x) = f'(x)$. On particular, since all variables in d' are not in d, we have $\forall i \in [1; n]$, $f'(d'_i) = s'(d'_i) = f(d'_i)$. Which we can rewrite as:

$$f(\boldsymbol{d'}) = f'(\boldsymbol{d'}) \tag{4.8}$$

From equation 4.7 and 4.8, we have $f'(\boldsymbol{d}) = f'(\boldsymbol{d'})$. Since $s' \xrightarrow{a} f'$ and $f'(\boldsymbol{d}) = f'(\boldsymbol{d'})$, we can now use the differential refinement lemma 2. It states that their is a state f'' such that $s' \xrightarrow{b} f''$ and $f''(\boldsymbol{d}) = f''(\boldsymbol{d'})$. This is summarised in Figure 4.20.

Thanks to the Coincidence Lemma 4, we can conclude, from $s' \xrightarrow{b} f''$ and $s'(\boldsymbol{d}) = s(\boldsymbol{d})$ by definition of s', that their is a state f''' such as $s \xrightarrow{b} f'''$ and $f'''(\boldsymbol{d}) = f''(\boldsymbol{d})$. This is illustrated in Figure 4.21.

We will now prove that $\forall x \in \mathcal{V}, f'''(x) = f(x)$. First if $x \notin d$, from the bound effect lemma 3, $f'''(x) = s(x)$ and (from the same lemma) $s(x) = f(x)$, hence $f'''(x) = f(x)$. Then, we prove that $f'''(\boldsymbol{d}) = f(\boldsymbol{d})$. From the coincidence lemma, we have $f'''(\boldsymbol{d}) = f''(\boldsymbol{d})$. We also have $f''(\boldsymbol{d}) = f''(\boldsymbol{d'})$ and from bound effect, $f''(\boldsymbol{d'}) = s'(\boldsymbol{d'})$. And from Equation 4.6 $s'(\boldsymbol{d}) = f(\boldsymbol{d})$. So $f'''(\boldsymbol{d}) = f''(\boldsymbol{d'}) = s'(\boldsymbol{d'}) = f(\boldsymbol{d})$ which finishes the proof. □

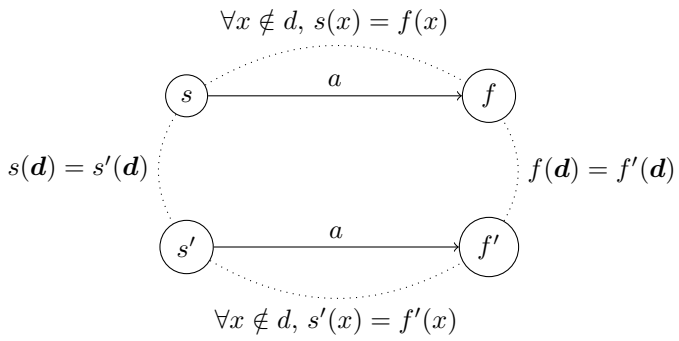


Figure 4.19: Definition of f' with coincidence lemma

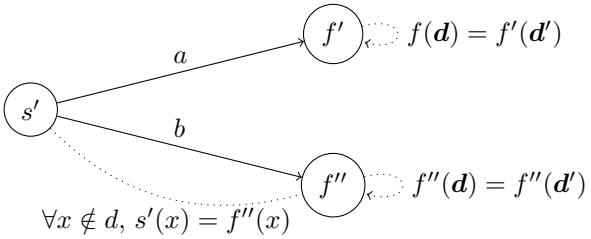


Figure 4.20: Definition of f'' with differential refinement lemma

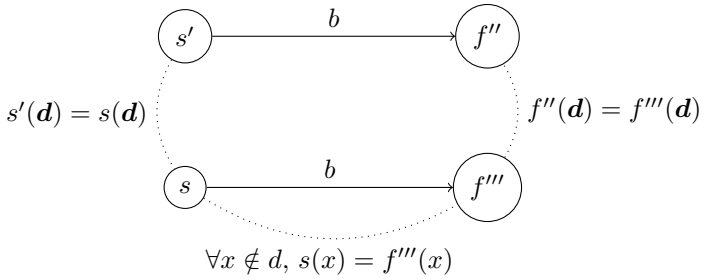


Figure 4.21: Definition of f''' with coincidence lemma

4.5 Abstract Programs

In this section, we introduce *abstract programs*, a subset of hybrid programs specifically designed to define contracts. So far, we expressed three requirements for the definition of abstract programs, they must be:

- closed by conjunction, disjunction and negation,
- writable as hybrid programs, and
- expressive enough to specify of systems.

We also want to instantiate the type class described in Section 3.7, which we do in Section 4.6. The instantiation will give us contracts that are sufficient to specify the examples of Section 4.2, and are compatible with the components defined in Section 4.3.

4.5.1 Definition

Abstract programs consist in the disjunctive normal form of atoms. Atoms are hybrid formulas which denote either a precondition or a postcondition. An `atom` is made of two parts, the `atom_` with the formula which can be either a precondition or a postcondition, and `atom_in_alphabet` which is the proof that the alphabet of the formula is `d`.

```
Inductive atom_ :=
  | preF : Formula → atom_
  | postF : Formula → atom_.

Definition atom_in_alphabet (d : alphabet) (e : atom_) : Prop :=
  match e with
  | preF f ⇒ formula_in_alphabet d f
  | postF f ⇒ formula_in_alphabet d f
  end.

Definition atom (d : alphabet) := {e | atom_in_alphabet d e}.
```

The type `aProgram` is an abstract program, coded as a list of list of atoms. The sublists of atoms encode the conjunction of the atoms, and the global list is the disjunction of all sublists. An example of abstract program is represented on Figure 4.22.

```
Definition aProgram (d : alphabet) :=
 list (list (atom d)).
```

4.5.2 Satisfaction Function

The satisfaction function works inductively on the structure of the abstract programs. It returns a proposition determining if a behavior satisfies an abstract program.

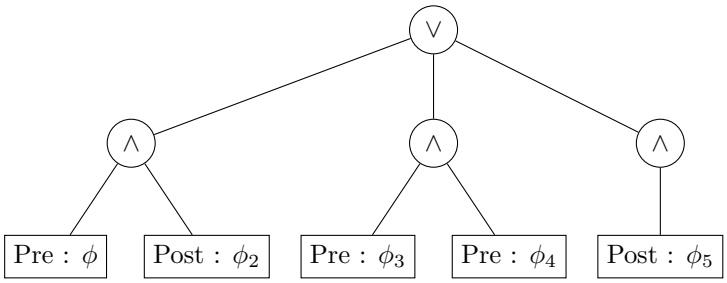


Figure 4.22: The structure of abstract programs

Satisfaction of Atoms The satisfaction of an atom depends on the nature of the atom. If it is a "Pre" atom, we check that the initial state of the behavior satisfies the formula in differential dynamic logic. If it's a "Post" atom, we check that the final state of the behavior satisfies the formula. The `proj1_sig e` unwraps the `atom` and returns the `atom_` inside it.

```
Definition aProgram_sat_atom (d : alphabet) (t : behavior d) (e : atom d) : Prop :=
  let (prestate, poststate) := to_transition t in
  match proj1_sig e with
  | preF f ⇒ dynamic_semantics_formula I f prestate
  | postF f ⇒ dynamic_semantics_formula I f poststate
  end.
```

Satisfaction of Abstract Programs The satisfaction of an abstract program is structural: the function `aProgram_sat_aux` takes a list of atoms and holds if the behavior satisfies all the atoms; `aProgram_sat` holds if one of the lists of atoms is satisfied by the behavior.

```
Fixpoint aProgram_sat_aux (d : alphabet) (t : behavior d) (sp_and : list (atom d)) : Prop :=
  match sp_and with
  | nil ⇒ True
  | h :: q ⇒ aProgram_sat_atom d t h ∧ aProgram_sat_aux d t q
  end.

Fixpoint aProgram_sat (d : alphabet) (s : behavior d) (sp : aProgram d) : Prop :=
  match sp with
  | nil ⇒ False
  | h :: t ⇒ aProgram_sat_aux d s h ∨ aProgram_sat d s t
  end.
```

4.5.3 Construction Operators

In this section we define the usual operators defined for expressions.

Disjunction The disjunction operator is made by appending two `aProgram` as seen on Figure 4.23

```
Definition aProgram_or (d : alphabet) (a b : aProgram d) : aProgram d :=
  a ++ b.
```

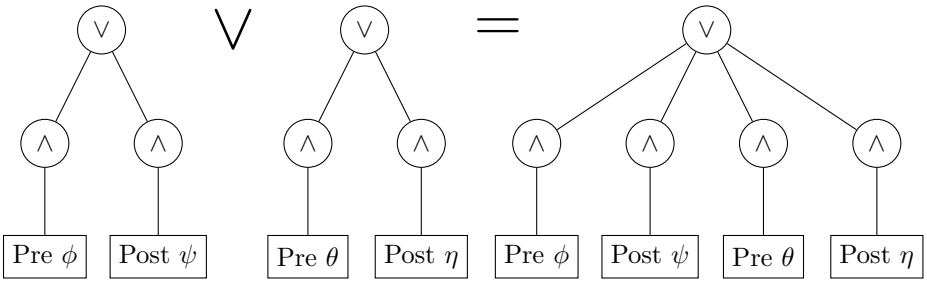


Figure 4.23: The $\vee$ operator.

Conjunction The conjunction operator needs a little more work. We first need to define `aProgram_and_aux` which, for a list of atom and a program, returns the conjunction of the lists of atoms and the program. The idea is to recursively make the conjunction of an atom a and a list of disjunction $l = h :: t$ with the equality: $a \wedge (h \vee t) = (a \wedge h) \vee (a \wedge t)$. Then, the "and" operator needs to call this function on all sub-lists of the program. We give an example in Figure 4.24.

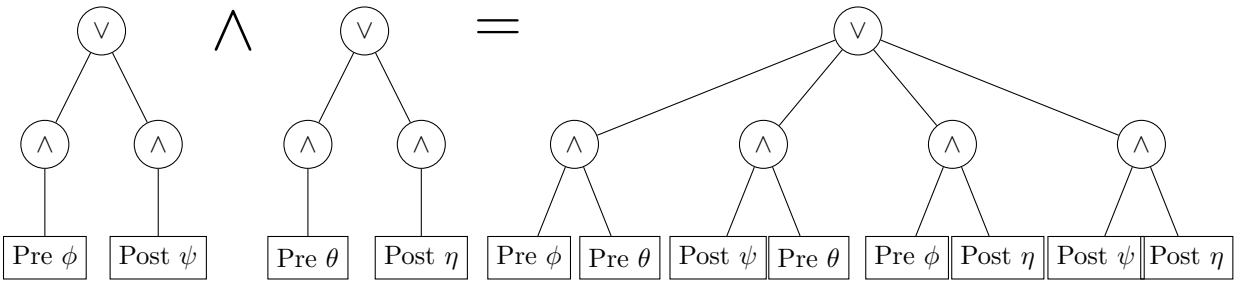


Figure 4.24: The $\wedge$ operator

```
Fixpoint aProgram_and_aux (d : alphabet) (l : list ( atom d)) (prog : aProgram d) : aProgram d :=
  match prog with
  | nil ⇒ nil
  | h ::  t ⇒ (l ++ h) :: aProgram_and_aux d l t
  end.
```

```
Fixpoint aProgram_and (d : alphabet) (a b : aProgram d) : aProgram d :=
  match a with
  | nil ⇒ []
  | h :: t ⇒ aProgram_or d (aProgram_and_aux d h b) (aProgram_and d t b)
  end.
```

Negation To define the negation of an abstract program, we first define the negation of an element `aProgram_not`, as shown on Figure 4.25. We define the negation of list of atom with `aProgram_not_aux`. With De Morgan's Law, the negation of a conjunction of atoms is the disjunction of their negation. Then, we use again the De Morgan's law to compute the negation of the whole abstract program with `aProgram_not`. An example is given in Figure 4.26.

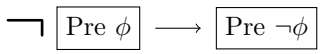


Figure 4.25: The negation operator of an atom

```
Definition aProgram_not_atom (d : alphabet) (e : atom d) : atom d :=
  match e with
  | exist (preF f) pf ⇒ exist _ (preF (KFnot f)) (KFnot_in_d d f pf)
  | exist (postF f) pf ⇒ exist _ (postF (KFnot f)) (KFnot_in_d d f pf)
  end.

Fixpoint aProgram_not_aux (d : alphabet) (l : list (atom d)) : aProgram d :=
  match l with
  | nil ⇒ []
  | h :: t ⇒ ((aProgram_not_atom d h)::nil) :: (aProgram_not_aux d t)
  end.

Fixpoint aProgram_not (d : alphabet) (sp : aProgram d) : aProgram d :=
  match sp with
  | nil ⇒ nil :: nil
  | h :: t ⇒ aProgram_and d (aProgram_not_aux d h) (aProgram_not d t)
  end.
```

4.5.4 Transforming to Hybrid Programs

In this section, we explain how to transform an abstract program into a hybrid program. This is done inductively on the structure of `aProgram`. More precisely, there is a function that transforms list of atoms into programs. Then, these programs are joined with the choice operator.

Translating of an atom The translation of the two kind of atoms is described in Figure 4.27.

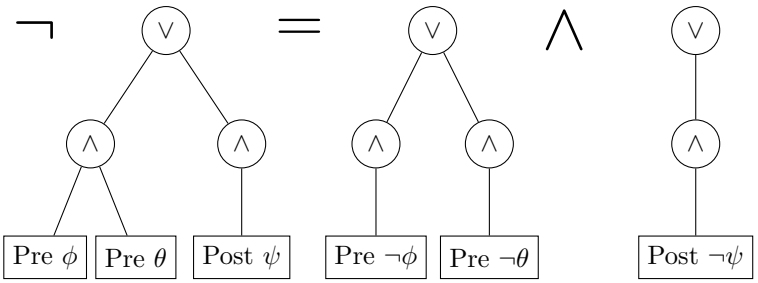


Figure 4.26: The negation using de Morgan's rule

$$\boxed{\text{Pre } \phi} \longrightarrow (?\phi \ ; \ \boldsymbol{d} = * \ ; \ ?\top)$$

$$\boxed{\text{Post } \psi} \longrightarrow (?\top \ ; \ \boldsymbol{d} = * \ ; \ ?\psi)$$

Figure 4.27: Transforming a abstract atom into an hybrid program

Undeterminate Assignation of Alphabet We use $\boldsymbol{d} = *$ which stands for the indeterminate assignment of all variables in alphabet d. This implies that the alphabet d must be finite, we use the same axioms as in Section 4.4. To define this operator `AssignAny`, we define `assignAny_aux` which iterate over the identifiers of the alphabet and compose hybrid programs that assign any value to them.

```
Fixpoint assignAny_aux (l : list ident) : Program :=
  match l with
  | nil ⇒ KPtest KFtrue
  | h ::  t ⇒ KPcompose (KPassignAny h) (assignAny_aux t)
  end.

Definition assignAny (d : alphabet) (fd : Finite d) : Program :=
  assignAny_aux (elements_of d fd).
```

Aggregation of List of Atoms To transform a list of atoms into an hybrid programs, we aggregate the atoms that are precondition and the atoms that are postconditions with `flatten_pre` and `flatten_post`. Then the function `to_program_aux` creates the program with the sequence of the precondititon, `assignAny d` and the postcondition. We give an example of the process in Figure `fg_list_hp`.

```
Fixpoint flatten_pre (d : alphabet) (l : list (atom d)) : Formula :=
  match l with
  | nil ⇒ KFtrue
  | exist (preF f) _ :: t ⇒ KFand f (flatten_pre d t)
  | exist (postF _) _ :: t ⇒ flatten_pre d t
  end.

Fixpoint flatten_post (d : alphabet) (l : list (atom d)) : Formula :=
  match l with
  | nil ⇒ KFtrue
  | exist (preF _) _ :: t ⇒ flatten_post d t
  | exist (postF f) _ :: t ⇒ KFand f (flatten_post d t)
  end.

Definition to_program_aux (d : alphabet) (fd : Finite d) (s : list (atom d)) : Program :=
      KPcompose
        (KPcompose
          (KPtest (flatten_pre d s))
          (assignAny d fd))
        (KPtest (flatten_post d s))
        .
```

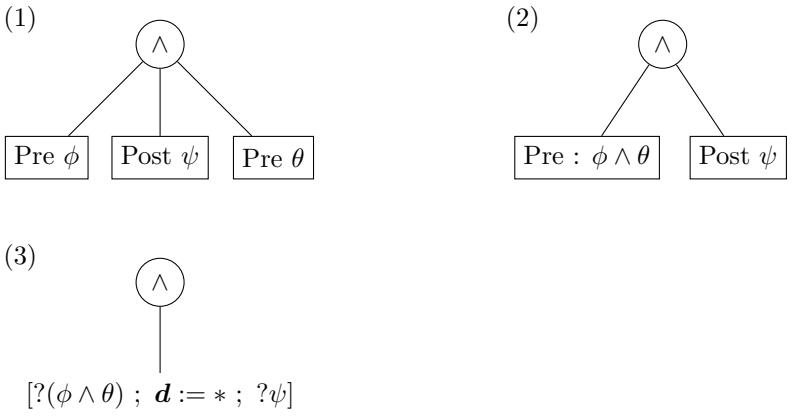


Figure 4.28: Transforming a list of abstract atom into an hybrid program

Transforming an Abstract Program To transform an abstract program to an hybrid program, we first transform each sublists to hybrid programs with `to_program_aux`. Then we join each program with the choice operator of differential dynamic logic with `to_program` as shown on Figure 4.29.

```
Fixpoint to_program (d : alphabet) (fd : Finite d) (s : aProgram d) : Program :=
  match s with
  | nil ⇒ KPtest KFfalse
  | h :: t ⇒ KPchoice (to_program_aux d fd h) (to_program d fd t)
  end.
```

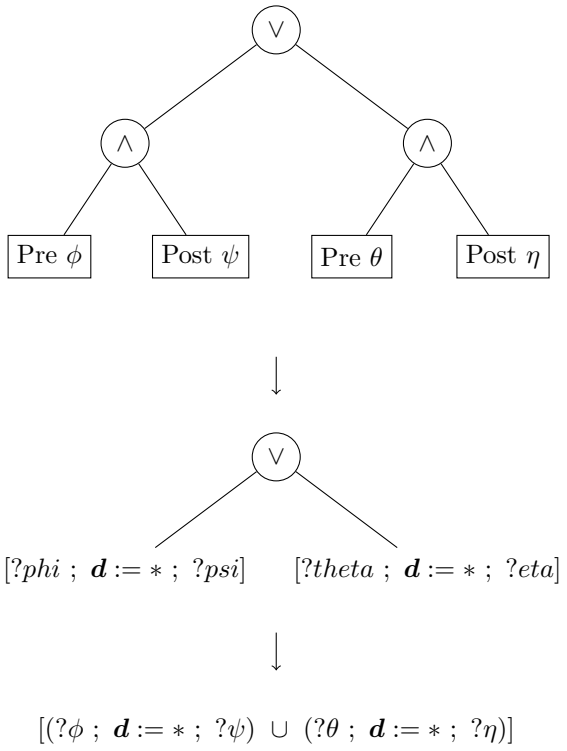


Figure 4.29: Transforming an abstract program into a hybrid program

4.5.5 Proving the Transformation is Sound

We want to prove that `to_program` is correctly defined, to do that we prove that a behavior `t` satisfying an abstract program `sp aProgram_sat d t sp` also satisfies the translation of the abstract program `to_program d fd sp`.

Proof of Correctness We want to prove that the translation function is correct. The proof is done by induction on the abstract program. First we prove that `to_program_aux` is sound, then with induction we will prove `to_program` is sound. We need to prove that every behavior on `d` satisfy `assignAny d`. This is done by induction on the list of identifiers in the alphabet `d`.

```
Theorem sat_to_program_sound :
    ∀ (d : alphabet) (fd : Finite d) (sp : aProgram d) (t : behavior d),
    let (preS, postS) := to_transition t in
    dynamic_semantics_program I (to_program d fd sp) preS postS ↔ aProgram_sat d t sp.

Lemma sat_to_program_sound_aux :
  ∀ (d : alphabet) (fd : Finite d) (a : list (atom d)) (t : behavior d),
  let (preS, postS) := to_transition t in
  dynamic_semantics_program I (to_program_aux d fd a) preS postS ↔
  aProgram_sat_aux d t a.

Lemma assignAny_dsp : ∀ (d : alphabet) (fd : Finite d) (t : behavior d),
  let (preS, postS) := to_transition t in
  dynamic_semantics_program I (assignAny d fd) preS postS.
```

4.6 Contracts with Differential Dynamic Logic

This section instantiates the assumption/guarantee theory of contracts withwith abstract programs. We are going to instantiate the theory using the interface we defined in Section 3.7. Then, we focus on defining contracts using abstract programs. We show how to transform an implementation property into a hybrid formula to prove in $\mathbf{d}\mathcal{L}$. Finally we use the composition given by the instantiation to create a contract for the system, and use the contract as a validation of an hybrid program modeling the system.

4.6.1 Instantiating the Theory of Contract

Using the Interface We use the interface defined in Section 3.7, it is particularly fit for this purpose because abstract programs are alphabetized expressions. To be able to use hybrid programs and abstract programs in conjunction, we use the same types for the definition of behaviors, namely `KAssignable` for `ident` and (`R * R`) for `value`. The instantiation needs an expression type, operators, proof of correctness of operators and decidability of satisfaction function. The operators "and", "or", and "not" and the satisfaction function were developed in Section 4.5. We still need to prove their correctness and the decidability of the satisfaction function.

Proof Operators are Correct We prove that each operator respect its specification. For each operator, the proof is done by induction on `sp1` and poses no real challenge.

```
Lemma aProgram_or_correct : forall (d : alphabet) (s : behavior d) (sp1 sp2 : aProgram d),
  aProgram_sat d s sp1 ∨ aProgram_sat d s sp2 ↔ aProgram_sat d s (aProgram_or d sp1 sp2).

Lemma aProgram_and_correct : forall (d : alphabet) (s : behavior d) (sp1 sp2 : aProgram d),
  aProgram_sat d s sp1 ∧ aProgram_sat d s sp2 ↔ aProgram_sat d s (aProgram_and d sp1 sp2).
```

```
Lemma aProgram_not_correct : forall (d : alphabet) (s : behavior d) (sp : aProgram d),
  ¬ aProgram_sat d s sp ↔ aProgram_sat d s (aProgram_not d sp).
```

Decidability We axiomatize the decidability of the satisfaction function, since proving it is outside of our scope of work.

```
Axiom aProgram_sat_dec : forall (d : alphabet) (s : behavior d) (sp : aProgram d),
  aProgram_sat d s sp ∨ ¬aProgram_sat d s sp.
```

Now, we are ready to instantiate the type class `AlphabetizedExpression`

```
Instance abstract_program : AlphabetizedExpression (R*R) KAssignable aProgram := {
  e_and := aProgram_and ;
  e_or := aProgram_or ;
  e_not := aProgram_not ;
  sat := aProgram_sat ;
  sat_e_not := aProgram_not_correct ;
  sat_e_or := aProgram_or_correct ;
  sat_e_and := aProgram_and_correct ;
  sat_dec := aProgram_sat_dec ;
}.
```

Results This instantiation gives us operators to define contracts. The function `contractF` constructs a contract given two abstract programs. The function `saturateF` computes the saturated version of a contract. The function `composeF` computes the composition of two contracts. For the example we define `aSaturate` and `aCompose`, to facilitate the use of the operators.

```
Definition aSaturate := @saturateF _ _ _ (abstract_program I) d.
Definition aCompose := @composeF _ _ _ (abstract_program I) d.
```

4.6.2 Example of Contracts

Contract for the valve The valve has no assumption, so we say its assumption holds for any behavior. We define `aTrue` as an abstract program that holds for any behavior. The guarantee has one atom with formula $h \geq H_{limit} \rightarrow v = 0$, which we reproduce in `atom_1_f`. After having proved that this formula's alphabet is in `d`, we create the guarantee. The contract is described in Figure 4.30.

```
Definition ca_assume := aTrue d.

Definition atom_1_f : atom_ := postF ((KFimply
      (KFgreaterEqual (KTread h) HLimit)
      (KFequal v (KTnumber (KTNreal 0))))).
Lemma atom_1_in_d : atom_in_alphabet d atom_1_f. Proof. firstorder. Qed.

Definition atom_1 := exist _ atom_1_f atom_1_in_d.
Definition ca_guarantee : aProgram d := [[ atom_1 ]].

Definition ca_contract : contractF d := ContractF d ca_assume ca_guarantee.
```

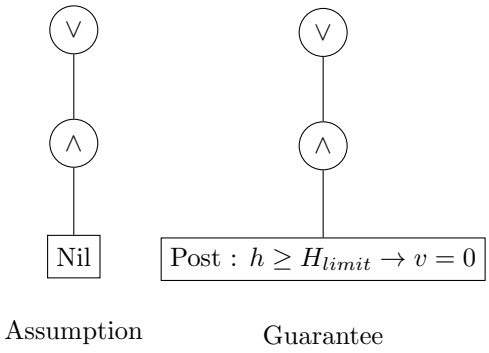


Figure 4.30: The contract of the valve α

Contract of the Water Tank We create the contract for the water tank. The tank assumes that the level of water h starts below H_{max} wich we encode in `atom_2`. The assumption is the conjunction of `atom_2` and `atom_1` the guarante of the valve. The water tank guarantee it doesn't overflow, we write this assertion as `atom_3`. We give Figure 4.31 a visual representation of the contract.

```
Definition atom_2_f := preF (KFlessEqual h HMax).
Lemma atom_2_in_d : atom_in_alphabet d atom_2_f.
Proof. firstorder. Qed.
Definition atom_2 := exist _ atom_2_f atom_2_in_d.

Definition cb_assume : aProgram d := [[ atom_2 ; atom_1 ]].

Definition atom_3_f := postF (KFlessEqual h HMax).
Lemma atom_3_in_d : atom_in_alphabet d atom_3_f.
Proof. firstorder. Qed.
Definition atom_3 := exist _ atom_3_f atom_3_in_d.

Definition cb_guarantee : aProgram d := [[ atom_3 ]].

Definition cb_contract := ContractF d cb_assume cb_guarantee.
```

4.6.3 Implementation of a Contract by a Component

Thanks to the previous definitions, we can express the implementation of a contract by a component as a $\mathbf{d}\mathcal{L}$ property. Indeed the assumption/guarantee theory of contract we can show that for a saturated contract $c = (A, G)$ and a component σ

$$\sigma \vdash c \equiv \sigma \preceq G \tag{4.9}$$

With the help the relation we proved in Section 4.4, we can express the refinement as a a differential refinement which is provable as a property of differential dynamic logic. We also need to use the translations from abstract programs to

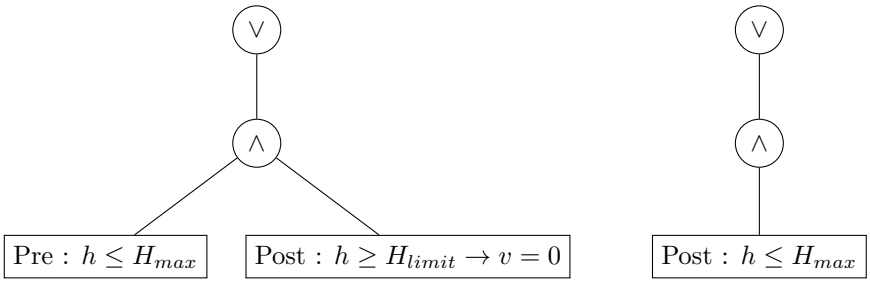


Figure 4.31: The contract of the water tank

hybrid programs to transform the contract's guarantee into an hybrid program. This is proved in `Theorem proof_trans`.

```
Theorem proof_trans (d : alphabet) (fd : Finite d)
  (a : Program) (a_in_d : prog_in_d d a) (c : contractF d) :
  (forall preS : KState,
  (dynamic_semantics_formula I
    (KFrefine d fd a (to_program d fd (G d (saturateF d c)))) preS)) →
  implements d (to_component I a a_in_d) (contract_of d c).
```

Example Here we show how we would discharge the proof that the hybrid program modeling the valve is an implementation of the contract to a differential dynamic logic formula with `proof_trans`.

```
Theorem ca_implements_alpha :
  (forall preS, dynamic_semantics_formula I
    (KFrefine d d_finite alpha
      (to_program d d_finite (G d (aSaturate ca_abstract_contract))))
    preS)
  → implements d alpha_component (to_contract ca_abstract_contract).
Proof.
  apply (proof_trans I d fd).
Qed.
```

4.6.4 Composition of Contracts

Thanks to the instantiation of the theory of assumption/guarantee contracts, the composition of contracts made with abstract program is already defined.

Example We build the contract for the system by composition of the contract of the valve and the water tank. Since the guarantee of the valve is the same as the assumption of the water tank, the assumption of the system reduce to a more simple abstract program. The mechanism of reduction is not automatised in our formalisation but we can prove this result by hand. We show in Figure 4.32 a representation of the contract.

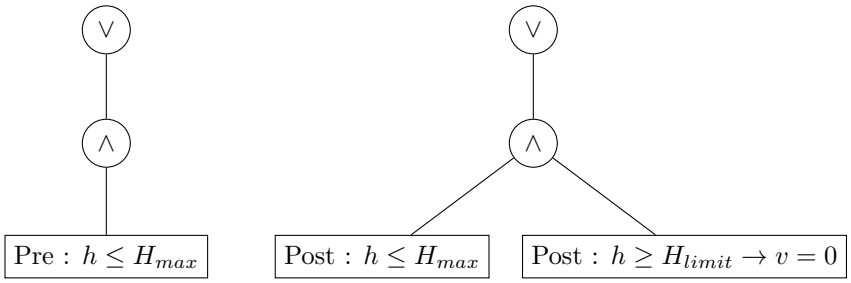


Figure 4.32: The (reduced) contract of the system

```
Definition cg_contract : contractF d :=
  aCompose ca_abstract_contract cb_contract.

Lemma cg_contract_reduction :
  aSaturate cg_contract =
  aSaturate (ContractF d [[ atom_2 ]] [[  atom_1 ; atom_3 ]]).
```

System's Hybrid Program The hybrid program modeling the system was written by hand. It is not the result of an operation between α and β. Because, to the best of our knowledge, the parallel composition of hybrid programs is not defined in differential dynamic logic. The proof obligation is now to verify that γ implements the contract `cg_contract`. With `proof_trans`, we can discharge this proof into differential dynamic logic. This make it possible to validate the component γ as the correct composition of the valve and the water-tank.

$$\gamma = ((\dot{h} = v \& h < H_{limit}) \; ; (?h < H_{limit} \vee v := 0))^* \tag{4.10}$$

```
Definition gamma :=
  KPloop
    (KPcompose
      (KPchoice
        (KPtest (KFlessEqual (KTread h) HMax))
        (KPassign v (KTnumber (KTNreal 0%R)))
      )
      (KPodeSystem
        (ODEatomic (ODEsing (KAssignDiff h) (KTread v)))
        (KFless h HMax)
      )
    ).
```

```
Theorem gamma_implments_cg :
  (forall preS : KState,
  dynamic_semantics_formula I
    (KFrefine d fd gamma (to_program d fd (G d (aSaturate cg_contract))))
    preS)
  →
  implements d gamma_component (to_contract cg_contract).
Proof.
  apply (proof_trans I d fd).
Qed.
```

4.7 Conclusion

In this chapter, we showed two different instantiation of the assumption/guarantee theory of contracts. The first instantiation used the generic interface, simply by implementing the type class. The instantiation of the verified contract theory gives a natural definition of the refinement relation. Yet, a refinement relation was already defined for hybrid programs [LP16]. We proved that this refinement implies the refinement defined in the assume/guarantee contracts, which allow us to prove refinement relations by proving hybrid formulas using specialized tools such as Keymaera X. To define contracts in differential dynamic logic, we introduced abstract programs. We gave the algorithms to construct and manipulate abstract programs, and the function to decide the satisfaction of an abstract programs by behaviors. We also gave the translation function from abstract programs to hybrid programs. Finally, we proved this function to be correct, meaning a behavior satisfying the abstract programs also satisfies the translated hybrid programs and vice versa. Next, we showed how both instances work together. Specifically, we defined the implementation relation of a contract by a component. The contract being written with abstract programs, and the component being written with hybrid programs. We also gave the water tank system as an example of an use case for components and contracts. This example demonstrates the usefulness of modularity and abstraction in the process of verifying systems. The water tank system being a cyber-physical system, the example uses all the potentials of differential dynamic logic to model ordinary differential equation. This example showed how the mechanization of assumption/guarantee contracts can be useful to define contracts with different logics. The aim of this mechanization is to be a common base of definition of contracts. Since different logic have different features, their natives definition of components and contract are different. The Verified Contract Theory shows we can link theses definitions together.

Chapter 5

Conclusion

5.1 Overview

We presented a formalization of the set-theoretical assumption/guarantee contracts in the proof assistant Coq. To the best of our knowledge, it is the first mechanized formalization of a theory of assumption/guarantee contracts for system design. The formalization gives us the assurance that the notion of assumption/guarantee contract is a correct instance of the meta-theory of contract. We also formalized the mechanism to extend the alphabet on which a contract is defined, as well as the abstraction of a contract by eliminating one its variable. Both mechanisms are validated regarding the meta-theory of contract. Finally, we introduce an interface made to facilitate to instantiation of the theory.

In the second part of our thesis, we instantiated the theory with differential dynamic logic. It is a practical demonstration of the use of the parametric assumption/guarantee contract theory. Firstly, we instantiated the theory to construct components from hybrid programs. We exhibit the relation between the refinement of hybrid program and the refinement of component define in the assumption/guarantee theory of contract. Secondly, we defined abstract programs, to instantiate the theory of contract with them. Abstract programs are a restriction of hybrid programs, though they are closed by conjunction and negation, thus we also gave the translation function from abstract programs to hybrid program. The instantiation of the theory of contract with abstract programs gave us contract defined with abstract programs as assumptions and guarantee. The contracts came with the operators defined in the theory, namely, conjunction, refinement and composition. The implementation of a contract component by a contract is also defined between the components defined as hybrid programs and the contracts defined with abstract programs. Finally, we introduced an example to apply all the definitions above, it gives a practical usage of the theory of contract to specify a system.

5.2 Perspectives

One of our interest would be to exhibit the links between the works done to define contracts and components for differential dynamic logic. Lunel defines components for differential dynamic logic with a particular form [Lun+19] whereby components are considered as the sequences of discrete computations followed by a continuous behaviour. This fixed form allows to define a commutative and associative composition operator. The operator is only defined if its arguments validate specific timing requirements. The operator defines the sequence of the discrete computations and the parallelization of the continuous behaviour. It would be interesting to formally establish the relation between this operator and the operator naturally defined from our instantiation mechanism.

Müller et al. define components and contracts for differential dynamic logic [Mül+18] in a similar manner as Lunel, yet without explicit timing constraints. Their model uses ports to manage inputs and outputs of the contracts, and their contracts are specifically made to specify those ports. Müller et al. also define different kinds of contracts to specify specific aspects of a component. The different contracts may be defined as different instances of the theory of contracts.

Another approach would be to instantiate the verified contract theory with another logic, for example duration calculus and hybrid Hoare logic [Liu+10; HC97] It is expressive enough to capture cyber-physical systems, and already has a formalisation in Isabelle [Che+17]. The formalisation in Coq has been investigated already, without complete success to date [CPM03]. Contracts have been defined for hybrid Hoare logic [WZG12], we are interested in the relation between the contracts defined by Wang and the contract that are defined from the contract theory.

We could also augment the mechanization, for example recent work have defined a quotient for the assumption/guarantee contract [Ínc+18]. The quotient is an important part of the meta theory of contract as it permits to exhibit the necessary contracts needed for a system to function. Yet, the quotient has no trivial definition, it is thus missing from our mechanization, we should add the definition of Incer to our formalization.

Nuzzo made some very interesting developments to the assumption/guarantee contracts [Nuz15]. In particular the heterogeneous refinement of contracts seems quite fitted to our mechanization. We could implement it to define refinement of contract from different instances of the theory.

We believe extending the work of our mechanization can help the community understanding the theory of contracts and creating better processes for the specification and validation of systems.

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 13, 2010. 613 pp. ISBN: 978-1-139-64397-9. Google Books: 23UgAwAAQBAJ.

[AK15] Abhishek Anand and Ross Knepper. "ROSCoq: Robots Powered by Constructive Reals". In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 34–50. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_3.

[AL93] Martín Abadi and Leslie Lamport. "Composing Specifications". In: *ACM Transactions on Programming Languages and Systems* 15.1 (Jan. 1, 1993), pp. 73–132. ISSN: 0164-0925. DOI: 10.1145/151646.151649. URL: https://doi.org/10.1145/151646.151649 (visited on 03/17/2021).

[AL95] Martín Abadi and Leslie Lamport. "Conjoining Specifications". In: *ACM Transactions on Programming Languages and Systems* 17.3 (May 1995), pp. 507–535. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/203095.201069. URL: https://dl.acm.org/doi/10.1145/203095.201069 (visited on 03/17/2021).

[Alu+93] Rajeev Alur et al. "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems". In: *Hybrid Systems*. Ed. by Robert L. Grossman et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1993, pp. 209–229. ISBN: 978-3-540-48060-0. DOI: 10.1007/3-540-57318-6_30.

[Bau+12] Sebastian S. Bauer et al. "Moving from Specifications to Contracts in Component-Based Design". In: *Fundamental Approaches to Software Engineering*. Ed. by Juan de Lara and Andrea Zisman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 43–58. ISBN: 978-3-642-28872-2. DOI: 10.1007/978-3-642-28872-2_3.

[BC13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, Mar. 14, 2013. 492 pp. ISBN: 978-3-662-07964-5. Google Books: FeklBQAAQBAJ.

[Beh+99] Patrick Behm et al. "Météor: A Successful Application of B in a Large Project". In: *FM'99 — Formal Methods*. Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 369–387. ISBN: 978-3-540-48119-5. DOI: 10.1007/3-540-48119-2_22.

[Ben+08] Albert Benveniste et al. "Multiple Viewpoint Contract-Based Specification and Design". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 200–225. ISBN: 978-3-540-92188-2. DOI: 10.1007/978-3-540-92188-2_9.

[Ben+15a] Albert Benveniste et al. *Contracts for Systems Design: Methodology and Application Cases*. report. July 2015. URL: https://hal.inria.fr/hal-01178469 (visited on 12/02/2019).

[Ben+15b] Albert Benveniste et al. *Contracts for Systems Design: Theory*. report. INRIA, July 2015. URL: https://hal.inria.fr/hal-01178467 (visited on 11/29/2019).

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. "Coquelicot: A User-Friendly Library of Real Analysis for Coq". In: *Mathematics in Computer Science* 9.1 (Mar. 1, 2015), pp. 41–62. ISSN: 1661-8289. DOI: 10.1007/s11786-014-0181-1. URL: https://doi.org/10.1007/s11786-014-0181-1 (visited on 03/03/2023).

[BNH14] Albert Benveniste, Dejan Nickovic, and Thomas Henzinger. *Compositional Contract Abstraction for System Design*. report. Jan. 29, 2014. URL: https://hal.inria.fr/hal-00938854 (visited on 10/29/2019).

[Boh+17] Brandon Bohrer et al. "Formally Verified Differential Dynamic Logic". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France). CPP 2017. New York, NY, USA: ACM, 2017, pp. 208–221. ISBN: 978-1-4503-4705-1. DOI: 10.1145/3018610.3018616. URL: http://doi.acm.org/10.1145/3018610.3018616 (visited on 12/02/2019).

[Che+17] Mingshuai Chen et al. "MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems". In: *Provably Correct Systems*. Ed. by Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog. NASA Monographs in Systems and Software Engineering. Cham: Springer International Publishing, 2017, pp. 39–58. ISBN: 978-3-319-48628-4. URL: https://doi.org/10.1007/978-3-319-48628-4_3 (visited on 01/13/2020).

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. "A Calculus of Durations". In: *Information Processing Letters* 40.5 (Dec. 13, 1991), pp. 269–276. ISSN: 0020-0190. DOI: 10.1016/0020-0190(91)90122-X. URL: http://www.sciencedirect.com/science/article/pii/002001909190122X (visited on 12/23/2019).

[CJR96] Zhou Chaochen, Wang Ji, and Anders P. Ravn. "A Formal Description of Hybrid Systems". In: *Hybrid Systems III*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 511–530. ISBN: 978-3-540-68334-6. DOI: 10.1007/BFb0020972.

[CM20] Zheng Cheng and Dominique Méry. *A Refinement Strategy for Hybrid System Design with Safety Constraints*. Research Report. Université de Lorraine ; INRIA ; CNRS, July 2020. URL: https://hal.inria.fr/hal-02895528.

[Cou+12] Denis Cousineau et al. "TLA+ Proofs". In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 147–154. ISBN: 978-3-642-32759-9. DOI: 10.1007/978-3-642-32759-9_14.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. "Thoughts about the Implementation of the Duration Calculus with Coq". In: *Fourth Workshop on the Implementation of Logics*. 2003, p. 33.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[Dup+18] Guillaume Dupont et al. "Proof-Based Approach to Hybrid Systems Development: Dynamic Logic and Event-B". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Michael Butler et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 155–170. ISBN: 978-3-319-91271-4. DOI: 10.1007/978-3-319-91271-4_11.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann, Oct. 23, 2014. 631 pp. ISBN: 978-0-12-800800-3. Google Books: Ze60AwAAQBAJ.

[Fos+20] Simon Foster et al. "Unifying Theories of Reactive Design Contracts". In: *Theoretical Computer Science* 802 (Jan. 8, 2020), pp. 105–140. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.09.017. arXiv: 1712.10233. URL: https://www.sciencedirect.com/science/article/pii/S0304397519305614 (visited on 07/02/2021).

[Fre10] Patrick Christopher Frey. "A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems". PhD thesis. University of Ulm, 2010. URL: http://vts.uni-ulm.de/docs/2011/7505/vts%5C_7505%5C_10701.pdf.

[Ful+15] Nathan Fulton et al. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 527–538. ISBN: 978-3-319-21401-6. DOI: 10.1007/978-3-319-21401-6_36.

[FW17] Simon Foster and Jim Woodcock. "Towards Verification of Cyber-Physical Systems with UTP and Isabelle/HOL". In: *Concurrency, Security, and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*. Ed. by Thomas Gibson-Robinson, Philippa Hopcroft, and Ranko Lazić. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 39–64. ISBN: 978-3-319-51046-0. DOI: 10.1007/978-3-319-51046-0_3. URL: https://doi.org/10.1007/978-3-319-51046-0_3 (visited on 03/30/2020).

[FZW15] Simon Foster, Frank Zeyda, and Jim Woodcock. "Isabelle/UTP: A Mechanised Theory Engineering Framework". In: *Unifying Theories of Programming*. Ed. by David Naumann. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 21–41. ISBN: 978-3-319-14806-9. DOI: 10.1007/978-3-319-14806-9_2.

[GLB87] Thierry Gautier, Paul Le Guernic, and Loïc Besnard. "SIGNAL: A Declarative Language for Synchronous Programming of Real-Time Systems". In: *Functional Programming Languages and Computer Architecture*. Ed. by Gilles Kahn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 257–277. ISBN: 978-3-540-47879-9. DOI: 10.1007/3-540-18317-5_15.

[Gra+18] Susanne Graf et al. "Building Correct Cyber-Physical Systems: Why We Need a Multiview Contract Theory". In: *Formal Methods for Industrial Critical Systems*. Ed. by Falk Howar and Jiří Barnat. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 19–31. ISBN: 978-3-030-00244-2. DOI: 10.1007/978-3-030-00244-2_2.

[Hal+91] N. Halbwachs et al. "The Synchronous Data Flow Programming Language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/5.97300.

[HC97] Michael R. Hansen and Zhou Chaochen. "Duration Calculus: Logical Foundations". In: *Formal Aspects of Computing* 9.3 (May 1, 1997), pp. 283–330. ISSN: 1433-299X. DOI: 10.1007/BF01211086. URL: https://doi.org/10.1007/BF01211086 (visited on 10/13/2020).

[Hen00] Thomas A. Henzinger. "The Theory of Hybrid Automata". In: *Verification of Digital and Hybrid Systems*. Ed. by M. Kemal Inan and Robert P. Kurshan. NATO ASI Series. Berlin, Heidelberg: Springer, 2000, pp. 265–292. ISBN: 978-3-642-59615-5. DOI: 10.1007/978-3-642-59615-5_13. URL: https://doi.org/10.1007/978-3-642-59615-5_13 (visited on 12/02/2019).

[HJ98] Charles Antony Richard Hoare and He Jifeng. *Unifying Theories of Programming*. Vol. 14. Prentice Hall Englewood Cliffs, 1998.

[HP22] Jérôme Hugues and Sam Procter. "Contracts in System Development: From Multiconcern Analysis to Assurance With the Architecture Analysis and Design Language". In: *IEEE Software* 39.4 (July 2022), pp. 34–38. ISSN: 1937-4194. DOI: 10.1109/MS.2022.3167533.

[Hug+22] Jérôme Hugues et al. "Mechanization of a Large DSML: An Experiment with AADL and Coq". In: *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). Oct. 2022, pp. 1–9. DOI: 10.1109/MEMOCODE57689.2022.9954589.

[Ínc+18] Íñigo Íncer Romeo et al. "Quotient for Assume-Guarantee Contracts". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). Oct. 2018, pp. 1–11. DOI: 10.1109/MEMCOD.2018.8556872.

[Inc+22] Inigo Incer et al. "Hypercontracts". In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 674–692. ISBN: 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_36.

[Lam93] Leslie Lamport. "Hybrid Systems in TLA+". In: *Hybrid Systems*. Ed. by Robert L. Grossman et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1993, pp. 77–102. ISBN: 978-3-540-48060-0. DOI: 10.1007/3-540-57318-6_25.

[LBT17] Simon Lunel, Benoît Boyer, and Jean-Pierre Talpin. "Compositional Proofs in Differential Dynamic Logic dL". In: *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*. 2017 17th International Conference on Application of Concurrency to System Design (ACSD). June 2017, pp. 19–28. DOI: 10.1109/ACSD.2017.16.

[Liu+10] Jiang Liu et al. "A Calculus for Hybrid CSP". In: *Programming Languages and Systems*. Ed. by Kazunori Ueda. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 1–15. ISBN: 978-3-642-17164-2. DOI: 10.1007/978-3-642-17164-2_1.

[LP16] Sarah M. Loos and André Platzer. "Differential Refinement Logic". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16. New York, NY, USA: Association for Computing Machinery, July 5, 2016, pp. 505–514. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934555. URL: https://doi.org/10.1145/2933575.2934555 (visited on 10/12/2020).

[Lun+19] Simon Lunel et al. "Parallel Composition and Modular Verification of Computer Controlled Systems in Differential Dynamic Logic". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 354–370. ISBN: 978-3-030-30941-1. DOI: 10.1007/978-3-030-30942-8_22.

[Lun19] Simon Lunel. "Parallelism and Modular Proof in Differential Dynamic Logic." PhD thesis. University of Rennes 1, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02102687.

[Mal+16] G. Malecha et al. "Towards Foundational Verification of Cyber-Physical Systems". In: *2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*. 2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS). Apr. 2016, pp. 1–5. DOI: 10.1109/SOSCYPS.2016.7580000.

[MAL22] Amel Mammar, Meryem Afendi, and Régine Laleau. "Modeling and proving hybrid programs with Event-B: An approach by generalization and instantiation". In: *Science of Computer Programming* 222 (2022), p. 102856. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2022.102856. URL: https://www.sciencedirect.com/science/article/pii/S0167642322000892.

[MEO98] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. "Physical System Modeling with Modelica". In: *Control Engineering Practice* 6.4 (Apr. 1, 1998), pp. 501–510. ISSN: 0967-0661. DOI: 10.1016/S0967-0661(98)00047-1. URL: https://www.sciencedirect.com/science/article/pii/S0967066198000471 (visited on 02/21/2023).

[Mey92] B. Meyer. "Applying 'Design by Contract'". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 1558-0814. DOI: 10.1109/2.161279.

[MN04] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech

and Sergio Yovine. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 152–166. ISBN: 978-3-540-30206-3. DOI: 10.1007/978-3-540-30206-3_12.

[Mül+16] Andreas Müller et al. "A Component-Based Approach to Hybrid Systems Safety Verification". In: *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*. Ed. by Erika Ábrahám and Marieke Huisman. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 441–456. ISBN: 978-3-319-33692-3. DOI: 10.1007/978-3-319-33693-0_28.

[Mül+17] Andreas Müller et al. "Change and Delay Contracts for Hybrid System Component Verification". In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 134–151. ISBN: 978-3-662-54493-8. DOI: 10.1007/978-3-662-54494-5_8.

[Mül+18] Andreas Müller et al. "Tactical Contract Composition for Hybrid System Component Verification". In: *International Journal on Software Tools for Technology Transfer* 20.6 (Nov. 1, 2018), pp. 615–643. ISSN: 1433-2787. DOI: 10.1007/s10009-018-0502-9. URL: https://doi.org/10.1007/s10009-018-0502-9 (visited on 02/04/2020).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Lect.Notes Computer. Tutorial. Berlin Heidelberg: Springer-Verlag, 2002. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9. URL: https://www.springer.com/gp/book/9783540433767 (visited on 07/02/2021).

[Nuz15] Pierluigi Nuzzo. "Compositional Design of Cyber-Physical Systems Using Contracts". UC Berkeley, 2015. URL: https://escholarship.org/uc/item/5hk5w3bg (visited on 07/21/2020).

[Osb+05] Leon F. Osborne et al. *Clarus: Concept of Operations*. FHWA-JPO-05-072. Oct. 1, 2005. URL: https://rosap.ntl.bts.gov/view/dot/3710 (visited on 03/13/2023).

[Pla08] André Platzer. "Differential Dynamic Logic for Hybrid Systems". In: *Journal of Automated Reasoning* 41.2 (Aug. 1, 2008), pp. 143–189. ISSN: 1573-0670. DOI: 10.1007/s10817-008-9103-8. URL: https://doi.org/10.1007/s10817-008-9103-8 (visited on 01/10/2020).

[Pla17] André Platzer. "A Complete Uniform Substitution Calculus for Differential Dynamic Logic". In: *Journal of Automated Reasoning* 59.2 (Aug. 1, 2017), pp. 219–265. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9385-1. URL: https://doi.org/10.1007/s10817-016-9385-1 (visited on 09/17/2021).

[PQ08] André Platzer and Jan-David Quesel. "KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description)". In: *Automated Reasoning*. Ed. by Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 171–178. ISBN: 978-3-540-71070-7. DOI: 10.1007/978-3-540-71070-7_15.

[Raj+10] Ragunathan Rajkumar et al. "Cyber-Physical Systems: The next Computing Revolution". In: *Design Automation Conference*. Design Automation Conference. June 2010, pp. 731–736. DOI: 10.1145/1837274.1837461.

[Ric+15] Daniel Ricketts et al. "Towards Verification of Hybrid Systems in a Foundational Proof Assistant". In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). Austin, TX, USA: IEEE, Sept. 2015, pp. 248–257. ISBN: 978-1-5090-0237-5. DOI: 10.1109/MEMCOD.2015.7340492. URL: http://ieeexplore.ieee.org/document/7340492/ (visited on 09/17/2021).

[RST20] Adnan Rashid, Umair Siddique, and Sofiène Tahar. "Formal Verification of Cyber-Physical Systems Using Theorem Proving". In: *Formal Techniques for Safety-Critical Systems*. Ed. by Osman Hasan and Frédéric Mallet. Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 3–18. ISBN: 978-3-030-46902-3. DOI: 10.1007/978-3-030-46902-3_1.

[SDP12] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*". In: *European Journal of Control* 18.3 (Jan. 1, 2012), pp. 217–238. ISSN: 0947-3580. DOI: 10.3166/ejc.18.217-238. URL: http://www.sciencedirect.com/science/article/pii/S0947358012709433 (visited on 12/11/2019).

[SO08] Matthieu Sozeau and Nicolas Oury. "First-Class Type Classes". In: *Theorem Proving in Higher Order Logics*. Ed. by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 278–293. ISBN: 978-3-540-71067-7. DOI: 10.1007/978-3-540-71067-7_23.

[Tea18] The Coq Development Team. *The Coq Proof Assistant, Version 8.7.2*. Zenodo, Feb. 16, 2018. DOI: 10.5281/zenodo.1174360. URL: https://zenodo.org/record/1174360#.YKJOf5MzZXQ (visited on 05/17/2021).

[WC04] Jim Woodcock and Ana Cavalcanti. "A Tutorial Introduction to Designs in Unifying Theories of Programming". In: *Integrated Formal Methods*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2999. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 40–66. ISBN: 978-3-540-21377-2. DOI: 10.1007/978-3-540-24756-2_4. URL: http://link.springer.com/10.1007/978-3-540-24756-2_4 (visited on 02/21/2023).

[WZG12] Shuling Wang, Naijun Zhan, and Dimitar Guelev. "An Assume/Guarantee Based Compositional Calculus for Hybrid CSP". In: *Theory and Applications of Models of Computation*. Ed. by Manindra Agrawal, S. Barry Cooper, and Angsheng Li. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 72–83. ISBN: 978-3-642-29952-0. DOI: 10.1007/978-3-642-29952-0_13.

[WZZ15] Shuling Wang, Naijun Zhan, and Liang Zou. "An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems". In: *Formal Methods and Software Engineering*. Ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 382–399. ISBN: 978-3-319-25423-4. DOI: 10.1007/978-3-319-25423-4_25.

[Xu+23] Xiong Xu et al. "Semantics Foundation for Cyber-physical Systems Using Higher-order UTP". In: *ACM Transactions on Software Engineering and Methodology* 32.1 (Feb. 13, 2023), 9:1–9:48. ISSN: 1049-331X. DOI: 10.1145/3517192. URL: https://doi.org/10.1145/3517192 (visited on 02/17/2023).

[Yu+15] Huafeng Yu et al. "The challenge of interoperability: model-based integration for automotive control software". In: *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, 58:1–58:6. DOI: 10.1145/2744769.2747945. URL: https://doi.org/10.1145/2744769.2747945.

[Yua+16] Zhenghen Yuan et al. "Hybrid Lustre". In: *Perspectives of System Informatics*. Ed. by Manuel Mazzara and Andrei Voronkov. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 325–340. ISBN: 978-3-319-41579-6. DOI: 10.1007/978-3-319-41579-6_25.

[ZH10] Chaochen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Softcover reprint of hardcover 1st ed. 2004 Edition. Berlin; London: Springer, Dec. 9, 2010. 260 pp. ISBN: 978-3-642-07404-2.

[Zou+14] Liang Zou et al. “Verifying Chinese Train Control System under a Combined Scenario by Theorem Proving”. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Ernie Cohen and Andrey Rybalchenko. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 262–280. ISBN: 978-3-642-54108-7. DOI: 10.1007/978-3-642-54108-7_14.

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Sur la vérification mécanisée de la méta-théorie des contrats et son instanciation à la logique dynamique différentielle

Mots clés : Méthodes formelles, conception par contrats, systèmes cyber-physiques

Résumé : L'augmentation de la complexité et de l'hétérogénéité des systèmes critiques pose un défi dans leur conception et leur assurance de sécurité. Les méthodes formelles sont utilisées pour valider les modèles de système, mais la difficulté réside dans la vérification de la sécurité du système global à partir des spécifications de composants validées. La théorie des contrats résout ce problème en utilisant les contrats d'assomption/garantie comme spécifications de composants. Les contrats sont validés en vérifiant que leurs hypothèses et garanties sur-approximent les pré- et post-conditions résultant des évaluations valides du modèle de composant. Les contrats individuels peuvent être combinés en faisant correspondre les hypothèses et garanties de chaque composant. Le manuscrit définit une formalisation algébrique des contrats d'assomption/garantie implémenté dans le calcul de construction de l'assistant de preuve Coq. Cette formalisation est prouvée pour valider une méta-théorie des contrats de Benveniste et al. pour tous les opérateurs tels que la composition, la conjonction, l'abstraction, le raffinement ainsi que l'introduction et l'élimination de variables. Le cas d'utilisation pratique du modèle de contrat est illustré avec la logique différentielle dynamique et deux instances du modèle de contrats. La théorie est appliquée à une étude de cas pour illustrer sa puissance dans la modélisation de composants pour valider un système cyber-physique.

Title : On the mechanized verification of the meta-theory of contracts and its instantiation to differential dynamic logic

Keywords : Formal methods, contract-based design, cyber physical systems

Abstract : The increasing complexity and heterogeneity of safe-critical systems present a challenge in designing and ensuring their safety. Formal methods are employed to validate system models, but the difficulty lies in verifying the safety at the global level, starting from validated component specifications. Contract theory addresses this problem by using assume/guarantee contracts as component specifications. The theory is equipped with operators for combining contracts and incrementally building a verified specification from the low-level functional requirements up to system-level requirements. Concretely, a contracts abstract component interfaces as pairs of assumptions-guarantees, specifying component-environment relations. This thesis introduces the mechanized formalization of an assume/guarantee contracts algebra in the calculus of construction of the proof assistant Coq. In the meantime, the formalization has been proven correct against the Benveniste et al.'s meta-theory formalized in Coq for all the necessary operators: composition, conjunction, abstraction, refinement, as well as the variable introduction and elimination. The practical use case of the contract model is demonstrated with differential dynamic logic and two instances of the contracts model. The theory is exercised on a case study to illustrate its power in modeling components and contractual abstractions.