UNIVERSITÉ DE GENÈVE
Centre Universitaire d'Informatique

FACULTÉ DES SCIENCES
Professeur D. Buchs

SATEL — A Test Intention Language for Object-Oriented Specifications of Reactive Systems

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Levi Pedro SILVA LÚCIO
du
Portugal

Thèse No 3988

Genève
Atelier d'impression ReproMail
2009

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
mention informatique

Thèse *de Monsieur Levi Pedro SILVA LÚCIO*

intitulée :

"SATEL – A Test Intention Language for Object-Oriented Specifications of Reactive Systems"

La Faculté des sciences, sur le préavis de Monsieur D. BUCHS, professeur adjoint et directeur de thèse (Département d'informatique), Madame M.-C. GAUDEL, professeur (Université de Paris-Sud – Laboratoire de Recherche en Informatique – Orsay, France), Messieurs B. LEGEARD, professeur (Laboratoire d'informatique de l'Université de Franche-Comté – Centre National de Recherche Scientifique - Besançon, France), A. PRETSCHNER, docteur (Eidgenössische Technische Hochschule Zürich – Departement Informatik, Information Security – Zürich, Suisse), et Ph. DUGERDIL, docteur (Département d'informatique), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 17 juin 2008

Thèse - 3988 -

[illegible]

Le Doyen, Jean-Marc TRISCONE

N.B. - La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

Nombre d'exemplaires à livrer par colis séparé à la Faculté : - 7 -

Contents

Remerciements

Je remercie d'abord le Prof. Didier Buchs. Didier m'a accueilli dans son équipe et a toujours oeuvré pour que cette thèse se déroule dans les meilleurs conditions. Sa présence pendant toutes les étapes du développement de ce travail m'a permis d'avancer de façon soutenue et de beaucoup apprendre sur l'importance de l'intuition, la rigueur et de la patience dans le travail scientifique. Didier m'a dirigé avec sensibilité et une grande ouverture d'esprit. Je lui témoigne ma reconnaissance et mon amitié.

Marie-Claude Gaudel, Bruno Legeard, Alexander Pretschner et Phillipe Dugerdil m'ont fait l'honneur d'accepter d'être le jury de ce travail. Leurs commentaires pendant les différentes étapes de rédaction de ce document m'ont permis de l'améliorer et de remettre en question ma vision du test. En particulier, l'intérêt d'Alexander pour la bonne présentation des résultats obtenus m'a beaucoup aidé dans la clarification de cette recherche et de ce document.

Je remercie également tous les membres du groupe SMV pour les discussions fréquentes, les séminaires du vendredi où j'ai beaucoup appris et pu élargir mes connaissances ou pour m'avoir directement aidé dans les taches d'enseignement ou autres pendant que je rédigeai ce document. C'était un plaisir travailler et partager ces dernières années avec Andrei Berlizev, Ang Chen, Steve Hostettler, David Hurzeler, Luís Pedro, Matteo Risoldi et Shane Sendall. Parmis les étudiants qui ont collaboré avec le groupe, je remercie Adrien Chantre et Sérgio Coelho pour leur travail dans l'implémentation de l'éditeur du langage SATEL.

La dernière année de rédaction ce travail a été de "réclusion volontaire" et probablement n'aurait pas été possible sans la compréhension et l'amitié de plusieurs personnes qui m'ont soutenu et motivé au long du parcours: Joachim Flammer a été un ami et un appui depuis presque le début de mon séjour à Genève; les Stockinger — Heinz, Flavia et Kilian — m'ont souvent sauvé d'un samedi ou un dimanche enfermé devant l'ordinateur; ma famille m'a aidé pendant des moments délicats où la motivation était loin d'être a son sommet.

Finalement j'aimerai remercier plusieurs personnes qui m'ont aidé à grandir et à me construire autant qu'homme et que professionnel. Ils ont ouvert des portes dans ma carrière, partagé des bouts de chemin de vie avec moi, m'ont inspiré, ou ont simplement écouté et fait confiance quand cela était important ou indispensable: Orlanda Gomes, Bob Jones, Christiane Bézuchet, Cinzia Borel, Vasco Amaral, Magdalena Żydek et Patricia Simioni.

La recherche présentée dans ce mémoire a été financée par la Fondation Hasler, en particulier par l'initiative DICS (Dependable Information and Communications Systems). Je remercie chaleureusement la Fondation Hasler pour sa fléxibilité et son dévouement au développement de l'informatique en Suisse.

Je veux partir pour mieux revenir

- Diams

Résumé

Cette thèse traite de la problématique de la génération de jeux de tests à partir d'une spécification logicielle. En particulier nous nous intéresserons au langage de spécification CO-OPN(Concurrent Object-Oriented Petri Nets) [1, 2], basé sur les rèseaux algébriques étendus grâce à des méchanismes objets et de distribution.

Pour traiter ce problème nous avons dû revoir le langage CO-OPN. En effet la sémantique du langage ayant été définie par itérations successives, celle-ci s'est avérée inadaptée à la génération de jeux de tests. Nous proposons à travers notre travail une nouvelle version de CO-OPN où la syntaxe et la sémantique ont été complètement revues et qui intègrent les travaux précédents où des descriptions précises et formelles ont été employés.

La contribution principale de cette thèse est le concept d'*intention de test*, que nous avons concrétisé dans le langage SATEL (Semi-Automatic Testing Language). Les *intentions de test* s'inspirent des travaux de Bernot, Gaudel et Marre [3] (BGM) ainsi que de ceux de Péraire et Barbey [4, 5]. Nous proposons l'utilisation de la notion de réduction des *jeux de tests exhaustifs*, restreinte à certaines parties du comportement du système à tester. Il s'agit de rendre la théorie BGM utilisable dans le contexte des systèmes logiciels complèxes où une approche exhaustive n'est adaptée ni à lintervention humaine pour la réduction, ni à l'application de techniques opérationèlles.

La description du langage SATEL, tant sur le plan syntaxique que sémantique, a été réalisée à l'aide de techniques formelles. Cela pour permettre une intégration complète avec la description du langage CO-OPN d'une part; et une description formelle suffisamment détaillée pour permettre une implémentation rapide en programmation logique de l'autre.

CO-OPN

Au niveau du langage CO-OPN nos efforts se sont concentrés sur la construction d'une sémantique pour les concepts de *context module* et de *gate*, introduits récément et pour lesquels il n'y a pas de sémantique bien définie. Ces concepts permettent l'écriture de spécifications hiérarchiques où la modélisation des sorties d'un système est facilitée, enrichissant les possibilités de l'activité de génération de jeux de tests.

Notre nouvelle sémantique pour le langage CO-OPN est définie de façon inductive, ce qui reflète la structuration hiérarchique des *context modules*. Nous proposons également une méthode pour la composition de composants CO-OPN (objets ou *context modules*) inspirée des travaux de Huerzeler [6]. Un des résultats de cette recherche a été une compréhension profonde de la sémantique du langage CO-OPN, ce qui nous a mené à proposer une nouvelle définition de *classe d'équivalence* pour la réduction des jeux de tests exhaustifs produit à partir des *intentions de test*.

SATEL

SATEL est une solution concrète au concept d'*intention de test*. Le langage est adapté au domaine du test et permet d'exprimer des contraintes sur les variables qui représentent les différentes dimensions dun *cas de test*, c'est-à-dire : les *chemins*, les *entrées*, les *sorties* et les paramètres de ces *entrées* et *sorties*. La construction de SATEL se base sur un raffinement du langage de contraintes proposé par Péraire et Buffo. En particulier nous proposons de nouveaux méchanismes permettant la construction modulaire d'*intentions de test* et leur composition permettant d'envisager le test *unitaire*, d'*intégration* ou *système* en utilisant le même formalisme.

En ce qui concerne la *décomposition en sous-domaines* développé par Péraire et Buffo, nous l'avons révisée et proposons un nouveau critère pour le calcul de classes d'équivalence qui prend en considération la structure hiérarchique d'une spécification CO-OPN. Les classes d'équivalence se calculent comme suit : un évènement e d'un *context module* est dans la même classe d'équivalence qu'un évènement e' si et seulement si les axiomes qui définissent le comportement des composants impliqués dans le tir de e, sont exactement les mêmes que ceux impliqués dans le tir de e'. En d'autres mots, un cas de test t incluant e ne trouve pas plus d'erreurs qu'un cas de test t' qui diffère de t seulement en remplaçant e par e'.

Grâce aux réseaux de Petri et de leur modélisation implicite de la concurrence, SATEL permet également la génération de jeux de test pour des systèmes concurrentiels. Nous tirons parti de cet avantage pour permettre au testeur d'exprimer des entrées simultanées dans les intentions de test et en en calculant les *oracles* associés.

Nous avons développé un éditeur pour le langage SATEL qui est intégré avec l'outil *CoopnBuilder* [7]. Cet outil précède nos travaux et permet l'édition, la simulation et la génération de prototypes pour des spécifications CO-OPN.

Sémantique de SATEL

Notre recherche nous a amené à proposer une sémantique complète pour le langage d'*intentions de test* SATEL. En nous inspirant de la sémantique de CO-OPN nous avons employé une approche algébrique en ce qui concerne les types de données introduits par SATEL. La *décomposition en sous-domaines* et la validation des cas de test extraits d'une *intention de test* — le calcul des *oracles* — sont exécutés à travers l'utilisation de la sémantique de *systèmes de transitions* de CO-OPN. En particulier notre solution se base sur l'annotation du *système de transitions* qui représente la sémantique d'une spécification CO-OPN.

Chapter 1

Overview

Let us start by defining the computer science domain our work is concerned with. In recent years we have observed an increasing interest of the computer science community in the development of *modeling* languages and associated methodologies that enable the efficient and cost-effective development of quality software. This interest stems from the conjunction of several aspects: on the one hand the electronics engineering has long reached a level of maturity that allows the duplication of computing power (processing, storage) every 18 months, as predicted by Moore [8] in 1965; on the other hand programming languages and compilers seem also to have reached maturity. The later comes somewhat as a disappointment as it has been long believed that the solution to the *software crisis* — which has been well described by Brooks in [9] — relies on a programming language which would encompass a number of constructs that would implicitly lead to massive productivity increase in software building. In the paper "No Silver Bullet" [10] published in 1987 Brooks claims that tackling *accidental complexity*, which comes from technical difficulties induced by the tools used in software construction, was mainly achieved by the advent of *structured programming* instigated by Dijkstra in [11]. Brooks distinguishes *accidental complexity* from *essential complexity* and claims that the latter is induced by the real-world domain tackled by the software system being developed — in that sense improvements in programming language technology cannot on their own massively improve the productivity in software engineering. This view is disputed by other authors such as Meyer who have proposed several extensions to the Object-Oriented paradigm (in particular to Eiffel [12]) in order to address the quality of software in general.

It is now generally accepted that the Object-Oriented programming paradigm failed to meet the original "Silver Bullet" expectations. It has nonetheless sparked an enthusiasm in the software engineering community in the development of tools and methodologies to address the modeling — using an Object-Oriented approach

— of automatable real-world domains. The purpose of this activity is to address the *essential complexity* as described by Brooks, rather than the *accidental complexity*. The Unified Modeling Language (UML) [13] and associated Rational Unified Process (RUP) [14] methodology seem to have been a clear step in this sense, as it promoted among practitioners the habitude of seeing the design of software as a series of refinement steps: In the first step the *model* is expressed in a semi-formal language which suitably expresses concepts of the real-world domain; in the last step the *model* has been transformed in a computer program. This trend seems to be currently further refined by the concepts of Model Driven Architecture (MDA) [15] and Domain Specific Languages (DSL) [16].

Modeling is however not a new discipline in software engineering and the UML itself is composed of many different modeling languages which have been developed separately by different researchers. Typically these modeling languages have a formal mathematical basis, allowing the definition of classes of models that share a common amount of properties. These common properties can be both seen as *useful* and *constraining*. They are *useful* in the sense that the properties normally imply that some syntactic or even semantic (based on the structure) checks are decidable and can thus be implemented in tools. Also, depending on how operational the semantics of the modeling language is, interpreters or compilers for the produced models may be automatically (or semi-automatically) generated. The *constraining* aspect of using formal modeling languages comes from the fact that the degree of freedom (or expressivity) for representing real-world domains is necessarily limited by both the syntax and semantics of the used formal modeling language.

The *formal methods* community has also proposed several solutions to the problem of effectively modeling real-world domains. Tools and associated methodologies such as the Vienna Development Method (VDM) [17] or the B-Method [18] have been successful especially in the areas of critical systems where quality needs to be guaranteed. Both methods are based on formal languages having formal semantics. The methodology consists of consecutively refining the specifications in a controlled fashion, i.e. by filling in missing parts while making sure the refined specification keeps a number of properties defined in the original specification.

The *formal methods* community has often been criticized by the *software engineering* community given that the precision of the involved languages and methods often involve mastering mathematical notations and precise formal logics. Despite the efforts of certain authors such as Dijkstra (e.g. in [19]) or Harel (e.g. in [20]), formal methods go on being regarded by practitioners as "too complicated" to perform software modeling tasks except for domains such as health, aeronautics and in general domains for which critical systems are necessary. This criticism is not completely unfounded as in fact formal methods are often too rigid to be directly applied to problems of the real world, or require vast experience from the modeler to be applied in an efficient manner.

In spite of the caveats presented by formal methods, they present sound methodologies that induce quality software. Moreover, formal models lend themselves to formal *verification* activities, such as automatic proofs of correctness or *Model Checking*. *Verification* can be seen as an orthogonal activity to *Modeling* and consists in checking that a *model* is consistent and/or that the *implementation* produced from a given *model* does in fact implement what the *model* defines. Differences between the *model* and the *implementation* may be induced by the *model* refinement steps where human intervention is necessary, or by the interfacing of the implemented software with its environment, e.g. *operating systems*, *external libraries*, *networks*.

The *software engineering* community has also proposed approaches to the *verification* problem, which is maybe even more important when semi-formal modeling languages are used. Semi-formal modeling languages are less precise than formal languages both in syntactic and semantic terms, giving rise to a higher probability of appearance of discrepancies between the model and the *implementation*. In the extreme case no explicit *model* is produced and the *implementation* is directly coded — the *model* is then a purely intellectual artifact existing in the mind of the engineer charged with the coding of the application. In these cases formal verification is in general not possible and the standard solution is *testing*. *Testing* is by nature a non-exhaustive activity and the most we can expect is to be able to find errors. As Myers writes in his seminal work "*The Art of Software Testing*" [21]:

"Testing is the process of executing a program with the intent of finding errors."

With *testing* we aim at "*providing evidence that the behavior of an* implementation *does or does not conform to its intended behavior*" [22]. After having *tested* an *implementation* we are probably more confident that it performs as expected, but never sure that it is error free. *Testing* thus contrasts with the concept of *verification* as in general seen by the *formal methods* community, where the goal is to *prove* that the *model* or the *implementation* satisfy a given property. With *testing* we cannot in the general case *prove* that the *implementation* is correct regarding a given *model* — we can only increase our confidence in that fact by performing a limited number of *experiments* which will most likely find discrepancies between the *model* and the *implementation*: the so-called *errors*.

The present thesis is thus concerned with the problem of *functional testing* of *reactive systems* in a *formal methods* context. Let us precise these notions:

- *functional testing* sees the *implementation* as a black box, i.e we can only observe how it reacts to a given stimulus and we do not know what internal mechanism produced a particular reaction. Also, only *functional* aspects of the *implementation* are considered and we discard non-functional concerns (e.g. performance, robustness, security) — unless non-functional concerns are somehow expressed in the *model* in a functional fashion;

- *reactive systems* are state-based systems which operate in an infinite loop, accepting inputs from an *environment* and producing responses to that same *environment*;
- The usage of *formal methods* refers to the fact that we use as reference to generate *test cases* a *model* expressed in a formal language. This contrasts with typical approaches in software engineering where *test cases* are produced by an engineer who is versed in the functionality of the *implementation* — with or without the help of an explicit *model* which is typically expressed in some semi-formal language.

Our scientific domain of interest is thus *functional testing* using as reference a *formal model* — also called *model-based testing*. In particular we are interested in understanding how the use of *formal methods* can help us to: 1) *automatically* produce quality *test cases*; 2) relate the notion of selecting a finite number of *test cases* (or, in the previously used vocabulary, "experiments") to the notion of *proving* the *correctness* of an *implementation* regarding a *model*.

1.1 Motivation and Objectives

As mentioned in the previous section, we are interested both in *automatically* building test cases and in investigating the relation between *proving* and *testing*. Let us start by the former.

We will begin by analyzing what can be automated in the production of test cases given a formal model of the SUT. Firstly it is clear that we have access to the *signature*[1] of the model, i.e. the names of the *inputs* the model accepts from the environment and the names of the *outputs* the model produces to the environment. The signature can be used as the basic vocabulary for a grammar that can produce *test cases*. Unfortunately the *test cases* produced in this fashion are of little use if the vocabulary of *inputs* and *outputs* is very vast, given that it will be difficult to find *test cases* which are meaningful — in particular which represent expected behaviors. In fact, despite the fact it can easily produce *test cases*, this technique does not allow us to automatically decide if, when experimented on the SUT, those

[1]In this work we assume there is a bijective function mapping specification *inputs* into SUT *inputs* and specification *outputs* into SUT *outputs*.

test cases uncover errors. This corresponds to what is called in the testing literature the *oracle* problem [23].

Also, in typical reactive systems the relation between *inputs* and *outputs* depends on the current state of the SUT. For all these reasons it becomes necessary to take into consideration in the model how the internal state of the SUT evolves in order to generate *oracles* for the produced *test cases*.

These issues have been previously investigated in the PhD. theses of Péraire [4] and Barbey [5]. The work described in both theses (which are complementary), applies the BGM (Bernot, Gaudel, Marre) [3] testing theory and tools to the context of the CO-OPN (Concurrent Object-Oriented Petri Nets) [1, 2] formalism. The BGM theory discusses a model-based framework for producing *test cases*. The theory can be seen as an instance of the work of Bernot [24] where the modeling language are *algebraic specifications*. *Test sets* produced using the BGM framework keep two essential properties: *Unbiasedness*, meaning a *test set* does not detect falsely errors in a correct SUT; *Validity*, meaning a *test set* uncovers *all* errors in an SUT. *Test cases* are produced by reducing an initial fictitious *exhaustive test set* by stating *testing hypothesis* about the implementation of the SUT. These *testing hypothesis* are generalizations about the correct implementation of certain functional features. However, the *unbiasedness* and *validity* properties are only kept if the SUT satisfies those *testing hypothesis*.

Let us precise that in the BGM theory a *test case* is *statically* produced with an implicit *oracle* — as opposed to *dynamic testing* where the *oracles* are calculated during the application of a *test case* to the SUT. Given that our work is based on the BGM theory, from here on in this thesis we will use the term *test case* to mean *test case* and its respective *oracle*, unless explicitly mentioned otherwise.

The novelty of the work of Péraire and Barbey resides in the fact that, while the original BGM theory was designed around *algebraic specifications* [25], CO-OPN relies on *algebraic nets* [26] as its base formalism. CO-OPN also includes Object-Oriented features encompassing dynamic object instantiation. The *algebraic specifications* formalism is suited to the description of stateless SUTs where the *outputs* are calculated as a deterministic function of the *input*. CO-OPN, on the other hand, models the state explicitly through the usage of Petri Nets [27, 28, 29] — as a way of specifying the behavior of CO-OPN objects — and is better adapted to the description of *reactive systems*. This difference implies a different fashion of producing *test cases*.

In the context of *algebraic specifications*, Marre has developed in his PhD. thesis [30] an operational method for automatically reducing the number of *test cases* in the exhaustive *test set*. The method can be seen as the automatic extraction of *testing hypothesis* from the model. It consists of applying an *unfolding strategy* to the axioms that describe the behavior of the operations in the *algebraic specification*. This strategy allows to operationally pick a small amount of *test cases*, notably by translating the specification axioms into Prolog programs and using controlled resolution to select interesting values for the variables present in the axioms. In what concerns CO-OPN specifications, Péraire and Barbey describe how to calculate *test cases* reusing Marre's technique. This is achieved by: considering state evolution by analyzing the semantic rules that define message passing between CO-OPN objects; applying an *unfolding strategy* to the algebraic conditions present in the axioms that allow transition firing in CO-OPN.

Marre has also proposed in his work on test generation from *algebraic specifications* a language to express *testing hypotheses*. This is achieved by using predicates to constrain the domains of the variables present in an *algebraic specification* axiom — thus implicitly stating that *test cases* generated for the axiom using values within the restricted domains are enough to test all the SUT behaviors coded in the axiom. Also, the previously mentioned *unfolding strategy* can be seen as a particular kind of hypothesis, which makes use of the structure of the *algebraic specification* axioms to automatically find values for the axiom's variables. Clearly, the quality of the produced *test cases* using this language is dependent on the capability of the test engineer to connect the constraining of the domains of the axiom variables to the testing of the real functional features in the SUT. In other words, the quality of the *test cases* depends on the capability of test engineer to express *true* hypothesis about the SUT using the test language. One of the main advantages of the work of Marre is that operational strategies for generating *test cases* using the test language and the specification can be efficiently implemented, as described in [31].

As for *algebraic specifications*, Péraire and Barbey have also introduced a test language for the specification of *testing hypothesis* in the context of CO-OPN specifications. This language was deeply influenced by the work of Marre, although the state-based property of CO-OPN models implied adopting a different approach. In particular, the behavior of the objects that form a CO-OPN specification is determined by axioms which essentially define the *pre-* and a *post-conditions* of a *public* guarded Petri-Net transition. By *public* we mean that the transition can only occur if synchronized with an event produced by the environment, which is a similar notion to what can be found in OO languages such as C++ or Java. From an observational point of view a *test case* for a CO-OPN specification can then be seen as a sequence of these synchronized transitions, which are called *method calls*. *Method calls* can be parameterized with algebraic values. The testing language in this new context is also based on constraining variables, but this time the variables represent more than only algebraic values as in *algebraic specifications*. They represent *algebraic values* which are parameters of *method calls*, *method calls* themselves or sequences of method calls, which allow testing state evolution. Péraire and Barbey defined in their work a new set of predicates to constrain these new domains, as well as a set of operational strategies that allow their implementation.

As a ***first objective*** of this thesis we are interested in building a new version of the *test language* for CO-OPN specifications. We aim at introducing in the language additional mechanisms that allow seeing the test production activity not as a reduction from an initial *exhaustive test set*, but rather as a construction by sequentially adding the test of different functionalities while trying to cover all the specified functionality — much as a test engineer would proceed while testing a real world SUT. This implies major changes given that the first version of the *test language* is deeply influenced by the fictive notion of *exhaustive test set*, which the *testing hypothesis* reduce to a finite and usable *test set*. We aim nonetheless at understanding under which conditions the properties of *unbiasedness* and *validity* are kept under such an approach.

As a ***second objective*** of this thesis we are interested in improving the *unfolding technique* applied to CO-OPN specifications. In the previous work no formalization of this aspect of the *test case* generation was produced, although an operational prototype of the method was implemented as described in [4]. This objective is also motivated by the fact that the CO-OPN language has evolved in the meantime — in particular the appearance of a new mechanism for modeling distribution allows the description of better structured models. We are thus interested in changing the original informally defined *unfolding technique* applied to CO-OPN to a formally defined method, having an emphasis on the state and structural characteristics of the new version of the CO-OPN language.

1.2 The CO-OPN specification language

CO-OPN (Concurrent Object-Oriented Petri Nets) [1, 2] is a specification language based on algebraic nets [26] but extending it Object-Oriented with and *distribution* features. Given that CO-OPN is the chosen specification language for our studies on model-based testing, we will provide in this section a brief introduction to the language by means of an example. In particular we will present the most recent version of the language whose semantics have been extensively modified by our work.

1.2.1 The Drink Vending Machine Example

Let us specify a simple Drink Vending Machine (DVM) controller using the CO-OPN specification language. The DVM will distribute several kinds of drinks, each one

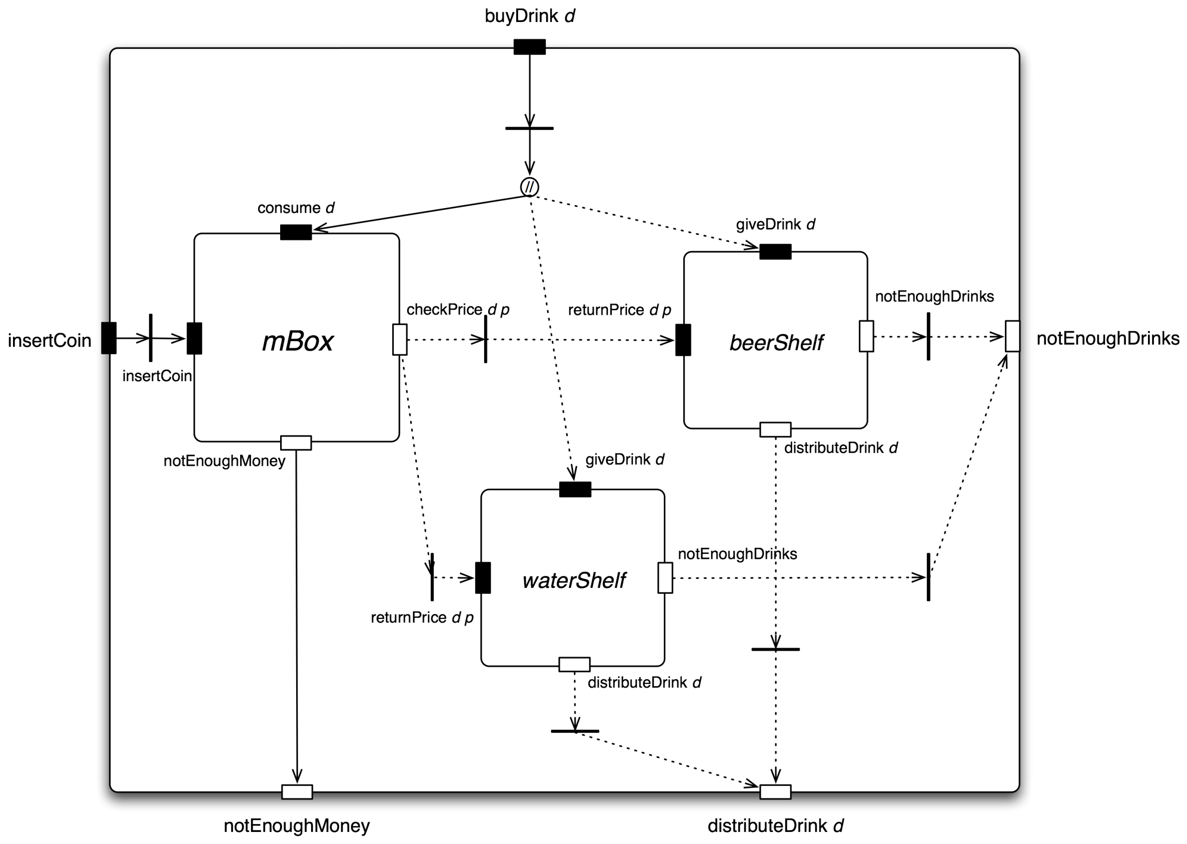


Figure 1.1: Drink Vending Machine — graphical syntax

having a certain price. As payment the DVM accepts unitary coins. The operation of the DVM can be resumed as follows: the user inserts a number of coins and selects a drink. If the number of inserted coins is enough then the drink is distributed. Otherwise the DVM issues a message to the user warning that the payment is not sufficient. It can also happen that there are no more drinks of the selected kind left, in which case the DVM will also issue an appropriate warning.

The CO-OPN language has both a graphical and textual syntax. Figure 1.1 presents an incomplete version of the DVM specification using the graphical syntax. The figure depicts the structure of the system, which includes: an object *moneyBox* responsible for controlling coin acquisition and checking if the inserted money is enough for a given drink; two *beerShelf* and *waterShelf* objects which are responsible for holding the number of available beer and water drinks and the unitary price. More *Shelf* objects containing other kinds of drinks could be envisaged. In the figure we purposefully left out the detail of the Petri Nets encapsulated by the objects.

The outer box is called a *context* in CO-OPN and is responsible for the coordination of the three objects we mentioned in the above paragraph. Coordination is achieved by synchronizing *ports* — the black and white rectangles in the borders of the boxes, called *methods* and *gates* respectively — *requiring* a service with ports *providing* a service. At the level of the CO-OPN objects *methods* and *gates* are implicitly synchronized with Petri Nets transitions. A *method call* or a *gate call* is then a predicate corresponding to the firing of one or several Petri Net transitions. From a more operational point of view *method calls* can be seen as *inputs* and *gate calls* as *outputs.*

Let us illustrate the coordination capabilities of a *context* by exposing how the '*buyDrink* _' operation is computed in the model. Assume a user has inserted a sufficient amount of coins to buy a *water*. The '*buyDrink water*' *method call* is activated on the *context* and it synchronizes simultaneously (as specified by the '//' synchronization operator) with the '*consume water*' and the '*givedrink water*' (in the *waterShelf object*) *method calls.* The detail of the synchronization can be observed in figure 1.4 representing the textual syntax of the DVM *context.* In particular in line 7 the axiom describes that synchronization where the *required* and the *provided* services are separated by the ***with*** keyword. Notice also that: the usage of the *d* variable allows parameter passing; the *shelf* variable allows synchronizing simultaneously with all objects which are instances of the *DrinkShelf* class, without having to explicitly state additional synchronizations.

Also simultaneously the '*checkPrice water p*' *gate call* synchronizes with the '*checkPrice water p*' (in the *waterShelf object*) *method call* — this allows calculating the price of the *water* drink which is contained in *p* and which is known by the *waterShelf* object — see also the axiom in line 6 of figure 1.4. Still simultaneously the number of coins in the *moneyBox* will be decreased by the price of the *water* drink and the number of units of *water* in the *waterShelf* object will be decreased. This is specified in textual syntax respectively by the axioms in lines 4-5 of figure 1.2 and in lines 6-8 of figure 1.3. Briefly, the syntax of a class axiom is separated into a *condition*, the *event* associated to the operation, a *pre-condition* marking in the object's Petri-Net and a *post-condition* marking.

Finally, simultaneously with all of the previous, a synchronization between the '*distributeDrink water*' *gate call* in the *waterShelf* object and the '*distributeDrink water*' *gate call* in the *context* is produced— see line 8 of figure 1.4 — thus providing the order to the physical machine to distribute the real drink.

Notice that we have described all the actions that lead to the distribution of the *water* drink as being simultaneous. This is due on the one hand to the fact that we only use the *simultaneity* synchronization operator '//' — others exist, notably the *sequence* and the *alternative* synchronization operators. On the other hand, due to CO-OPN semantics, all enabled transitions in the structured Petri Net representing the DVM fire simultaneously by default. We can then see the '*buyDrink water*' *method call* as a predicate, which is allowed at the current state of the DVM (enough tokens were inserted to buy *water*) and happens ***with*** the '*distributeDrink water*' *gate call.*

```
Place
    coinHolder _ : natural;

Axioms
    (am >= p) = true =>
        consume d With checkPrice d p :: coinHolder am -> coinHolder am - p;

Where
    am : natural;
    p : natural;
    d : drink;
```

Figure 1.2: The MoneyBox Class

```
Places
    availableUnits _ : natural;
    name _ : drink;
    price _ : natural;

Axioms
    (units > 0) = true =>
        giveDrink d :: availableUnits units, name d ->
        availableUnits units - 1, name d;
    returnPrice d p :: name d, price p -> name d, price p;

Where
    units : natural;
    p : natural;
    d : drink;
```

Figure 1.3: The DrinkShelf Class

In figures 1.2, 1.3 and 1.4 we have presented incomplete versions of the DVM specification in CO-OPN's textual syntax. The full version can be found in appendix A.

1.2.2 Formal aspects of CO-OPN

We will now provide a brief overview of the fashion in which the CO-OPN language was defined. Given the recent changes in the language (defined by Buffo in [2]) it became necessary to fully revise its formal definition in order to produce a semantics for our *test intention* language. Buffo has defined the semantics of the latest

```
Objects
    beerShelf : drinkshelf;
    waterShelf : drinkshelf;
    mBox : moneybox;

Axioms
    mBox . checkPrice (d, p) With shelf . returnPrice (d, p);
    buyDrink d With mBox . consume d // shelf . giveDrink d;
    shelf . distributeDrink d With distributeDrink d;

Where
    shelf : drinkshelf;
    d : drink;
    p : natural;
```

Figure 1.4: The DVM Context

version of CO-OPN (CO-OPN$_{/2c}$) in a transformational manner, by converting a CO-OPN$_{/2c}$ specification into a CO-OPN$_{/2}$ specification — whose semantics was defined by Biberstein in [1]. We provide with our work a direct semantics, which, in the course of our research, allowed devising a clear fashion of extracting *test cases*. This new method of *test case* generation is distinct from the one provided by Péraire and Barbey in [4, 5] and more adapted to CO-OPN$_{/2c}$ specifications. We have thus introduced with our work the CO-OPN$_{/2c++}$ version. The formal definition of CO-OPN$_{/2c++}$ follows the typical definition of formal languages — we start by providing an *abstract syntax* which we then exploit to build the language's *semantics*.

The fashion in which the *abstract syntax* is defined is inspired by the syntax of *algebraic specifications*. In these kind of specifications we start by stating a *signature*, including a set of *sort* names and *operation* names with their respective *arities*. An *algebraic specification* corresponds to a *signature*, a set of *variables* and a set of *axioms* which are equalities between terms of the same sort — formed using the *signature* and the *variables*.

CO-OPN$_{/2c++}$ includes three kinds of modules — ADT, *Class* and *Context* modules, which have been informally introduced in section 1.2. The definition of the abstract syntax of all the modules follows the strategy of definition of *algebraic specifications*. We start by defining a *signature* — called *interface* in the case of *class* and *context* modules — and then proceed to the definition of the *modules* themselves. The definitions of ADT *signatures* and *Class interfaces* are intimately connected and we define a *global signature* including both. The *global signature* allows building in the subsequent development a unified semantics for the simultaneous management of data values and *object identifiers*.

At the level of the *abstract syntax* of the modules themselves we introduce the concept of *behavioral formulas* for *class modules* and *coordination formulas* for *context modules*. These formulas correspond to the notion of *axioms* in *algebraic* *specifications* but are fundamentally different in what they describe — *behavioral formulas* correspond to the *conditions* under which a given transition in a Petri Net can fire and *coordination formulas* correspond to the fashion in which the several modules that compose a CO-OPN$_{/2c++}$ communicate. Finally, the *abstract syntax* of a CO-OPN$_{/2c++}$ specification is defined as a set of ADT, *Class* and *Context* modules which use the same set of *signatures* and *interfaces*. A CO-OPN$_{/2c++}$ specification satisfies two properties of acyclicity in the dependency graphs of the modules composing the specification. These properties are necessary in order to allow the existence of a semantics for the specification. A more complex semantics allowing cyclic dependency graphs in the specification was defined by Buffo in [32].

CO-OPN$_{/2c++}$ has a Labeled Transition System (LTS) semantics which we calculate in an inductive fashion, following the dependency graph established by the *context modules* in the specification. The *inductive base case* corresponds to calculating the semantics of a *context module* which is either empty or only coordinates *objects*. The *inductive step* corresponds to calculating the semantics of a *context module*, assuming that the semantics of the *context modules* it coordinates is known.

In order to calculate the *inductive step* a series of operations is necessary. Firstly we produce the LTS semantics of the set of *classes* which objects are coordinated by the *context module*. This is done using a set of *inference rules* defined much in the style of the Structural Operational Semantics framework described by Plotkin in [33]. These inference rules directly take into consideration *dynamicity*, i.e. object *creation* and *destruction*. We then compose the LTS semantics of *classes* with the LTS semantics of the coordinated *context modules* — which is assumed known — by a series of LTS composition functions. In particular, the calculation of the LTS resulting from the *coordination* of all the components inside a *context module* is achieved using a technique introduced by Huerzeler in [6]. The last step of the *inductive* step corresponds to *filtering* the LTS resulting from the previous steps using the *context module's* interface.

1.3 The SATEL Test Intention language

SATEL (Semi-Automatic Testing Language) is a language we have developed in the context of this thesis in order to express *test intentions* for CO-OPN$_{/2c++}$ specifications. *Test intentions* correspond roughly to what has been described in the literature as *test purposes* [34]. Generally speaking, a *test intention* allows choosing a behavior of the SUT and producing for it a '*reasonable*' or *'practicable'* amount of *test cases* — using the CO-OPN$_{/2c++}$ model as reference. The choice of the particular part of the SUT the test engineer wishes to test is done by expressing constraints

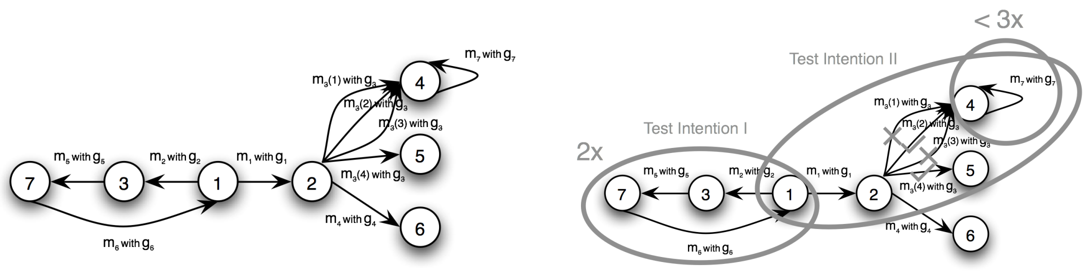


Figure 1.5: a) Fictitious SUT (left); b) fictitious SUT covered by *test intentions* (right)

about syntactic domains representing several dimensions of the SUT's behavior — as inspired by the work of Bernot, Gaudel and Marre [3]. These syntactic dimensions are extracted from the *model's signature* and consist of:

- the shape of the execution paths;
- the kind of input/output pairs inside a path;
- the parameters of the inputs and outputs.

For the purpose of illustration of the concept of *test intention*, figure 1.5a) displays a transition system representing the semantics of a fictitious SUT. The labels in the arcs of the transition system correspond to events of the SUT, composed of an *input* and an *output* part and split by the ***with*** keyword. Inputs having a shape $m(p)$ represent an operation m having a parameter of type p — as opposed to inputs having a shape m where no parameters exist.

In figure 1.5b) we depict a possible coverage of that transition system by test intentions expressed in SATEL. The coverage which would be formally defined in SATEL is informally expressed in the figure by: ellipses representing a covered path; ellipse annotation representing the number of times that path should by tried in the produced *test cases*; 'v' or 'x' over an LTS transition if the transition is respectively chosen or not as part of the produced test cases. *Test intention I* produces a *test case* involving two sequential activations of inputs 'm_2', 'm_5' and 'm_6'. *Test intention II* is more complex and produces four *test cases*. Three of those *test cases* correspond a to an activation of input 'm_1', followed by an activation of input '$m_3(2)$' and by a number of of activations of input 'm_7' any number of times inferior to three. The fourth *test case* consists of the activation of input 'm_1' followed by the activation of input '$m_3(4)$'.

Note that the test intention II chooses only one parameter from the set $\{1, 2, 3\}$ for input m_3. Given that transitions '$m_3(1)\, with\, g_3$', '$m_3(2)\, with\, g_3$' and

'$m_3(3)\,with\,g_3$' lead to the same state, a *test case* involving any of them would ensure the behavior for any transition in this group is tested. SATEL allows expressing these kind of 1-to-n hypotheses about the correctness of the SUT by using either "manual" constraints on data types — in *test intention II* we would restrain the values of the parameter of transition 'm_3' to set $\{2, 4\}$ — or by using automatic mechanisms for choosing one parameter per possible behavior of input 'm_3'. These automatic mechanisms are based on the analysis of the behavioral axioms present in a CO-OPN model of the SUT and which describe the conditions necessary for each input to activate.

SATEL is fully integrated with CO-OPN as a language and in a toolkit, allowing the creation of test intentions as part of the modeling effort. This integration allows us to: access the signatures of *inputs*, *outputs* and *data types* in order to syntactically construct *test cases*; access the semantics of the model of the SUT in order to build *oracles* for our *test cases*.

1.3.1 Test Intentions for the Drink Vending Machine

In figure 1.6 we introduce a very simple test intention named '*ThreeEvents*'. '*ThreeEvents*' is defined inside the test intention module '*TestBanking*' and uses a '*path*' variable having as domain the *execution paths* of the DVM context module (see figure 1.1). The *nbEvents* function allows measuring the number of events (input/output pairs) contained in an *execution path* — thus the '*nbEvents(path)<=3*' condition reduces the the domain of '*path*' to execution paths including less than three events. The idea behind this test intention — admittedly not very sophisticated — is thus to generate tests that cover at most three interactions with the SUT.

```
TestIntentionSet TestBanking Focus DVM;
   Interface
      Intentions
          ThreeEvents;

   Body
      Axioms
          (nbEvents(path) <= 3)  =>   path in ThreeEvents;

Variables
     path : primitiveHML;

End TestBanking;
```

Figure 1.6: Very Simple Test Intention

For the purpose of demonstrating our approach, let us assume our DVM CO-

OPN model was initialized with 5 shelves including the following drinks: water bottles costing 1CHF each; soda cans costing 1CHF each; beer cans costing 2CHF each; milk cartons costing 2CHF each; apple juice cartons costing 3CHF. We present in figure 1.7 possible test cases produced by the '*threeEvents*' test intention in figure 1.6.

```
HML( {insertCoin with null} {insertCoin with null} {buyDrink(water) with
    distributeDrink(water)} T), True
HML( {buyDrink(beer) with distributeDrink(water)} {insertCoin with null} T), False
HML( {buyDrink(beer) with null} T), False
```

Figure 1.7: Non -exhaustive test set generated by the '*ThreeEvents*' test intention

Notice that the last two *test cases* are annotated with the *false* logic value. This is because that particular sequence of operations is an invalid behavior according to the specification — the domain of variable *path* includes any path that can be generated from the signature of the DVM context, independently from the fact that it corresponds to a *valid* or *invalid* behavior of the SUT. The *test set* presented in figure 1.7 is not complete as many other test cases holding three or less events would be possible. Notice that we use the *HML* and *T* keywords both while defining *test intentions* and in the syntax of *test cases*. This is due to the fact that our test language is the HML (Hennessy-Milner) temporal logic [35]. Also, the '*null*' keyword represents the absence of observation in an event.

Summarizing, a test intention is formally written as a set of partially instantiated HML formulas where the variables present in those formula are by default universally quantified. All the combined instantiations of the variables will produce a (possibly infinite) number of test cases.

Each test intention may be given by several rules, each rule having the form:

```
[ condition => ] inclusion
```

In the *condition* part of the rule it is possible to define constraints over the variables present in the HML formulas that make up the test. The *inclusion* part is comprised of the HML formulas with variables and a name for the test intention defined by that rule.

SATEL allows guiding test generation in a more precise fashion than the "brute force" approach in the '*ThreeEvents*' test intention. We define two additional test intentions in figure 1.8. The '*RepeatInsertCoin*' *test intention* produces a set of test cases that correspond to sequences of any size of '*insertCoin*' operations. Notice that in order to state this test intention we use an inductive definition having as

```
TestIntentionSet TestBanking Focus DVM;
   Interface
      Intentions
          RepeatInsertCoin;
          SellDrink;

   Body
      Axioms
          T in RepeatInsertCoin;

          path in RepeatInsertCoin =>
              path . <insertCoin with null> in RepeatInsertCoin;

          subUniformity(drink) | path in RepeatInsertCoin, nbEvents(path) <= 3 =>
              path . <buyDrink(drink) with buyOutput> in SellDrink;

      Variables:
         path : primitiveHML;
         buyOutput : observation;
         drink : Drink;

End TestBanking;
```

Figure 1.8: Very Simple Test Intention

base case the empty HML formula (noted '*T*') and as step the concatenation[2] of an '*insertCoin*' operation. This test intention cannot be used directly — given that it produces an infinity of test cases — but is rather used indirectly in the '*SellDrink*' test intention that builds test cases for testing the '*buyDrink*' operation of the DVM. The '*SellDrink*' test intention is built by starting with a number of '*insertCoin*' operations inferior to three, followed by a '*buyDrink*' operation. Notice that the '*drink*' variable is constrained with the *subUniformity* predicate, meaning one value should be chosen for '*drink*' per behavior of the operation. Also, the '*buyOutput*' variable allows several instantiations resulting from the several behaviors chosen by the *subUniformity* predicate. Possible *positive* test cases produced by the *SellDrink* test intention would include the following:

```
0 HML({buyDrink(water) with notEnoughMoney} T), True
  HML({insertCoin with null} {buyDrink(water) with distributeDrink(water)} T), True
2 HML({insertCoin with null} {buyDrink(beer) with notEnoughMoney} T), True
  HML({insertCoin with null} {insertCoin with null} {buyDrink(soda) with
      distributeDrink(soda)} T), True
4 HML({insertCoin with null} {insertCoin with null} {buyDrink(apple_juice) with
      notEnoughMoney} T), True
```

Figure 1.9: Test Cases for the *SellDrink* test intention

The test case in line 0 corresponds to the concatenation of zero *insertCoin* operations — which is the base case of the recursion of the *RepeatInsertCoin test*

[2]Concatenation between HML formulas is achieved using the "." operator.

intention — with one *buyDrink* operation. In this case only one *positive* behavior can be tested, the one where we try to buy a drink not having inserted sufficient coins. Test cases in lines 1 and 2 correspond to the insertion of one coin, followed by either buying a drink we can afford or one we cannot afford. The test cases in lines 3 and 4 are similar to the ones in lines 1 and 2, but following the insertion of 2 coins. The main point to retain from the test cases in figure 1.9 is that, for each number of inserted coins, the test generation mechanism only chooses one drink to test each possible behavior of the *buyDrink* operation.

1.3.2 Formal aspects of SATEL

In the context of this thesis we have built the formal syntax and semantics of the SATEL language. One of our main worries while doing so was to achieve a formal integration with CO-OPN$_{/2c++}$. This allows us not only a precise logical reasoning about the properties of SATEL, but also a seamless integration with the existing implemented CO-OPN tools [36].

At the syntactic level we have built SATEL as a an additional *test intention module* in the already existing CO-OPN$_{/2c++}$ specifications. The definition of the abstract syntax of a *test intention module* follows a slightly different order than the one we have used for the ADT, *Class* or *Context* modules. We start by building what we call *execution patterns* which correspond to expressions in a temporal logic (Hennessy-Milner Logic[3] [35]) built using the *signature* of the CO-OPN$_{/2c++}$ model of the SUT. The temporal aspect of the logic allows expressing the evolution of the state of the SUT. On the other hand the predicates at a given moment of time correspond to the fact of necessarily providing a given *input* to the SUT and observing a given output. *Execution patterns* also include variables over the already mentioned dimensions of an SUT's behavior.

We then build the abstract syntax of *test intention* axioms. A *test intention* axiom consists of both an *execution pattern* and a set of *constraints* over the variables present in that *execution pattern.* In order to introduce the *constraint* language we build the syntax of a set of functions and predicates that allow measuring and constraining the dimensions of an SUT's behavior.

As for the construction of ADT, *Class* or *Context* modules, we have also followed the syntactic style of *algebraic specifications* — we have defined an *interface* holding the basic vocabulary of the module, i.e. the *test intention* names. Given that a *test intention* module is written for a particular context module — called the *focus* of the *test intention* — in the *abstract syntax* we also include the interface of that *context module.* We include equally a number of *signatures* and *interfaces* of

[3]Very simple temporal logic including only the *next* modal operator.

ADT, *Class*, *Context* and *Test Intention* modules. These are necessary in order to access the signatures of the "imported" modules and that compose a CO-OPN$_{/2c++}$ specification. Note that, in particular, the usage of *test intention* interfaces enables the composition of *test intentions* spread over several *test intention modules.*

Finally, we build a new kind of specification which we call a *CO-OPN and SATEL* specification. This object unites the two languages under the same notation and allows integrating the *modeling* and the *verification* aspects in the same specification.

In what concerns the semantics of a *test intention*, we calculate it in a sequence of steps. Firstly we "expand" the axioms of the chosen test intention which will yield a set of *expanded execution patterns.* These *expanded execution patterns* resolve the mechanisms of *composition* and *recursion* in SATEL. Each *expanded execution pattern* has associated a set of *conditions* which are the reduction hypothesis present in the *test intention axioms.* The *exhaustive test set* for a *test intention* is then calculated by instantiating the variables in each *expanded execution pattern* according to its *conditions* and checking the satisfaction of the obtained HML formulas w.r.t. the specification's semantics.

We then reduce the *exhaustive test set* by finding its equivalence classes and picking one test case per equivalence class — when this has been specified by the test engineer in the *test intention.* The calculation of the equivalence classes is achieved by collecting all substitutions for the variables in a given *expanded execution pattern* and finding which *events* or *sequence of events* result from the same structural conditions in the CO-OPN specification.

In order to form *equivalence classes* we assume all the transitions in the LTS semantics of the CO-OPN model are annotated with the conditions of the *coordination* or *behavioral* formulas that allowed the corresponding firing(s) in the model. In particular this means all transitions departing from the same state and marked with the same annotation define an *equivalence class.* We are then able to select values for variables marked with the *subuniformity* predicate in the *test intention* definition by choosing one value per group of transitions annotated with the same conditions. Note this kind of *subuniformity* hypothesis is clearly different from the one proposed by Marre in [30] and reused by Péraire and Barbey in [4, 5]. While these authors focus mainly on an *unfolding* method which is directly connected to the fashion in which the axioms of *algebraic specification* are defined, we propose a method which is based on the structure of the specification. Given that CO-OPN$_{/2c++}$ specifications are hierarchical, the conditions annotating the LTS reflect this structure and provide a different fashion of defining equivalence classes — which is more adapted to structurally complex SUTs.

1.4 Contributions of this thesis

Although the main topic of the present thesis is the extraction of *test cases* from CO-OPN specifications, we have contributed at several other levels. Due to the fact that the CO-OPN language has been developed in several iterations with the goal of including additional expressiveness, the overall syntax and semantics was dispersed over several non-uniform pieces of work. We propose an integrated framework where we have completely rewritten the syntax and semantics of the language from the formal point of view.

The main contribution of this thesis is the notion of *test intention*, which we have concretized in the proposal of the SATEL language. *Test Intentions* borrow much from the work of Bernot, Gaudel and Marre, as well as from the work of Péraire and Buffo. Our proposal aims at using the fictive notion of exhaustive test set in order to *cover* only certain behaviors of the SUT, rather than reducing an initial test set covering the full behavior. In this fashion we adapt the BGM theory to the testing of SUTs with complex functionality and we aim at providing an engineer-oriented tool. We were particularly careful in the description of the SATEL language, adopting a detailed formal style of describing the syntax and semantics of the language. The purpose of the formal description is dual: on the one hand we provide a full syntactic and semantic integration of SATEL with the CO-OPN framework; on the other hand the formal description is sufficiently detailed to allow the fast implementation of an operational interpreter in logic programming.

1.4.1 CO-OPN

Our main work in the CO-OPN language consisted of providing an integrated semantics for the notions of *gate* and *context*. These notions were not present in the previous work on *test case* generation from CO-OPN specifications. Given that *context modules* induce a hierarchy in CO-OPN specifications, we have built the semantics in an inductive fashion which reflects the *context containment* hierarchy. Also, we propose a method for solving the composition of CO-OPN components (objects or context modules) which was inspired from the work of Huerzeler. One of the results of this work was the deep understanding of the semantics of CO-OPN, which led us to the definition of a new notion of *equivalence class* for the reduction of exhaustive test sets.

We have also devoted some effort to the unification of the abstract syntax of *context modules* with the remaining modules of CO-OPN. As a result we have obtained a methodology which allowed us to include without difficulty *test intention* modules in the existing CO-OPN framework.

1.4.2 SATEL

With SATEL we propose a concrete solution to the concept of *test intention*. SATEL is an expressive language for the domain of testing, allowing the constrained instantiation of variables representing the several dimensions of a *test case* — the covered *paths*, the *inputs*, the *outputs* and the possible parameters of those inputs/outputs. We have refined the constraint language previously proposed by Péraire and Buffo and we have proposed new mechanisms for modular construction of *test intentions* — by allowing building *test intentions* based on other *test intentions*. This compositional aspect of *test intentions* clearly goes in the sense of the engineering notions of *unit*, *integration* or *system* testing. In terms of *sub-domain* decomposition we have proposed in SATEL a criterion for equivalence class calculation which reflects the hierarchical structure of a CO-OPN specification. In particular, an event e of a CO-OPN *context module* will be in the same equivalence class as e' if the CO-OPN *object* or *context* axioms involved in the firing of e are exactly the same as the ones involved in the firing of e'. Extrapolating this property, a test case t including an event e will not find more errors than a test case *t'* which only differs of t by having e replaced by *e'*.

SATEL allows the production of tests for concurrent systems due to the implicitly concurrent nature of CO-OPN specifications. We have exploited this fact in SATEL by allowing the expression of simultaneous *inputs* and calculating the *oracles* for such tests.

Finally, we have developed an editor for SATEL which we have included in the CoopnBuilder Tool. The editor is fully integrated with the editor of CO-OPN specifications, as we have formally defined in this thesis. The results of this implementation are described in [37] and [38].

1.4.3 Semantics of SATEL

We propose a complete semantics for our test language SATEL. As for the semantics of CO-OPN, we have used an algebraic approach for the data types of SATEL. In order to provide a semantics to the *validation* (oracle calculation) of the test cases implied by a *test intention*, we have made use of the *transition system* semantics of CO-OPN. In particular we propose an elegant formal solution to the calculation of the equivalence classes for SATEL variables. Our solution is based in performing an annotation of the *transition system* representing the semantics of the CO-OPN specification.

1.5 Document structure

This document is organized as follows:

- Chapter 2 provides a first state of the art on Model-Based Testing. We present several approaches and extrapolate a framework of concepts in order to precisely define our approach;
- Chapter 3 is an brief and concise description of some basic formal tools we use throughout the thesis;
- In Chapter 4 we describe the history of CO-OPN, provide examples and introduce its abstract syntax;
- Chapter 5 is the complete description of CO-OPN's semantics;
- In Chapter 6 we provide a description of the BGM (Bernot, Gaudel and Marre) theory on testing and it's previous implementation in the context of CO-OPN. Chapter 6 presents a second, deeper state of the art and motivates the work we present in the subsequent chapters;
- Chapter 7 introduces SATEL, a *test intention* language for CO-OPN specifications. We introduce SATEL by means of examples, discuss its features and introduce an abstract syntax for the language;
- In Chapter 8 we describe the complete semantics of SATEL;
- Chapter 9 provides a case study based on an industrial collaboration;
- Chapter 10 presents our final considerations and directions for future work.

1.6 Introduced Notions

- **Formal Language**: a language with precisely defined syntax and semantics.
- **Semi-Formal Language**: a language including constructs whose semantics and/or syntax is implicitly or ambiguously defined.
- **Functional Testing**: the SUT is seen as a function and its structure is not taken into consideration while producing *test cases*;
- **Model-Based Testing**: a model is used as reference in order to produce *test cases* and oracles;

- **SUT**: System Under Test, also called *implementation*.
- **Model**: represents partially or completely (and possibly at a higher level of abstraction) the structure and the behavior of the software system we aim at testing. Also called *specification*.
- **Error**: a discrepancy between the *model* and the SUT.
- **Test Case**: a structure that allows describing an experiment that can be carried out on an SUT in order to try to find errors. Also called *experiment*. In the context of our work we assume the *test case* implicitly includes its *oracle*.
- **Oracle**: a procedure for deciding whether or not an SUT *passes* a given *test case*. If the SUT does not *pass* a *test case* then an *error* is found.
- **Test Set**: a set of *test cases*.

1.7 Summary

We start the chapter by providing an overview of the scientific domain of our work. In particular we discuss the different proposals by the *software engineering* and the *formal methods* communities in terms of *verification*. The software engineering community usually approaches verification by *testing*, while the formal methods community typically uses other kinds of techniques such as *automatic theorem proving* or *model checking*. Our work is concerned with both *testing* and *formal methods* — we wish to produce *functional test cases* using a model described in a *formal language*.

The motivation for our work lies in the desire to adapt the BGM theory of testing to the CO-OPN language and to a *test engineering* point of view. Previous work has been done by Péraire and Barbey on adapting BGM to CO-OPN, although the approach was more concerned with the formal correctness of the adaptation than with how ergonomic or practical the solution is for the *test engineer* or how it reflects the testing reality. We have thus decided to introduce a new language for *test selection* based on the most recent *gate* and *context* features of the CO-OPN language. We are also interested in improving the state of the art on the automatic aspect of test set *reduction* — which was previously done operationally in Prolog, using the *unfolding* technique initially developed for algebraic specifications. Finally, we want to understand in which measure a more *test engineer* oriented approach has an impact in the completeness of the BGM theory in the context of CO-OPN.

We then introduce the most recent version of the CO-OPN language which we have named CO-OPN$_{/2c++}$. CO-OPN$_{/2c++}$ has the same syntax as CO-OPN$_{/2c}$ but its semantics were defined in an entirely different fashion by our work. CO-OPN$_{/2c++}$'s concrete syntax is introduced by means of a Drink Vending Machine (DVM) example, followed by a brief description of its formal syntax and semantics. CO-OPN$_{/2c++}$'s formal syntax is composed of the definitions of *ADT*, *Class* and *Context* modules, written in the syntactic style of *algebraic specifications*. The dynamic aspects of the language are defined using *algebras* — for the semantics of *data types* and *object identifiers* — and *transition systems* — for the semantics of state evolution.

SATEL is the *test intention* language we introduce in this thesis. *Test intentions* adapt BGM test selection — using CO-OPN models — to the test engineer and to current practices of testing. The main idea behind the language is to perform test set reduction upon parts of the behavior of the SUT, rather than on the whole system simultaneously. We present SATEL by providing an example of *test intentions* on the DVM example and then describing SATEL's abstract syntax and semantics. The abstract syntax is similar to that of the other modules in the CO-OPN language. The semantics of SATEL includes an algebraic component for *data types* as well as the calculation of *oracles* and behavioral equivalence classes using annotated transition systems denoting the semantics of the considered CO-OPN model.

Finally we mention the main contributions of this document: a new semantics for CO-OPN specifications; the *test intention* language SATEL with complete semantics; a formal study on behavioral class equivalence in terms of CO-OPN specifications.

Chapter 2

Model-Based Testing State of the Art

As we have mentioned in chapter 1, the work we present in this thesis is mainly concerned with the *functional testing* of *reactive systems* in a *formal methods* context. In particular, we introduce the notion of *test intention. Test Intentions* allow picking from the set of all possible behaviors of an SUT a particular subset, as well as reducing it to *practicable* size. While writing a state of the art for the present thesis we are interested in analyzing the most relevant formal approaches to *model-based testing*, as well as work which is similar to our proposal of *test intentions.*

The present chapter is organized as follows: we start by identifying the artifacts underlying *model-based testing* and how they translate into the formal world. Our goal with this analysis is to establish a solid framework of concepts and vocabulary in which our work can be mapped onto. In a second stage we will use the framework of concepts we devised in the first step to classify some relevant formal approaches to *model-based testing.* Finally we will provide a survey on *test purposes* — which are similar to the notion of *test intention* — and establish their usefulness for automatic test selection and generation.

2.1 Formal Identification of the Actors in Model-Based Testing

In [39] Pretschner and Phillips present an account on methodological issues in model-based testing. An interesting contribution of the paper is that it provides a clear generalization of all the actors involved in model-based testing and a semi-formal framework in which, as far as our knowledge goes, all functional approaches to test

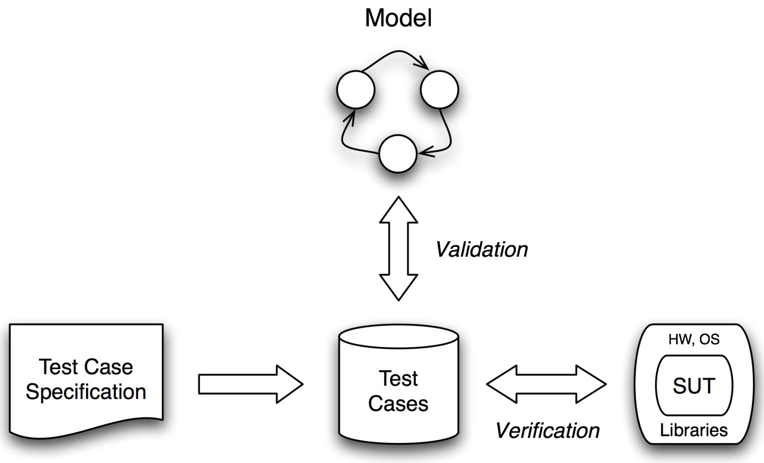


Figure 2.1: Actors involved in Model-Based Testing

case generation can be mapped onto.

The actors involved in model-based testing and identified by Pretschner and Phillips in [39] are depicted in figure 2.1. The figure is a simplified version of a schema introduced by these authors and introduces the following four artifacts: the *SUT* corresponds to the software we wish to test, which exists under a certain environment including hardware, operating system, communication network and auxiliary libraries; the *Model* is an abstraction of the SUT, used as the reference to produce *test cases* and their respective *oracles*; the *Test Case Specification* corresponds to some fashion of directing the activity of test generation in order to produce practicable test sets in a reasonable amount of time; *Test Cases* are the product of the *test case generation* activity.

Note that the arrow labeled *Validation* between the *Model* and the *Test Cases* indicates that the *Model* serves as a reference to generate *oracles* for *test cases* indicated by the *Test Case Specification.* The arrow is bidirectional given the fact that extracting *test cases* can help in validating the *Model* itself. The bidirectional arrow labeled *Verification* has a similar meaning: *test cases* allow verifying the SUT, but the verification activity is extended to the *test cases* themselves in the sense that a found error may incur from a badly produced *test case.*

Let us now identify from a formal point of view all the actors explicitly or implicitly stated in [39]. We do not provide their formal definition, but rather the fundamental characteristics their formalization should include and how they relate to each other.

- **Specification**: called *Model* in figure 2.1. Is formalized in a language allowing the expression of functional properties of the implementation. The formal language to express specifications may implicitly include mechanisms that directly yield information loss as a means of promoting abstraction from the SUT;

- **SUT**: is formalized in a language allowing the expression of functional properties of the implementation in an observational fashion. Note that while building a formal framework for *model-based testing* we differentiate an SUT from an *implementation*. While the implementation corresponds to the real software, the SUT is a formal object representing the *observation* of the functional behavior of the *implementation*;

- **Implementation Relation**: relation between a model given in the *formal specification language* and a model given in the *formal SUT language*. Note that due to the fact that the *formal specification language* can be the same, similar or very different from the *formal SUT language*, the implementation relation can range from very simple to very complex. Clearly, a complex implementation relation will impose more complexity on the test generation activity;

- **Test case**: formalized in a language sufficiently expressive to build tests sets that verify the property:

  $$\text{SUT } satisfies \text{ Test Set } \Rightarrow \text{ SUT } implements \text{ Model}$$

  Notice we wish to prove that the implementation relation holds between the model and the SUT. This is done indirectly by checking if the *satisfies* relation holds between *test set* derived from the *model* and the SUT. In practice the *satisfies* relation holds when the SUT *passes* all *test cases* in the considered *test set*;

- **Test derivation function**: takes as inputs the *specification*, the *implementation relation* and (possibly) a *test case specification* — also named in the literature *test purposes*. The *test derivation function* outputs a test set in the *test case* formalism;

- **Oracle**: decision procedure for the *satisfies* relation. Often in the testing literature the term *oracle* is overloaded given that it encompasses both of the following: the information contained in *test cases* allowing deciding of a *pass* or *fail* verdict — normally this information amounts to the *output* expected for each *input*; the decision procedure implementing the *satisfaction* relation. In this thesis we will consider both semantics for the *oracle* concept;

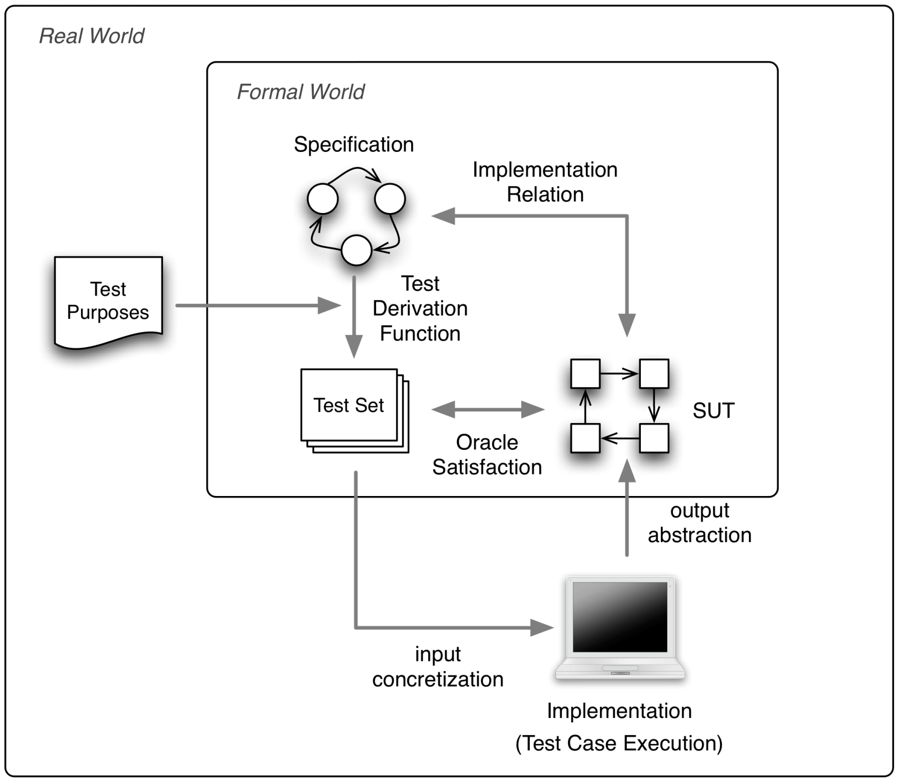


Figure 2.2: Testing — The Formal and the Real Worlds

- **Test purposes**: criteria for selecting test sets from the generally infinite set of *test cases* produced using the derivation function. The notion of *test intention* we propose in this thesis is a particular language implementing the concept of *test purpose*.

In figure 2.2 we present the connection between the formal actors we have described in the above item list and the real world — by real world we mean the software world where the implementation exists. The fact that we include the SUT object inside the *Formal World* may seem confusing, as the SUT formal object represents the result of performing a number of experiments by applying *test cases* to the *implementation*. In practice the SUT is obtained *on-the-fly* by the *test driver* which has the task of concretizing the *formal test set* into the language of the *implementation* and abstracting the outputs of the *implementation* into the SUT formalism — as Pretschner and Phillips explain in [39]. In real software testing environments the cycle *input concretization* – *test case execution* – *output abstraction* – *oracle satisfaction* is performed by the *test driver* for each *test case*.

Note that in order to connect the formal and the real worlds depicted in figure 2.2 there need to be some *minimal hypothesis* about the way in which the implementation is observed. In particular, the formalism used to represent SUTs must be rich enough to capture the functional aspects we wish to observe in the implementation in order establish a significative *implementation relation.*

2.2 Classification of formal Model-Based Testing Theories

Let us use the framework of concepts we have devised in section 2.1 to classify several different formal frameworks for model-based testing. Note that our classification is different from the one presented by Utting, Pretschner and Legeard in [40] in the sense that these authors are concerned with building a taxonomy for model-based testing *tools.* Our taxonomy regards *formal* model-based testing frameworks — or *theories* — which *explicitly* or *implicitly* underlie academic or commercial test generation tools.

One of the main characteristics of a model-based testing *theory* is the fact that an *implementation relation* between specifications and SUTs is clearly defined and thus algorithms for *exhaustive test set* construction exist. Establishing the implementation relation between a specification and an SUT by means of a *test set* amounts then to proving the correctness of the SUT — regarding, of course, that particular implementation relation. Given we are interested in black-box testing, the implementation relation disregards state and is only concerned with provided *inputs* and observed *outputs* of a specification and of an SUT.

When the *implementation relation* is not trivial, theories on model-based testing include the *algorithm* implementing the *test derivation function.* In this case it is essential to prove the *exhaustive test set* obtained by that algorithm has the properties of *soundness* and *completeness*[1] regarding the chosen *implementation relation. Soundness* refers to the fact the *exhaustive test set* does not falsely detect errors in correct SUTs; *Completeness* corresponds to the fact that if the SUT contains an error, the *exhaustive test set* will find it.

Unfortunately, in the general case, the *exhaustive test set* is infinite. This is due to the possibility of specifying infinite behaviors — e.g. a system which indefinitely processes inputs from the environment — or of having unbounded data types — e.g. recursively defined types in *algebraic specifications.* Given that *testing* is by definition a pragmatic activity, infinite *exhaustive test sets* cannot be produced. In order to avoid this problem and reach a *practicable* test set both in terms of size and

[1]Also called in the literature *unbiasedness* and *validity.*

cost, model-based testing theories usually formalize ways of reducing the *exhaustive test set*. The reduction — which in practice corresponds to *selecting* — can be performed by either using criteria extracted from the structure of the specification language or '*reasonable*' assumptions about the implementation of the SUT in order to find equivalence classes among test cases — i.e. *test cases* in the same *equivalence class* have the same *error uncovering power*.

Although some model-based testing theories are concerned with the conditions under which both *soundness* and *completeness* can be kept while reducing the *exhaustive test set*, from a practical point of view *soundness* seems more relevant is the sense that a test set finding errors in correct SUTs is not interesting and potentially harmful. On the other hand, *completeness* seems to be less important given that a *test set* not finding all the errors in an SUT may still discover some errors, which is the main purpose of the testing activity.

In the following sections we will analyze some model-based testing theories and classify them according to our taxonomy. We will start by the BGM (Bernot, Gaudel, Marre) testing theory which was initially devised for models written as *algebraic specifications*. We then move on to a theory designed around the *ioco* (input/output conformance) relation where the models and SUTs are specialized Labeled Transition Systems. These two theories are particularly interesting given that they use specifications written in formalisms which are at two opposite extremes: the BGM theory uses a *data type* based specification — algebraic specifications; *ioco* conformance testing uses LTS as specification language, which is a *behavior* centered formalism.

Subsequently we introduce an extension of the BGM theory to processes where the considered models are specialized LTSs where the edges are labeled by either an *inputs* or an *output*. Finally we will describe a model-based testing theory having as models CO-OPN specifications — which is the base for the work developed in the present thesis.

2.2.1 BGM

The BGM model-based testing theory was mainly introduced in the paper '*Testing can be Formal Too*' [41] by Gaudel. According to our taxonomy we can classify the approach in the following fashion:

- ***Specifications***: Algebraic Specifications;
- ***SUTs***: Pairs of equalities/inequalities between observable implemented sorts;
- ***Implementation Relation***: Notion of Σ-Algebra;

- ***Test Derivation Function***: Generate any pair of terms resulting from instantiating the variables of the axioms in the specification;
- ***Test cases***: Pairs of terms;
- ***Oracle Satisfaction***: Each test case corresponds to an equality in the SUT;
- ***Test Purposes***: Uniformity/Regularity hypothesis on the axioms variables;
- ***Minimal Hypothesis***: Implementation is a finitely generated Σ-Algebra.

One of the main characteristics of the BGM approach is the fact that *algebraic specifications* model stateless SUTs. The *implementation relation* corresponds to the fact that the implemented software behaves as a *model* for the *algebraic specification* — which is a Σ-Algebra. A Σ-Algebra is a family of sets and their associated functions (see definition 3.1.3) denoting the expected semantics of the implementation.

The *exhaustive test set* should verify the correct implementation of the functions described by the axioms in the algebraic specification. Given the implementations are stateless and deterministic, *test cases* can be built as equalities between the terms resulting from instantiating the left and the right side of the equations. A *test experiment* would then correspond to the evaluation of both terms of a *test case* by the implementation and their comparison using an equality predicate also present in the implementation. An error is found if the comparison fails.

Test purposes in the BGM theory are employed to reduce the *exhaustive test set*. The reduction consists of narrowing the domain of the variables present in the axioms of the algebraic specification. This narrowing can be performed by either *uniformity* hypothesis — all values in the domain induce the same behavior — or *regularity* hypothesis — all values in the domain induce the same behavior as a certain subset.

Reduction can be automatically performed using as criteria the structure of the specification. This is achieved using the *unfolding* technique [42] which corresponds to replacing an operation by its definition. Given an operation is usually defined by more than one axiom, performing *uniformity* hypothesis on the unfolded results produces an interesting coverage of the operation. Unfolding can be done more than once, in principle achieving further refined coverage.

Note that in our classification the SUT corresponds to pairs of equalities or inequalities between observable implemented sorts. This is in fact the formal object that results from performing a number of test experiments corresponding to the test cases produced from the specification.

The BGM model-based testing theory is introduced in further depth in chapter 6.

2.2.2 *ioco* Conformance Testing

Conformance testing has been studied for some time now, especially in the field of testing of communication protocols. In [43] Tretmans introduces a model-based testing theory using the *ioco* implementation relation. We classify Tretmans' approach as follows:

- ***Specifications***: LTS (Labeled Transition Systems);
- ***SUTs***: IOTS (Input Output Transition Systems);
- ***Implementation Relation***: *ioco* (input/output conformance);
- ***Test Derivation Function***: *ioco* test derivation algorithm;
- ***Test cases***: LTS with leaves including the test verdict {pass,fail};
- ***Oracle Satisfaction***: Synchronous execution of a *test case* with the SUT only reaches pass leaves;
- ***Test Purposes***: Incomplete LTS;
- ***Minimal Hypothesis***: Implementation can be modeled by an IOTS.

The formalism used to express specifications are Labeled Transition Systems (LTS). LTS is a generic formalism that represents the semantics of many languages for describing communicating processes. In particular, the labels of the edges of the LTS used as specifications for conformance testing are built using a vocabulary which is divided into *input* and *output* actions. The vocabulary of *outputs* is extended by a special action called *quiescence* (noted 'δ') which is reflexive and denotes the absence of outputs from a given state.

The SUTs are a particular kind of LTS, called IOTS (input/output transition systems), where all inputs are always enabled — which in practice means the implementation always accepts all inputs, even if most of them do not lead to a state change. The *quiescence* action is also present in the label vocabulary of the SUT and is typically observed in the implementation by timeouts.

The implementation relation in this particular model-based testing theory is *ioco* (input/output conformance). The relation is established by observing if the *outputs* of the SUT after a trace of the specification are always contained in the *outputs* of the specification after the same trace. An algorithm for test generation in presented in [43] and a proof of its *soundness* is given (i.e., a test case will not detect an error in a conformant SUT). The author is not worried about *completeness*, which reflects empiric testing in the sense that the goal is to discover errors, but not necessarily all of them. Finally, test purposes are given as incomplete LTS with mandatory actions.

2.2.3 *ioco* Conformance Testing extended to data types

Gaudel and Lestiennes have introduced in [44] a model-based testing theory which builds on both the BGM (see section 2.2.1) and the *ioco* conformance (see section 2.2.1) model-based testing theories. The approach is aimed at testing communicating processes which manipulate and exchange typed data. According to our taxonomy the approach is classified in the following fashion:

- ***Specifications***: LTS (Labeled Transition Systems) extended with edges labeled by typed values;
- ***SUTs***: IOTS (Input Output Transition Systems) extended with edges labeled by typed values;
- ***Implementation Relation***: *ioco* (input/output conformance);
- ***Test Derivation Function***: optimized *ioco* test derivation algorithm;
- ***Test cases***: LTS with leaves including the test verdict {pass,fail};
- ***Oracle Satisfaction***: Synchronous execution of a *test case* with the SUT only reaches pass leaves;
- ***Test Purposes***: Uniformity/Regularity hypothesis on the variables of predicates representing *paths* of the specification;
- ***Minimal Hypothesis***: Implementation can be modeled by an IOTS.

In terms of specification formalism, SUT formalism and implementation relation there are no fundamental differences regarding the original *ioco* Conformance Testing — except that the *input* and *output* action vocabulary includes typed values. Although the formal framework remains unchanged, from a practical point a view this results in general in much larger LTS specifications and SUTs.

The *test case* representation format is the same as the one in the original *ioco conformance testing*. The *test derivation function* has been optimized in the sense that *test cases* always leading to a *success* verdict are not included in the *exhaustive test set*. Gaudel and Lestiennes prove that their optimized *ioco* test derivation algorithm produces an *exhaustive test set* which is both *unbiased* (sound) and *valid* (complete).

Finally, *test purposes* are expressed as *uniformity* and *regularity* hypothesis on the variables of predicates representing *paths* of the specification. As in the original BGM approach, *unfolding* using the axiomatic definition of the operations of data types can be used to automatically refine uniformity hypothesis.

2.2.4 Object Oriented Testing from CO-OPN Specifications

In [45] a model-based testing theory for object oriented testing is presented, having as specification formalism the CO-OPN$_{/2}$ language. According to our taxonomy, we classify it as follows:

- ***Specifications***: CO-OPN$_{/2}$;
- ***SUTs***: LTS;
- ***Implementation Relation***: Bisimulation relation between transition systems;
- ***Test Derivation Function***: Generate all $HML\,formula \times \{true, false\}$ pairs using the specification's signature and semantics;
- ***Test cases***: $HML\,formula \times \{true, false\}$ pairs;
- ***Oracle Satisfaction***: *true* HML formulas are *satisfied* by the SUT; *false* HML formulas are ***not*** *satisfied* by the SUT;
- ***Test Purposes***: Uniformity/Regularity hypothesis on the variables of HML formulas representing *execution paths*, *events* or *event parameters* of the specification;
- ***Minimal Hypothesis***: Implementation can be modeled by a LTS.

The approach is similar to *ioco* conformance testing extended to data types (see section 2.2.3), in the sense that CO-OPN$_{/2}$ specifications model communicating objects as synchronized Petri Nets. CO-OPN$_{/2}$ specifications have LTS semantics and the implementation relation corresponds to the *bisimulation* relation between the LTS denoting the semantics of the specification and the LTS resulting from the observation of the implementation.

The *test case* representation formalism is the HML temporal logic and the approach distinguishes between *positive* and *negative* test cases as *true* and *false* logic formulas. In fact, there is a full agreement between the *bisimulation equivalence* and *HML equivalence* which allows using HML formulas as test cases. The *exhaustive test set* is the set of all possible $HML\,formula \times \{true, false\}$ pairs where HML formulas are built using the specification's events and the truth value reflects the semantics of those formulas having the specification as model. The *oracle satisfaction* then corresponds to each HML formula having the same semantics when the SUT is the considered model.

Finally, test purposes are written as *uniformity* or *regularity* hypothesis over HML formulas with variables. The hypothesis are written using a constraint language which limits the values the variables inside the HML formula can be instantiated to. *Unfolding* can be used in conjunction with the CO-OPN$_{/2}$ specification to automatically find interesting uniformity hypothesis. In this case the unfolding is more complex than the one in the approach described in section 2.2.3. An operation in CO-OPN$_{/2}$ can only occur if the conditions defined by the user, the conditions enabling the specified synchronizations and all the Petri Net transition firing conditions are satisfied. The *unfolding* must then be performed taking into consideration the definitions of all involved predicates.

Further explanations on object oriented testing from CO-OPN specifications are given in chapter 6.

2.3 The Unifying Role of Test Purposes

In figure 2.2 the *test purposes* are placed outside the formal world. Although this is not strictly true as *test purposes* are expressed in a formalism (possibly close to the modeling formalism), we have opted for this representation due to the fact that *test purposes* are a means of adapting the *test derivation function* to the real world. *Test purposes* can be seen as heuristics allowing the direction of the *test derivation function* (or algorithm) in order to produce a finite *practicable test set*.

The concept of *Test Purposes* is introduced in the literature by Ledru *et al.* in [34]. The authors provide an theoretical overview of the subject and establish in a semi-formal fashion the relations between *test purposes*, *test cases* and *specifications*. Subsequently the authors present in [46] the TOBIAS test selection framework where they implement *test purposes* as bounded regular expressions involving the operations of a VDM specification [17].

Mostly every model-based testing theory and tool has an explicit or implicit notion of *test purpose*, as the possible test set for even simple SUTs is infinite. In section 2.2 we have presented several notions of *test purposes* associated to model-based testing theories. At the tool level we can cite the example of the work of Legeard and Peureux described in [47]. For Legeard and Peureux the notion of *test purpose* is associated with generating traces (test cases) that reach certain *boundary states* of a state machine — reflecting the testing of corner values for the conditions stated in a B specification. Campbell, Grieskamp *et al.* introduce the *Spec Explorer* tool which is a test generator based on the *Abstract State Machine* formalism (ASM) developed by Gurevich *et al.* [48]. Here too the notion of *test purpose* is present in several forms, including *state grouping* (of the ASM), *parameter selection*, *method restriction* among others.

It is interesting to note that the notion of *test purpose* as defined by Ledru *et al.* in [34] is vast enough to encompass the following semantics:

- *Test Purposes* may be seen as *hypothesis* in the BGM sense (see section 2.2.1). A *test purpose* may be considered as *reduction hypothesis* (uniformity/regularity) for the *exhaustive test set* necessary to establish the correction of an implementation;

- *Test Purposes* may be seen as a mechanism for *selecting* test sets for an *implementation* having a given specification. Tools implementing conformance testing using the *ioco* (input/output conformance relation) [49] such TVEDA [50], TGV [51] or TorX [52] use *test purposes* which are incomplete Labeled Transition Systems (LTS) allowing the selection of particular paths in the LTS of the specification;

- Most model-based testing theories and tools include an implicit notion of *sufficient* coverage of an SUT which is related to the the structure of the specification formalism. For example, when *Algebraic Specifications* are considered (see section 2.2.1) an interesting possibility for testing the implementation in a non-exhaustive fashion corresponds to the application of an *unfolding technique* [31] to the specification axioms. The previously mentioned technique of Legeard and Peureux [53] of testing *boundary states* is also an example of exploiting the specification formalism — in this case the B language — in order to perform informed assumptions that allow reducing the size of the required *test set*.

  Test purposes also encompass this exploitation of the constructs of the specification formalism, although this kind of *test purposes* are not supposed to be explicitly stated by the *test engineer* — they correspond to test generation assumptions which are normally formally described in model-based testing theories and implicit in test generation tools;

- *Test purposes* can be seen from an operational point of view, in the sense that they can be used as heuristics to minimize the state space explosion of the algorithm implementing the *test derivation function*.

2.4 Summary

We have presented a state of the art of model-based testing, with an emphasis on model-based testing *theories*. We have started by identifying from a formal point of view the actors involved in model-based testing, namely: the *specification*, the *SUT*, the *implementation relation*, the *test cases*, the *test derivation function*, the *oracle* and the *test purposes*. We have then described them in a precise fashion, such that these concepts may used as a taxonomy for the classification of model-based testing theories. In particular we establish the borders between the *real* and the *formal* testing worlds which allows a clear distinction between model-based testing *tools* and *theories*. Also, we state that in order to test *real* implementations using a *formal* framework some kind of *minimal hypothesis* are necessary to allow passing between the *formal* and the *real* worlds.

In a second step we have introduced a number of model-based testing theories and classified them according to our taxonomy. The fundamental characteristic of a model-based testing theory is the fact that a formal *implementation relation* exists and that a — generally fictitious — *exhaustive test set* can be built to establish that relation. We have started by the BGM theory introduced by Bernot, Gaudel and Marre in which test cases are derived from *algebraic specifications*. We have then proceeded to study Tretman's *ioco conformance testing* framework for concurrent processes where the models are Labeled Transition Systems and the implementation relation is the well-known *input-output conformance relation*. *ioco conformance testing* has subsequently been extended by Lestiennes and Gaudel in order to generate test cases for concurrent processes exchanging typed data. Finally we classify a model-based testing theory having as specification language the CO-OPN$_{/2}$ language — specifications in this case being built as concurrent communicating objects.

We then focus our attention on *test purposes*, which from our point view encompass many roles in merging the theory and practice of model-based testing. The main subject of the present document is in fact to introduce a comprehensive *test purpose* — or, in our vocabulary, *test intention* — language.

Chapter 3

Formal Grounds

Let us start by introducing some notation that will be extensively used while defining the formal aspects of this thesis.

Throughout the document we consider a universe including the disjoint sets: $\mathsf{S}, \mathsf{F}, \mathsf{M}, \mathsf{G}, \mathsf{V}, \mathsf{I}$. These sets correspond respectively to the set of all sort, operation, method, gate, variable and test intention names. In particular we consider the S set to be made of two disjoint sets $\mathsf{S^A}$ and $\mathsf{S^C}$ which correspond to sort names in algebraic specifications and type names in classes.

The "*S-sorted*" notation facilitates the subsequent development. An S-sorted set A is a family of sets indexed by $S \subseteq \mathsf{S}$ and noted $A = (A_s)_{s \in S}$. Given two *S-Sorted* sets A and B, an *S-sorted* function $\mu : A \to B$ is a family of functions indexed by S and noted $\mu = (\mu_s : A_s \to B_s)_{s \in S}$.

Let $\leq \subseteq (S \times S)$ be a partial order, i.e. a reflexive, transitive and antisymmetric binary relation. We often extend $\leq$ to strings of equal length in S^* (Kleene closure of S) by $s_1, \ldots, s_n \leq s'_1, \ldots, s'_n$ iff $s_i \leq s'_i$ $(1 \leq i \leq n)$.

3.1 Order-Sorted Algebras

Definition 3.1.1 *Order-Sorted Signature*

An order-sorted signature *is a triple* $\Sigma = \langle S, \leq, F \rangle$, *where* $S \subseteq \mathsf{S}$ *is a finite set of sorts,* $\langle S, \leq \rangle$ *is a poset and* $F = (F_{w,s})_{w \in \mathsf{S}^*, s \in \mathsf{S}}$ *is a* $(\mathsf{S}^* \times \mathsf{S})$*-sorted set of function names of* F. *Each* $f \in F_{\emptyset,s}$ *is called a* constant.

Definition 3.1.2 *Set of all Terms, Ground Terms*

Let $\Sigma = \langle S, \leq, F \rangle$ *be a global signature and* X *be an S-sorted set of variables.*

The set of all terms over Σ with sort $s \in S$ denoted by $(T_{\Sigma,X})_s$ is built in the following fashion:

- $x \in (T_{\Sigma,X})_s$ *for all* $x \in X_s$,
- $f \in (T_{\Sigma,X})_s$ *for all* $f :\longrightarrow s' \in F$ *such that* $s' \leq s$,
- $f(t_1, \ldots, t_n) \in (T_{\Sigma,X})_s$ *for all* $f : s_1, \ldots, s_n \longrightarrow s' \in F$ *such that* $s' \leq s$ *and for all* $t_i \in (T_{\Sigma,X})_{s_i}(1 \leq i \leq n)$

The ground terms *for sorts $s \in S$ are terms without variables and are noted* $(T_{\Sigma,\emptyset})_s$.

Definition 3.1.3 *Partial Order-Sorted Σ-algebra*

Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature. A partial order-sorted Σ-algebra *consists of an S-sorted set $A = (A_s)_{s\in S}$ and a family of partial functions $F^A = (f^A_{s_1\ldots s_n,s})_{f:s_1\ldots s_n\rightarrow s\in F}$ where $f^A_{s_1\ldots s_n,s}$ is a function from $A_{s_1} \times \ldots \times A_{s_n}$ into A_s such that:*

- $s \leq s' \in S$ *implies* $A_s \subseteq A'_s$
- $f \in F_{s_1\ldots s_n,s} \cap F_{s'_1\ldots s'_n,s'}$ *with* $(s_1 \ldots s_n, s) \leq (s'_1 \ldots s'_n, s')$ *implies* $f^A_{s_1\ldots s_n,s}(a_1, \ldots, a_n) = f^A_{s'_1\ldots s'_n,s'}(a_1, \ldots, a_n)$

for all $a_i \in A_{s_i}$ $(1 \leq i \leq n)$.

We usually omit the family F^A and write A for an order-sorted Σ-algebra (A, F^A). The set of all order-sorted Σ-algebras is denoted by $Alg(\Sigma)$.

Definition 3.1.4 *Ground Term Algebra*

Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature and $(X_s)_{s\in S}$ be an S-Sorted set. The term algebra $Term_{\Sigma,s}(X)$consists of the carrier set $A = (T_{\Sigma,\emptyset})_s$ where $s \in S$ and of the family of functions $F^A = (f^A_{s_1\ldots s_n,s})_{f:s_1\ldots s_n\rightarrow s\in F}$ where:

- $f^A_{s_1\ldots s_n,s}$ *is a function from* $X_{s_1} \times \ldots \times X_{s_n}$ *into* $(T_{\Sigma,\emptyset})_s$
- $f^A_{s_1\ldots s_n,s}(t_1 \ldots t_n) = f(t_1 \ldots t_n)$ *such that* $f \in F_{s_1\ldots s_n,s}$ *and* $t_i \in X_{s_i}$

Definition 3.1.5 *Assignment, Interpretation*

Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature, X be a S-sorted variable set and A in $Alg(\Sigma)$. An assignment *for X into A is an S-sorted function $\sigma : X \rightarrow A$. An* interpretation *of terms of $T_{\Sigma,X}$ in A is an S-sorted partial function $[\![\_]\!]^\sigma_s : (T_{\Sigma,X})_s \rightarrow A_s$ where $s \in S$ is defined as follows:*

- *if* $x \in X_s$ *and* $s \leq s'$ *then* $[\![x]\!]^{\sigma}_{s'} = \sigma_s(x)$
- *if* $f :\rightarrow s \in F$ *and* $s \leq s'$ *then* $[\![f]\!]^{\sigma}_{s'} = f^A_s$
- *if* $f : s_1, \ldots, s_n \rightarrow s \in F$ *and* $s \leq s'$ *then*

$$[\![f(t_1,\ldots,t_n)]\!]^{\sigma}_{s'} = \begin{cases} f^A_{s'_1 \ldots s'_n, s'}\big([\![t_1]\!]^{\sigma}_{s_1}, \ldots, [\![t_n]\!]^{\sigma}_{s_n}\big) & \text{if all } [\![t_i]\!]^{\sigma}_{s_i} \text{ are defined} \\ \text{undefined otherwise} \end{cases}$$

Whenever the interpretation function is applied with an empty assignment we will omit the σ *parameter, i.e. instead of writing* $[\![\ldots]\!]^{\emptyset}_s$ *we will write* $[\![\ldots]\!]_s$.

Definition 3.1.6 *Equation, Positive Conditional Equation*

Let $\Sigma = \langle S, \leq, F \rangle$ *be an order-sorted signature and* X *be an* S*-sorted variable set. An* equation *is a pair* (t, t')*, denoted* $t = t'$ *such that* $t, t' \in (T_{\Sigma,X})_{s \in S}$*. A* positive conditional equation *is a set of pairs* $\{(t_i, t'_i), (t, t') \mid 1 \leq i \leq n\}$*, noted* $t_1 = t'_1, \ldots, t_n = t'_n \implies t = t'$*, where* t_i, t'_i $(1 \leq i \leq n)$ *are equations.*

Definition 3.1.7 *Equation Satisfaction*

Let $\Sigma = \langle S, \leq, F \rangle$ *be an order-sorted signature,* X *be a* S*-sorted variable set and* A *be in* $Alg(\Sigma)$*. Equation satisfaction is defined as follows:*

- $A, \sigma \models t = t' \iff [\![t]\!]^{\sigma}_s = [\![t']\!]^{\sigma}_{s'}$ *with* $t \in (T_{\Sigma,X})_s$ *and* $t' \in (T_{\Sigma,X})_{s'}$*;*
- $A, \sigma \models (t_1 = t'_1, \ldots, t_n = t'_n \implies t = t') \iff (A, \sigma \models t_1 = t'_1 \wedge \ldots \wedge A, \sigma \models t_n = t'_n \Rightarrow A, \sigma \models t = t')$ $(1 \leq i \leq n)$

3.2 Substitutions

Definition 3.2.1 *Substitution*

Let $\langle S, \leq, F \rangle$ *be an order-sorted signature and* X *be an* S*-sorted variable set. A substitution* $\theta \in Subs_S(X)$ *is a family of functions* $\theta_s : X_s \rightarrow (T_{\Sigma,X})_s$ *where* $s \in S$*. A ground substitution* $\theta \in GroundSubs_S(X)$ *is a family of functions* $\theta_s : X_s \rightarrow (T_{\Sigma,\emptyset})_s$ *where* $s \in S$*. In the context of substitutions we will often use the notation* $[v/x] \in \theta$ *to represent* $\theta(x) = v$*.*

Definition 3.2.2 *Term Substitution*

Let $\langle S, \leq, F \rangle$ *be an order-sorted signature and* X *be an* S*-sorted variable set. Let also* $\theta \in Subs_S(X)$ *be a substitution. A term substitution* $\theta^{\#}_s : (T_{\Sigma,X})_s \rightarrow (T_{\Sigma,X})_s$ *with* $s \in S$ *is defined as follows:*

- *if $x \in X_s$ and $s \leq s'$ then $\theta^{\#}_{s'}(x) = \theta_s(x)$*
- *if $f :\rightarrow s \in F$ and $s \leq s'$ then $\theta^{\#}_{s'}(f) = f$*
- *if $f : s_1, \ldots, s_n \rightarrow s \in F$ and $s \leq s'$ then*

$$\theta^{\#}_{s'}(f(t_1, \ldots, t_n)) = \begin{cases} f\big(\theta^{\#}_{s_1}(t_1), \ldots, f(\theta^{\#}_{s_n}(t_n)\big) & \textit{if all } \theta^{\#}_{s_i}(t_i) \textit{ are defined,} \\ \textit{undefined otherwise.} \end{cases}$$

Definition 3.2.2 allows performing syntactic substitutions in the terms of an order-sorted Σ-signature.

Definition 3.2.3 *Substitution /*

Let $\langle S, \leq, F\rangle$ be an order-sorted signature and $x \in (X_s)_{s\in S}$ a variable. A substitution / of x by a term $v \in (T_{\Sigma,X})_{s\in S}$ in a formula $f \in T_{\Sigma,X}$ is defined as follows:

$$f[v/x] = \theta^{\#}(f) \textit{ where } \theta(y) = \begin{cases} v \textit{ for } x = y \\ y \textit{ otherwise.} \end{cases} \textit{ and } \theta \in Subs_S(X)$$

3.3 Multi-Set Extension

A multi-set over a set E is a total mapping from E to $\mathbb{N}$. The set of all multi-sets over a set E is defined by the set of all functions $[E] = \big\{f \mid f : E \rightarrow \mathbb{N}\big\}$ equipped with the operations [_] (coercion to the single element), + (set union) and $\emptyset$ (empty set) defined as follows:

$$[e]^{[E]}(e') = \begin{cases} 1 \text{ if } e = e' \text{ for all } e, e' \in E \\ 0 \text{ otherwise.} \end{cases}$$

$$(f +^{[E]} g)(e) = f(e) + g(e) \text{ for all } f, g \in [E] \text{ and for all } e \in E$$
$$\emptyset^{[E]}(e) = 0 \text{ for all } e \in E$$

Definition 3.3.1 *Multi-set Extension of Order-Sorted Algebras*

Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature. The multi-set extension of Σ is defined as follows:

$$[\Sigma] = \big\langle S \cup \bigcup_{s\in S}\{[s]\},\ \leq \cup \bigcup_{s,s'\in S}\{\langle[s],[s']\rangle\}, F \cup F'\rangle\}\big\rangle$$

where $F' = \bigcup_{s\in S}\big\{\emptyset^{[s]} :\rightarrow [s], [\,\_\,]^{[s]} : s \rightarrow [s], +^{[s]} : [s], [s] \rightarrow [s]\big\}$ and $s \leq s'$

Definition 3.3.2 *Multi-set Extension of an Algebra*

The multi-set semantics extension of an order-sorted Σ*-algebra* A *is defined as follows:*

$$[A] = \left\langle A \cup \Big(\bigcup_{s \in S} [A_s]\Big),\ F^A \cup \Big(\bigcup_{s \in S} \emptyset^{[A_s]}, [\_]^{[A_s]}, +^{[A_s]}\Big)\right\rangle$$

3.4 Summary

In this chapter we have presented a set of definitions which will be extensively used as the formal basis for this thesis. In particular we have introduced the notion of *order sorted signatures* and *order sorted algebras* which provide an elegant fashion of formalizing some of the syntactic and semantic aspects of our approach. We then provided some useful definition for substituting variables with terms. Finally we have described a multi-set extension of the order-sorted framework in order to be able to both represent values in Petri Net *places* and express concurrency in the semantics of CO-OPN.

Chapter 4

CO-OPN – Introduction and Syntax

The method we present in this thesis for model-based test case generation uses CO-OPN (Concurrent Object-Oriented Petri Nets) as the specification language for expressing models of the SUT. CO-OPN is a very rich language, including mechanisms to natively handle concurrency, Abstract Data Types and encapsulation (through Object-Orientation and coordination modules). These mechanisms have deeply influenced our choices while building a testing theory and in particular our test intention language SATEL. This said, in order to be able to correctly introduce and justify our test generation technique we will start by presenting in this chapter a detailed account of the CO-OPN language.

We will provide a formal description of CO-OPN, including its formal abstract syntax and semantics. The presentation of the syntax and the semantics borrows from the work of Biberstein [1], Buffo [2] and Huerzeler [6] which we extend in order to have a clear abstract syntax and an operational semantics which is suitable for the test generation activity.

4.1 Historical Background

CO-OPN has been in continuous development since 1990 and it is based on two core formalisms — Many-Sorted Algebras [25] to model data types (including subtyping and polymorphism) and algebraic Petri Nets [26] to model behavior and concurrency. One of the unique features of CO-OPN is the fact that it has a thoroughly defined abstract syntax and, more importantly, a formal semantics — something relatively rare for a specification language of CO-OPN's dimension. A toolset has also been

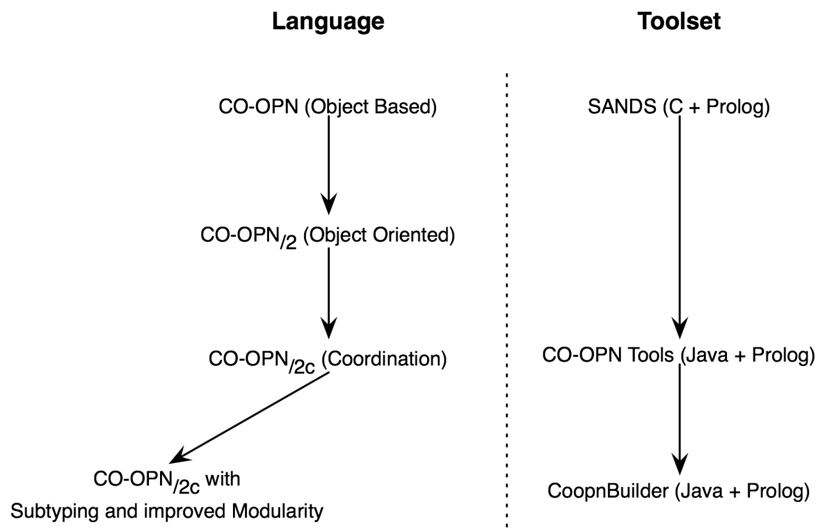

Language
Toolset
CO-OPN (Object Based)
CO-OPN/2 (Object Oriented)
CO-OPN/2c (Coordination)
CO-OPN/2c with Subtyping and improved Modularity
SANDS (C + Prolog)
CO-OPN Tools (Java + Prolog)
CoopnBuilder (Java + Prolog)

Figure 4.1: The evolution of CO-OPN

in continuous development along with the formal aspects of CO-OPN.

Figure 4.1 represents the evolution of the CO-OPN language, with the left row depicting the theoretical development and the right row depicting the tools that were built alongside. The distinguishing feature of the first version of CO-OPN (at the top of figure 4.1) is the fact that it is *object based*. With the term *object based* we mean that the typical concept of *class* as a template for the creation of object instances is not present. All the objects (encapsulated Petri Nets) are statically created and exist throughout all possible evolutions of the specification. CO-OPN objects collaborate with each other by means of synchronizations. Buchs and Guelfi [54] are the main authors of this first version for which a toolset called SANDS was written (in the C and Prolog programming languages). SANDS allows specification edition and animation.

Biberstein [1] was the main developer of CO-OPN$_{/2}$ which extends CO-OPN by introducing *object orientation*. This means that in CO-OPN$_{/2}$ the notion of *class* as a template for object creation is present, along with inheritance and dynamic object instantiation/destruction.

Buffo [2] added to CO-OPN$_{/2}$ the concept of *context* — implemented by COIL (Contexts and Objects Interface Language) — and created CO-OPN$_{/2c}$. *Contexts* are entities whose task is to coordinate the computation of a certain amount of objects that they manage. The idea behind the approach is to conceptually split the notion of computation — done by objects — and coordination — done by contexts. In particular, the notion of *context* is an interesting abstraction to model distributed systems where each context may correspond to a different execution environment (machine or network of machines). In each of those executions environments objects evolve and communicate with each other possibly using different technologies. Along with CO-OPN$_{/2}$ and COIL a new toolset was built, this time written in the Java and the Prolog programming languages. This toolset implemented most of the ideas present both in CO-OPN$_{/2}$ and COIL.

Huerzeler [6] introduced in his Ph.D. thesis a framework for describing subtyping relations (both syntactic and semantic) for component-based formalisms. One important contribution of Huerzeler's work was the clear formalization of the semantics of component-based systems, indirectly improving the work of Buffo on coordination. In fact, the work in the present thesis builds on both Buffo's and Huerzeler's ideas in order to provide a formal semantics to CO-OPN which is sufficiently elegant and unified to be useful for test case generation. Although Huerzeler's work was not concretized in a toolset, roughly at the same time a new toolset called *CoopnBuilder* was released. The new features of *CoopnBuilder* are that it includes a unified GUI (in the style of modern IDEs) and automatic code generation [36] (in Java).

It is important to notice that, given the very rich nature of the semantics of CO-OPN and the operational limitations of general purpose languages, the toolsets partially implement those semantics. However, important practical results have been achieved (e.g. [55, 56]) even only considering just a subset of CO-OPN's semantics.

In this thesis we will use the term CO-OPN to refer to the formal syntactic and semantic description included in CO-OPN$_{/2c}$, modified by our work in order to adapt some of the concepts developed by Buffo and Huerzeler to the activity of test generation.

4.2 Overview of CO-OPN

CO-OPN is a specification language built to describe concurrent and distributed systems, using the Object-Oriented approach to allow modularity and encapsulation. CO-OPN specifications are described along two axes — data types and behavior.

Data types are defined using an algebraic approach, more precisely the notion of Order-Sorted Algebras [57] which is an extension of the notion of Many-Sorted Algebras.

Behavior is fundamentally described by algebraic Petri Nets which are encapsulated within objects that may *provide* or *require* services from the outside — through ports respectively called *methods* and *gates*. CO-OPN objects collaborate with each other by means of synchronizations that connect *required* services

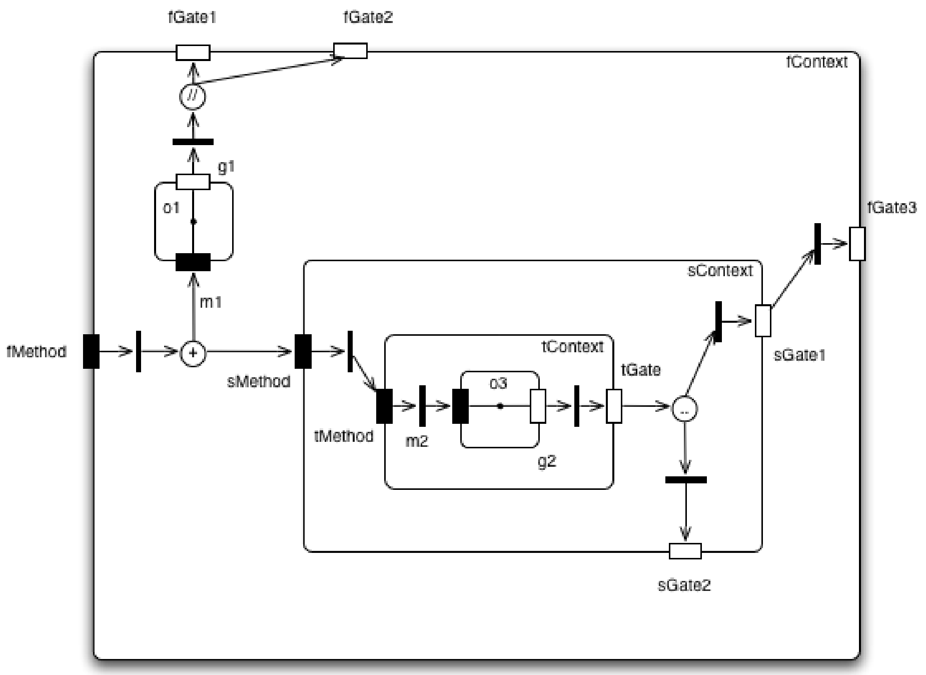


Figure 4.2: Example of CO-OPN specification

to *provided* services. These synchronizations (which may be complex expressions) are defined at the level of *contexts*, which are coordination units. Contexts also provide and require services *to* and *from* the outside through *method* and *gate* ports respectively. *Contexts* can be themselves coordinated by other contexts.

Example 4.2.1 *Figure 4.2 depicts a CO-OPN specification*[1] *composed of three imbricated contexts* fContext, sContext and tContext. fContext *coordinates* sContext *and the object* o1*;* sContext *coordinates* tContext*;* tContext *coordinates object* o3.

In the graphical syntax synchronization arcs connect the port that requires *a service to the port(s) that provide that (those) services — it is a 1-to-n connection. Complex synchronizations are defined through expressions that can be built using the operators '//' (simultaneity), '..' (sequence) and '⊕' (disjunction). For example, in figure 4.2 method* fMethod *of context* fContext *requires the service provided by method* m1 ***or*** *(notice the usage of the '⊕' operator) the service provided by method* sMethod*.*

[1]The graphical syntax in figure 4.2 is an abstraction of the real graphical syntax of CO-OPN where the internal behavior of objects is not described.

A CO-OPN specification is composed of modules that can be of three different types: *abstract data type* (ADT) modules to provide abstract algebraic definitions of data types; *class* modules to define templates for object creation; *context* modules to define coordination units. In the concrete syntax all CO-OPN modules share the same syntactic structure, including the following sections:

- A *Header*, including the module name and additional information concerning genericity or inheritance;
- An *Interface* section including information describing the services the module provides to other modules. The *interface* section is inspired by the notion of *signature* in the Order-Sorted Algebra framework which defines the sorts (data type names) and the names of the operations for those sorts together with their arities — arity meaning the sort names of the domains and the co-domains of the operations;
- A *Body* section, describing the behavior (the semantics) of the services it provides to the exterior. The *body* section is inspired from the notion of Order-Sorted Algebraic *specification* which, besides the *signature* for a set of data types, also includes a set of axioms in equational logic[2]. These axioms define in an abstract fashion the behavior of the operations described in the signature. The *body* of an *ADT*, *class* and *context* module include a set of specific axioms to describe the behavior of the operations or services that module provides.

Being a very rich specification language, CO-OPN includes a series of interesting mechanisms to tackle the modeling activity. The following sections provide descriptions of the most relevant of those mechanisms.

4.2.1 Abstract Data Structures

The ADT modules allow describing types in a completely abstract fashion, following the ideas of the Order-Sorted Algebra framework. In fact ADT modules implement the notion of Order-Sorted Algebraic specification — in an ADT module it is possible to describe a type by its signature (the module's interface) and its set of axioms (the module's body including algebraic formulas and theorems). This is a very powerful way of representing data structures given that, apart from the type's syntax, we also declaratively describe its properties independently of any implementation. Also, ADT modules allow declaring sub-sorting (using the sub-sort relation in the Order-Sorted Algebra framework), partial operations, overloading and polymorphism.

[2]First-order logic where the only predicate is the equality.

ADT modules are typically used to define basic types such as *integers*, *booleans*, *strings* or *stacks*. However, given that they have the power of Order-Sorted Algebraic specifications, they can actually be used to model any data structure and any associated computation.

4.2.2 Object Orientation

A CO-OPN object is instantiated from a template declared in a *class* module. The *interface* of a *class* module includes, among other information, the name of the class type and the names of the *methods* and *gates* present in the module along with their arity — in this case a list of sort names which make up the parameters of those methods or gates. The *body* of a *class* module declares a number of axioms that describe the behavior of the services the class provides to the outside (through *method* ports). This behavior is expressed in terms of the evolution of the Petri Net an instance of that class holds, and also in terms of required services from the outside (through *gate* ports).

In particular, CO-OPN implements the following concepts in Object Orientation:

- *Inheritance and Subtyping*: *Inheritance* is achieved syntactically, by inheriting all the services a given class provides. It is also possible to add services or to change the services provided by the parent class. *Subtyping* is a semantic concept, where an object of a certain type can be replaced by an object of another type only if the semantics of the whole system remains unchanged. The conformance relation is based on *bisimulation* between the semantics of the two types. Given the fact that verifying the *bisimulation* relation between the common parts of two Labeled Transition Systems (LTS) is an undecidable problem, the concept of *subtyping* has been defined in [1] and [6] but has not been implemented in the toolsets;

- *Object Encapsulation*: An object in CO-OPN holds a state which corresponds to the marking of the algebraic Petri Net inside the object. The only way to change the state of a CO-OPN object is by requesting a synchronization with one of the object's services, accessible through method ports. In this sense CO-OPN promotes object encapsulation, given that access to an object's internal state is not directly modifiable from the exterior;

- *Dynamic Object Instantiation*: All classes in CO-OPN implicitly include a *creation* and a *destruction* methods which allow dynamic instantiation and destruction of objects;

- *Object identity*: The formalism dynamically manages the identity of the objects present in a CO-OPN specification, automatically creating or deleting object references in case of object creation or destruction;

- *Communication*: CO-OPN components (objects or contexts) communicate by requesting services from each other in a synchronous fashion. A component requests a service from another component (or a set of components) by synchronizing one of its *gate* ports with a method (or a set of methods) belonging to the component(s) that can provide the service. The possible synchronizations for a group of components are defined by a surrounding context.

4.2.3 Coordination and Distribution

CO-OPN provides a specific mechanism for modeling distribution and coordination by means of *context* modules. These modules define the way in which communication happens between a set of components (objects or other contexts) using synchronization expressions.

More precisely, in order to provide a service a component may require several other services from various components to happen according to a certain synchronization expression. Synchronization expressions are built using three binary synchronization operators: *'a // b'* (simultaneity), meaning services a and b should be possible at the same time — in other words enough resources are available for the two services to execute simultaneously; *'$a \oplus b$'* (disjunction), meaning that either service a or service b should be possible; *'$a \,..\, b$'* (sequence), meaning that service b should happen after service a — in other words, after executing service a, there are enough resources left to execute service b. It is important to say that complex synchronizations between components are *atomic* and *transactional*: either all the services are provided the way the synchronization expression specifies, or no changes occur to the global state of the model.

In terms of distribution *contexts* provide an abstraction level for modeling interactions between components. The fact that interactions are described by synchronizations makes it possible to model systems that will eventually be implemented using heterogeneous communications mechanisms. Also, the fact that components rely on an upper layer to provide coordination helps in separating computing *and* communication, thus avoiding non-trivial communication protocols between components that accumulate both responsabilities.

4.2.4 Concurrency and Non-Determinism

In a CO-OPN specification components (objects and contexts) communicate and evolve concurrently. Concurrency is modeled at the level of the communication between components since when a component a requires a service from a component b, both a and b evolve simultaneously (if possible). The '//' (simultaneity) operator allows extending this concept by requiring services from several components simultaneously.

Non-determinism is modeled in the communication between components by using the '$\oplus$' (disjunction) operator or by requiring a synchronization with any object which is a member of a given class.

4.3 The Banking Server example

We will now introduce the example specification we will be using throughout this thesis to introduce our test selection technique. The specification models a simplified concurrent Banking system where several users may connect simultaneously and then perform standard banking operations. Before being able to perform operations on his/her account, the user has to authenticate in a two-step process: log in with a username; if the username exists, the system randomly proposes a *challenge* to the user and asks for the password corresponding to that challenge. If the user provides three wrong passwords, his/her account will become blocked and he/she will no longer be able to connect to the system. After having successfully authenticated, the operations available to a user are *balance display*, *money deposit* and *money withdrawal*.

We will build the model of our Banking Server system bottom-up, starting by the necessary data structures, then proceeding to an Object-Oriented analysis of the problem, and finally building the coordination layer that orchestrates the communication between the components of the system.

For clarity and space economy reasons in this chapter we will present a subset of the Banking Server specification. In particular all the error handling part (e.g. wrong password, user not logged) is absent, as well as the blocking behavior after three wrong login attempts. We refer the reader to appendix B for the full specification.

4.3.1 Data Structure analysis

A brief analysis of the problem statement allows identifying the need to model the following concepts as data structures: the *money* in the account; the *challenge* proposed by the system after login; the *password* for each user; the *users*[3] themselves.

Given the similarities between the data structures, we will only present in this chapter the definition of the ADTs *Challenge* and *Money*. The definition of the remaining data structures for the *Banking Server* specification can be found in appendix B.

```
ADT Password;

Interface

    Use
        Digit;
        Booleans;

    Sort
        password;

    Generator

        newPassword _ _ _ _ : digit digit digit digit -> password;
    Operation

        _ = _ : password password -> boolean;

Body

    Axioms
      (n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true
              => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = true;
      ! ((n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true)
              => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = false;

    Where
        n1, n2, n3, n4, m1, m2, m3, m4 : digit;

End Password;
```

Figure 4.3: The Password ADT

In figure 4.3 we can find the definition of the *password* ADT in its concrete syntax. The purpose of this data structure is to model the password the user will insert after the *login* step. As we can see in the *interface* of the module, the type name (the sort) defined by the module is *password*. The *Generator* and *Operation* fields[4] define the profile of the available operations for the *password* sort. In this

[3]In a real model a *user* would probably be described by large amount of information and would likely be modeled by a class. For our example purposes a simple identifier suffices.

[4]In the concrete syntax of ADT modules we distinguish between *Generator* operations — the

case the only operation we will need is an equivalence relation between elements of the sort (in order to be able to compare passwords) which we declare as an infix binary predicate '='.

In the *body* of the module we find the abstract behavior of the predicate '=' defined through conditional axioms. The first axiom states that the relation holds when all the digits of both passwords are equal one by one. The second case covers the cases where the relation does not hold. Both axioms use a set of variables (defined in the *where* field) in order to generalize the behavior.

```
ADT User;

  Inherit Characters;

    Rename
      char -> user;

End User;
```

Figure 4.4: The User ADT

The ADT in figure 4.4 defines the *user* sort as a renaming of the sort *char* declared in the ADT modules *Characters* — we identify a user simply by a character. The inheritance defined in the the module is syntactic, meaning that the ADT module *Characters* is totally reused simply changing the sort name *char* to *user*. The simple renaming does not imply a subtyping relationship between sorts, which in practice means we cannot substitute a *user* by a *char* — although the syntax of ADT modules does allow real subtyping.

4.3.2 Object-Oriented analysis

In order to model the Banking Server we have decided to create two classes that model two main concepts of the system: the security (login management) part and the accounts. The *LoginManager* class is described in both its graphical and textual concrete syntaxes in figures 4.5 and 4.6 respectively. The services provided by the class are defined as the black rectangles on the border of outer square in figure 4.5 and in the field *Methods* of figure 4.6. The *LoginManager* class provides four services: check if a user with a given identifier is currently logged ('*isLogged u*' method); login a user with a given identifier ('*login u*' method); insert a password for a given user ('*insertPassword u p*' method); logout a given user ('*logout u*' method).

ones which are used to build the available elements typed by sort — and "normal" *Operations* — the ones that have the *sort* as co-domain but do not produce new elements for that sort.

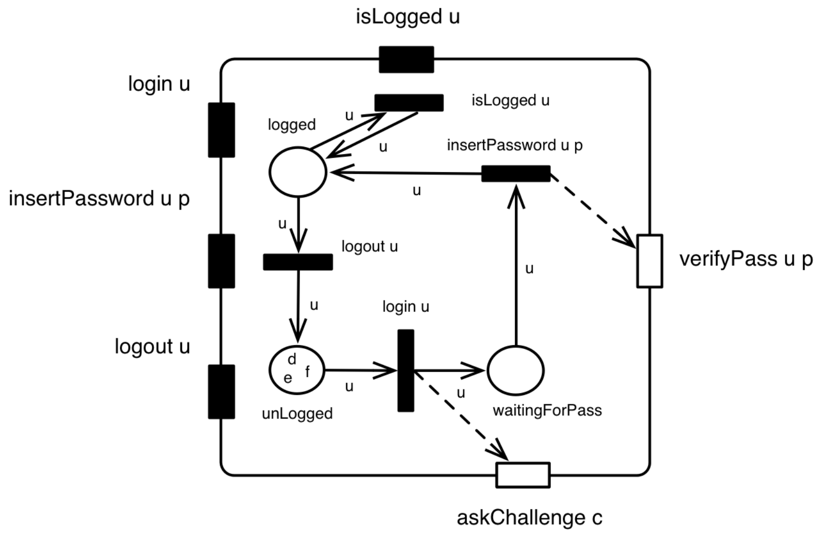


Figure 4.5: LoginManager Class (graphical syntax)

The behavior of the class is given by a Petri Net which can be graphically seen in figure 4.5. The Petri Net includes three places (*unLogged*, *waitingForPass* and *logged*) which represent the possible states of a given user. The initial state of the net includes three users in the *unLogged* place which we have represented by the three tokens (‘d’, ‘e’ and ‘f’) inside the place. In the textual syntax these three places are declared in the field *Places* and the initial state is defined in the field *Initial*.

The semantics of the Petri Net inside an object of type *loginManager* is the typical Petri Nets semantics extended by following modifications: the transitions inside the class are synchronized with the class methods holding the same name; the transitions may require other services to execute in order to fire. To illustrate this concept let us examine the ‘*insertPassword u p*’ transition in its graphical syntax in figure 4.5. This transition takes a *user* token from the *unLogged* place and puts it in the *Logged* place. However, this only happens when the ‘*insertPassword u*’ service is called through the method of the same name and the user *u*[5] exists in the *unLogged* place. Also, the transition requests the ‘*verifyPassword u p*’ service from the outside (the dotted arrow from the transition to the ‘*verifyPass u p*’ method) to check if the password is the correct one for the given user. Notice that in our model the password for a given user is contained in the user’s account.

Referring now to the textual syntax of the *loginManager* class in figure 4.6,

[5]In the present context *u* and *p* are variables of type *user* and *password* respectively.

```
 0 Class LoginManager;

 2 Interface

 4     Use
           Password;
 6         Challenge;
           User;
 8
       Type
10         loginManager;

12     Gates
           verifyUser _ : user;
14         verifyPass _ _ : user password;
           askChallenge _ : challenge;
16
       Methods
18         isLogged _ : user;
           login _ : user;
20         insertPassword _ _ : user password;
           logout _ : user;
22
   Body
24
       Places
26         unLogged _ : user;
           waitingForPass _ : user;
28         logged _ : user;

30     Initial
           unLogged d, unLogged e, unLogged f;
32
       Axioms
34
           login u With
36                 this . askChallenge (newChal a 1)::
                   unLogged u -> waitingForPass u;
38         insertPassword u p With
                   this . verifyPass u p::
40                 waitingForPass u -> logged u;
           logout u::
42                 logged u -> unLogged u;
           isLogged u::
44                 logged u -> logged u;

46     Where
           u : user;
48         p : password;
           this : loginManager;
50
   End LoginManager;
```

Figure 4.6: LoginManager Class (textual syntax)

the '*insertPassword u p*' transition is declared in the axiom in line 38. The axiom is divided into four parts:

- *Condition*: the empty condition;
- *Event*: '*insertPassword u p*' declares the name of the service the axiom describes and its parameters. In the graphical syntax in figure 4.5 this corresponds to the '*insertPassword u p*' transition and its associated method (in the border of the outer square);
- *Synchronization*: '*this . verifyPass u p*' corresponds to requiring a service from the outside (through a gate of the object) in order to verify the password is correct for the given user;
- *Preconditions/Postconditions*: '*waitingForPass u -> logged u*' declares the weight of the input and the output arcs of transition '*insertPassword u p*'.

4.3.3 Coordination analysis

Given the Object-Oriented analysis performed in section 4.3.2 let us now introduce the *context* module that will coordinate the activity of the objects that compose the model of the Banking Server. The module is depicted in figure 4.7 and includes three objects: the *lmObj* which is an instance of the *LoginManager* class and two instances of the *Account* class called *accObj1* and *accObj2*. In the textual syntax of the module (figure 4.8) the declaration of the objects that compose the class is done in the *Objects* field.

As we have previously mentioned, the object coordination itself is achieved by linking required services to provided services. Let us examine how the *withdraw* operation (see line 35 of figure 4.8) is achieved at the context level: the '*withdraw u am*' port of the *BankingServer* context requires the '*isLogged u*' service from the *loginManager* object and (simultaneously) the '*withdraw u am*' service from the user's account. In the textual syntax this coordination expression is declared by the axiom in line 35 of figure 4.8. The axiom is split into the *required* and the *provided* parts by the *With* keyword.

Notice that in the provided part of the axiom '*lmObj . isLogged u // accVar . withdraw u am*' the *accVar* variable stands for any object of type *Account*. In practice, this means that the synchronization will be done on any object of the class in a non-deterministic fashion. Since in fact only one account will respond at a time — the account belonging to the correct user — this mechanism allows modeling an account database. If we would add more *Account* objects to the model no changes would be necessary at the coordination level.

At the graphical level the coordination axioms are represented by arrows and synchronization operators connecting *required* to provided services. Since the rela-

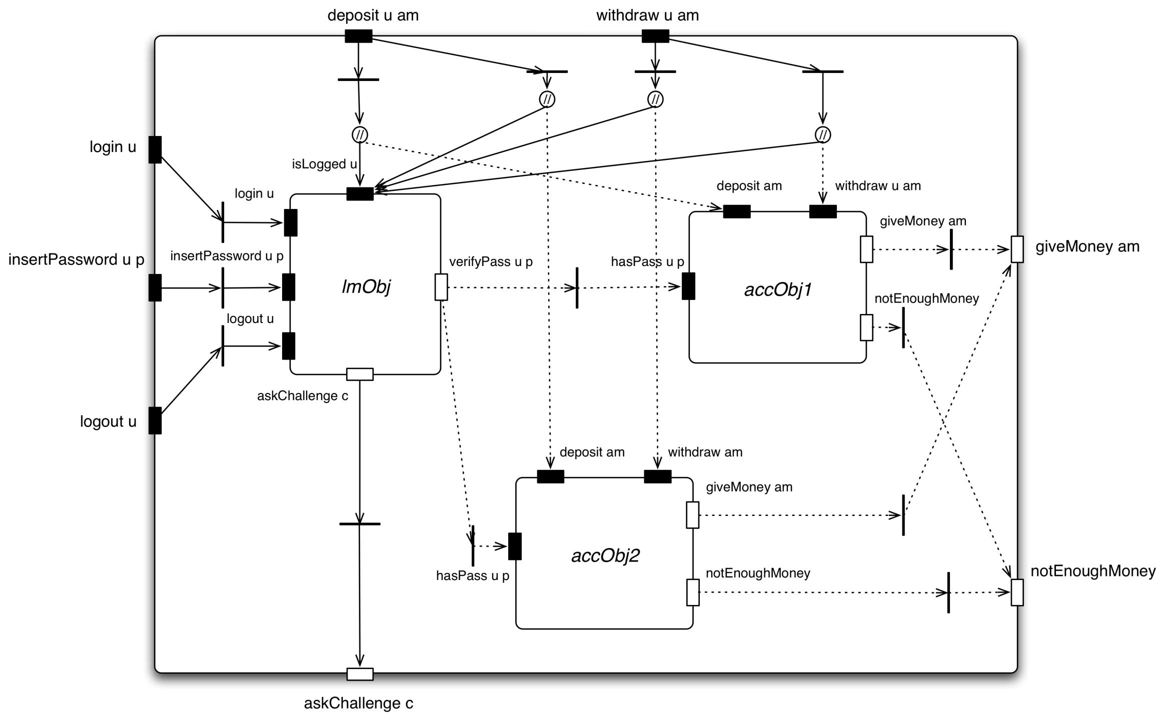


Figure 4.7: The BankingServer Context (graphical syntax)

tion between the textual syntax (figure 4.8) and the graphical syntax (figure 4.7) is trivial, we will not provide further explanations on the subject. Let us just mention that graphically we have replicated (using dotted arrows) coordination expressions involving services provided non-deterministically by any of the instances of the *Account* class.

```
Context BankingServer;

Interface

    Use
        User;
        Money;
        Password;
        Challenge;
        LoginManager;
        Account;

    Gates
        askChallenge _ : challenge;
        giveMoney _ : money;

    Methods
        login _ : user;
        insertPassword _ _ : user password;
        logout _ : user;
        deposit _ _ : user money;
        withdraw _ _ : user money;

Body

    Objects
        lmObj : loginManager;
        accObj1 : account;
        accObj2 : account;

    Axioms
        login u With lmObj . login u;
        insertPassword u p With lmObj . insertPassword u p;
        logout u With lmObj . logout u;
        deposit u am With lmObj . isLogged u // accVar . deposit am;
        withdraw u am With lmObj . isLogged u // accVar . withdraw u am;
        accVar . giveMoney am With giveMoney am;
        lmObj . askChallenge c With askChallenge c;
        lmObj . verifyPass u p With accVar . hasPass u p;

    Where
        accVar : account;
        u : user;
        p : password;
        c : challenge;
        am : money;

End BankingServer;
```

Figure 4.8: The BankingServer Context (textual syntax)

4.4 Abstract Syntax of CO-OPN/$_{2c++}$

In this section we will present the abstract syntax of CO-OPN/$_{2c++}$ in a formal fashion. The purpose of the description is not only to provide a detailed definition of our specification language, but also to establish a formal basis on which the semantics of CO-OPN and later of our test language SATEL will be based on. Note that in this description is strongly inspired from [58], although we introduce new concepts at the level of the abstract syntax for *context* modules.

We do not provide a concrete syntax for CO-OPN as we consider that the examples of specifications throughout the present thesis provide enough insight. However, the interested reader may consult [1] for an account on the subject.

Let us remind the reader that throughout following text we consider a universe including the disjoint sets: S, F,M, G and V. These sets correspond respectively to the set of all sort, operation, method, gate, and variable names. In particular we consider the S set to be made of two disjoint sets $\mathsf{S^A}$ and $\mathsf{S^C}$ which correspond to sort names in algebraic specifications and type names in classes.

4.4.1 Signature and Interfaces

An ADT module signature groups three elements of an algebraic abstract data type, i.e. a set of sorts, a sub-sort relation, and some operations. However, in the context of structured specifications, an ADT signature can intrinsically use elements not *locally* defined, i.e. defined outside the signature itself. For this reason, the profile of the operations as well as the sub-sort relation in the next definition are respectively defined over the set of *all* sorts names S and not only over the set of sorts S^{A} defined in the module itself. When a signature only uses elements locally defined we say that the signature is *complete.*

Definition 4.4.1 *ADT module signature*

An ADT module signature *(ADT signature for short) (over* S *and* F*) is a triple*[6] $\Sigma^{\mathsf{A}} = \langle S^{\mathsf{A}}, \leq^{\mathsf{A}}, F\rangle$*, where:*

- S^{A} *is a set of sort names of* $\mathsf{S^A}$*;*
- $\leq^{\mathsf{A}} \subseteq (S^{\mathsf{A}} \times \mathsf{S^A}) \cup (\mathsf{S^A} \times S^{\mathsf{A}})$ *is a partial order (partial sub-sort relation);*
- $F = (F_{w,s})_{w \in \mathsf{S^A}, s \in \mathsf{S}}$ *is a* $(\mathsf{S}^* \times \mathsf{S})$*-sorted set of function names of* F.

[6]The A superscript indicates that the module and its components are in relation with the abstract data type dimension.

Similarly to the notion of ADT module signature, the elements of a class module which can be used from the outside are grouped into a class module interface. The class module interface of a class module includes: the type of the class, a sub-type relation with other classes, the set of methods that corresponds to the services provided by the class and the set of gates that corresponds to the services required by the class.

Definition 4.4.2 *Class module interface*

A class module interface *(class interface for short) (over* S *and* F*) is a quadruple*[7] $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, G \rangle$*, where:*

- $c \in S^{\mathsf{C}}$ *is the type*[8] *name of the class module;*
- $\leq^{\mathsf{C}} \subseteq (\{c\} \times \mathsf{S}^{\mathsf{C}}) \cup (\mathsf{S}^{\mathsf{C}} \times \{c\})$ *is a partial order (partial sub-type relation);*
- $M = (M_{c,w})_{w \in \mathsf{S}^*}$ *is a finite* $(\{c\} \times \mathsf{S}^*)$*-sorted set of method names of* M*;*
- $G = (G_{c,w})_{w \in \mathsf{S}^*}$ *is a finite* $(\{c\} \times \mathsf{S}^*)$*-sorted set of gate names of* G*;*

Recall that a method is not a function but a parameterized transition which may be regarded as a predicate. The set of methods M is $(\{c\} \times \mathsf{S}^*)$-sorted, where c is the type of the class module and S^* corresponds to the sorts of the method's parameters. A method $m \in M_{c,s_1,\ldots,s_n}$ is often noted $m_c : s_1, \ldots, s_n$, while a method without any argument $m \in M_{c,\epsilon}$ is written m_c (ϵ denotes the empty string). The same argument is valid for gates.

Let us also introduce the notion of interface for context modules. As for class modules, a *context module interface* describes the elements of a context module that can be seen from the outside. Unlike class modules, context modules are not typed — they exist only as singletons. In this sense, all typing information that is needed for class modules is irrelevant for context modules. In the concrete syntax context modules have names, but since we can distinguish them simply from their set of *methods* and *gates* we will use only this information at the level of the interface's abstract syntax.

Definition 4.4.3 *Context module interface*

A context module interface *(over* S*) is a pair*[9] $\Xi^{\mathsf{X}} = \langle M, G \rangle$ *where:*

[7]The C superscript stresses the belonging to the class dimension.

[8]In general, we use s symbols for sorts of the abstract data type dimension and c symbols for types (in fact sorts) of the classes,

[9]As for the previous module interfaces, the X superscript indicates the belonging to the context dimension.

- *$M = (M_w)_{w \in \mathsf{S}^*}$ is a S^*-sorted set of method names of M;*
- *$G = (G_w)_{w \in \mathsf{S}^*}$ is a S^*-sorted set of gate names of G;*

We will now introduce a series of definitions which we will use in the subsequent development of this thesis. From a set of ADT signatures $\boldsymbol{\Sigma} = (\Sigma_i^{\mathsf{A}})_{1 \leq i \leq n}$ and a set of class interfaces $\boldsymbol{\Omega} = (\Omega_j^{\mathsf{C}})_{1 \leq j \leq m}$ we build a *global sub-sort/sub-type relation* denoted $\leq_{\boldsymbol{\Sigma},\boldsymbol{\Omega}}$ which gathers the partial sub-sort and sub-type relations of the elements of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Omega}$ as follows:

$$\leq_{\boldsymbol{\Sigma},\boldsymbol{\Omega}} = \Big(\bigcup_{1 \leq i \leq n} \leq_i^{\mathsf{A}} \cup \bigcup_{1 \leq j \leq m} \leq_j^{\mathsf{C}} \Big)^*$$

Note that the in the above definition the R^* notation represents the *reflexive* and *transitive* closure for binary relation R.

For each class interface $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, G \rangle$, we induce an ADT signature $\Sigma_{\Omega^{\mathsf{C}}}^{\mathsf{A}} = \langle \{c\}, \leq^{\mathsf{C}}, F_{\Omega^{\mathsf{C}}} \rangle$ in which $F_{\Omega^{\mathsf{C}}}$ contains the operations necessary to the management of object identifiers. These operations are defined on the global sub-sort/sub-type relation as follows:

$$\begin{aligned} F_{\Omega^{\mathsf{C}}} = \; & \{ init_c : \rightarrow c, new_c : c \rightarrow c \} \cup \\ & \{ sub_{c,c'} : c \rightarrow c', super_{c,c''} : c \rightarrow c'' \mid c' \leq_{\Sigma,\Omega} c, c \leq_{\Sigma,\Omega} c'' \} \end{aligned}$$

Definition 4.4.4 *Global signature*

Let $\boldsymbol{\Sigma} = (\Sigma_i^{\mathsf{A}})_{1 \leq i \leq n}$ be a set of ADT signatures and $\boldsymbol{\Omega} = (\Omega_j^{\mathsf{C}})_{1 \leq j \leq m}$ be a set of class interfaces such that $\Sigma_i^{\mathsf{A}} = \langle S_i^{\mathsf{A}}, \leq_i^{\mathsf{A}}, F_i^{\mathsf{A}} \rangle$ and $\Omega_j^{\mathsf{C}} = \langle \{c_j\}, \leq_j^{\mathsf{C}}, M_j, G_j \rangle$. The global signature over $\boldsymbol{\Sigma}$ and $\boldsymbol{\Omega}$ is:

$$\Sigma_{\boldsymbol{\Sigma},\boldsymbol{\Omega}} = \Big\langle \bigcup_{1 \leq i \leq n} S_i^{\mathsf{A}} \cup \bigcup_{1 \leq i \leq m} \{c_j\}, \leq_{\boldsymbol{\Sigma},\boldsymbol{\Omega}}, \bigcup_{1 \leq i \leq n} F_i \cup \bigcup_{1 \leq j \leq m} F_{\Omega_j^{\mathsf{C}}} \Big\rangle$$

Since the context interfaces do not define types, they are not included in the *global sub-sort/sub-type* relation. A possible extension to the current version of CO-OPN would include introducing dynamic contexts. This would justify context type names and introducing a context sub-type relation in the context interface.

4.4.2 ADT Module

ADT modules describe abstract data types which may be used by other ADT or class modules. An ADT module consists of an ADT signature, a set of *positive conditional* *equations* also called axioms, and some variables. Remember that, in the context of structured specifications, an ADT module may use elements not locally defined, i.e. defined in other modules.

Definition 4.4.5 *ADT module*

Let Σ be a set of ADT signatures and Ω be a set of class interfaces such that the global signature $\Sigma_{\Sigma,\Omega} = \langle S, \leq, F \rangle$ is complete. An ADT module *is a triplet $Md^{\mathsf{A}}_{\Sigma,\Omega} = \langle \Sigma^{\mathsf{A}}, X, \Phi \rangle$, where:*

- Σ^{A} *is an ADT signature;*
- $X = (X_s)_{s \in S}$ *is a S-disjointly-sorted set of variables of* $\mathbf{V}$*;*
- Φ *a set of positive conditional equations over $\Sigma_{\Sigma,\Omega}$ and X.*

4.4.3 Behavioral Formulas

Before defining a behavioral formula let us define the set of events possible over a set of CO-OPN objects. The *object events* correspond to internal transitions of an object which are implicitly synchronized with the object's *method* ports of the same name[10]. Internal transitions of a given object can optionally be synchronized with *gate* ports of that same object through synchronization expressions. As we have previously explained, three synchronization operators are provided: '//' for simultaneity, '..' for sequence, and '$\oplus$' for alternative. *Gate* ports express required services for a transition to be fired and can themselves be connected to other CO-OPN components (through *coordination events* as described in section 4.4.5) allowing component communication. As an example, if we consider an object o of type *LoginManager* as defined in figure 4.5, the *object event*[11]:

```
o.insertPass d (newPass 1 2 3 4) with o.verifyPass d (newPass 1 2 3 4)
```

would be a possible event of object o where the transition '*insertPass d (newPass1 2 3 4)*' requires the '*verifyPass d (newPass1 2 3 4)*' service from outside the object. Notice that we use the '.'(*dot*) notation to represent a method or gate port belonging to a specific object.

[10]In [1] the additional notion of *invisible event* is defined, where an object's transition may *not* be synchronized with a method port and thus may occur spontaneously. Due to the fact that this modeling feature is rarely used, we have not taken it into consideration in this thesis.

[11]written in CO-OPN's concrete syntax

CO-OPN$_{/2c}$ includes the possibility of directly synchronizing a method port of a given object with method ports of other objects (or itself). As an example, if we consider two objects *o1* and *o2* of type *LoginManager* the *object event*:

```
o1.insertPass d (newPass 1 2 3 4) with o2.insertPass e (newPass 5 6 7 8)
```

would be a possible event which involves simultaneous logins of users 'd' and 'e' in two different *LoginManager* objects. In CO-OPN$_{/2c++}$ we do not allow this kind of synchronization as, without loss of generality, a *method–method* synchronization can be converted into a *gate–method* synchronization at the level of the *context module* coordinating the synchronized objects. We propose a technique for converting CO-OPN$_{/2c}$ specifications to CO-OPN$_{/2c++}$ specifications in appendix D.

We write $\mathbf{OE}_{A,C,P,O}$ for the set of all events over a set of parameter values A, a set of types of classes C, set of ports P and a set of object identifiers O. Because this set is used for various purposes, we give here a generic definition.

Definition 4.4.6 *Gate Synchronization Expressions, Object Events*

Let $S = S^{\mathsf{A}} \cup S^{\mathsf{C}}$ be a set of sorts such that $S^{\mathsf{A}} \subseteq \mathbf{S^A}$ and $S^{\mathsf{C}} \subseteq \mathbf{S^C}$, $A = (A_s)_{s \in S}$ and $C \subseteq S^{\mathsf{C}}$ be a set of type names. Let also $P = \langle M, G \rangle$ be a pair of $S^{\mathsf{C}} \times S^*$-sorted sets of method and gate names and $O = (O_s)_{s \in S^{\mathsf{C}}}$ a set of object identifiers. The gate synchronization expressions $GateSyncExpr_{A,C,P,O}$ are built as follows:

- $o.g(v_1, \ldots, v_n) \in GateSyncExpr_{A,C,P,O}$ *for all* $g \in (G_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in O_c$
- $sync\ op\ sync' \in GateSyncExpr_{A,C,P,O}$ *for all* $sync, sync' \in GateSyncExpr_{A,C,P,O}$, $op \in \{//, .., \oplus\}$

The object events $\mathbf{OE}_{A,C,P,O}$ *are built as follows:*

- $o.create \in \mathbf{OE}_{A,C,P,O}$ *for all* $o \in (O_s)_{s \in S^{\mathsf{C}}}$
- $o.destroy \in \mathbf{OE}_{A,C,P,O}$ *for all* $o \in (O_s)_{s \in S^{\mathsf{C}}}$
- $o.m(v_1, \ldots, v_n) \in \mathbf{OE}_{A,C,P,O}$ *for all* $m \in (M_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in (O_c)_{c \in C}$
- $o.m(v_1, \ldots, v_n)$ ***with*** $sync \in \mathbf{OE}_{A,C,P,O}$ *for all* $m \in (M_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in (O_c)_{c \in C}$, $sync \in GateSyncExpr_{A,C,P,O}$
- $ev\ op\ ev' \in \mathbf{OE}_{A,C,P,O}$ *for all* $ev, ev' \in \mathbf{OE}_{A,C,P,O}$, $op \in \{//, .., \oplus\}$

In every 'm ***with*** *sync' event (sync $\in ObjSyncExpr_{A,C,P,O}$) 'm' and any gates of 'sync' are ports of the same object identifier instance. In other words, methods of an object are only synchronized with gates of the same object.*

Note that the actual and the formal parameters of a method involved in a synchronization may not have the same sorts, but their sorts must be connected through the sub-sort relation. Also, the left side of the event includes a single port synchronization, while on the right side it may include a complex synchronization expression involving several ports.

We now give the definition of the behavioral formulas that are used to describe the properties of events of class modules. A behavioral formula consists of an *object event* as established in definition 4.4.6, a condition expressed by means of a set of equations over algebraic values, and the usual Petri Net pre/post-condition of the event. Both pre/post-conditions are sets of terms (of sort multi-set) indexed by the places of the net. A event can occur if and only if the condition on the algebraic values is satisfied, enough resources can be consumed/produced from/in the places of the module, and if the events involved in the synchronization can occur.

Definition 4.4.7 *Behavioral formula*

Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature such that $S = S^{\mathsf{A}} \cup S^{\mathsf{C}}$ ($S^{\mathsf{A}} \subseteq \mathbf{S^A}$ and $S^C \subseteq \mathbf{S^C}$). For a given pair $P = \langle M, G \rangle$ of $S^{\mathsf{C}} \times S^*$-sorted port names, an S-disjointly-sorted set of places Pl, a set of types $C \subseteq S^{\mathsf{C}}$ and an S-disjointly-sorted set of variables X, a behavioral formula *is a quadruplet $\langle Event, Cond, Pre, Post \rangle$ where:*

- *$Event \in \mathbf{OE}_{(T_{\Sigma,X}),C,P,(T_{\Sigma,X})_s}$ such that $s \in S^{\mathsf{C}}$;*
- *$Cond$ is a set of equations over Σ and X;*
- *$Pre = (Pre_p)_{p \in Pl}$ and $Post = (Post_p)_{p \in Pl}$ are two families of terms over $[\Sigma], X$ indexed by Pl and of sort $[s]$ if p is of sort s.*

We also denote a behavioral formula $\langle Event, Cond, Pre, Post \rangle$ by the expression

$$Event :: Cond \Rightarrow Pre \rightarrow Post$$

It is important to notice in definition 4.4.7 that the event part of the behavioral formula belongs to $\mathbf{OE}_{(T_{\Sigma,X}),C,P,(T_{\Sigma,X})_s}$ where $T_{\Sigma,X}$ corresponds to terms with variables built on the signature Σ. In this way the *object events* in the behavioral formulas may contain variables, allowing the description of generic behaviors. The same reasoning is valid for the *condition* and the *pre-* and *post-* condition part of behavioral formulas.

Example 4.4.8 *Consider the following behavioral formula in line 38 of figure 4.6 which is defined in the concrete syntax of CO-OPN:*

insertPassword u p ***with*** *this . verifyPass u p::*
waitingForPass u -> logged u;

According to the abstract syntax in definition 4.4.7 this behavioral formula can be split in the following components:

- *the* event *part: 'insertPassword u p* ***with*** *this . verifyPass u p'*
- *the* condition *part: empty*
- *the* pre-condition *part: 'waitingForPass u'*
- *the* post-condition *part: 'logged u'*

Notice that in the example 'u'*,* 'p' *are variables.*

4.4.4 Class Module

The purpose of a *class module* is to describe a collection of objects with the same structure by means of an encapsulated algebraic net. Actually, a class module is considered as a template from which objects are instantiated. A class module consists of: a class interface, a set of places, some variables, the initial values of the places (also called the *initial state* of the module), and a set of behavioral formulas which describe the properties of the methods and of the internal transitions.

Note that the following definition establishes that class instances are able to store and exchange object identifiers because the sorts of the places, the variables, and the profile of the methods belong to the set of all sorts S, therefore, these components can be either of sort $\mathsf{S^A}$ or $\mathsf{S^C}$.

Definition 4.4.9 *Class module*

Let Σ be a set of ADT signatures and Ω be a set of class interfaces such that the global signature $\Sigma_{\Sigma,\Omega} = \langle S, \leq, F\rangle$ is complete. A Class module *is a quintuplet $Md^{\mathsf{C}}_{\Sigma,\Omega} = \langle \Omega^{\mathsf{C}}, P, I, X, \Psi\rangle$, where:*

- *$\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, G\rangle$ is a class interface;*
- *$P = (P_s)_{s\in S}$ is a finite S-disjointly-sorted set of place names;*

- *$I = (I_p)_{p \in P}$ is an initial marking, a family of terms indexed by P and of sort $[s]$ if p is of sort s;*
- *$X = (X_s)_{s \in S}$ is an S-disjointly-sorted set of variable of $\mathbf{V}$;*
- *Ψ is a set of behavioral formulas over: the global signature $\Sigma_{\Sigma,\Omega}$; a set of methods composed of M and all the methods of Ω; a set of gates composed of G and all the methods of Ω; the set of places P; the set $\{c\}$; and X.*

4.4.5 Coordination Formulas

Let us now define the concept of *coordination formula*, which is similar to the concept of *behavioral formula*. *Coordination formulas* deal with synchronizations involving any kind of CO-OPN component — objects and contexts — and also state under which conditions the events resulting from those synchronizations may occur. Let us first define the set of possible *coordination events* over a *local context*, a set of *interior contexts* being coordinated by the *local context* and a set of *interior objects* also being coordinated by the local context.

In order to introduce the notion of *coordination events* let us analyze figure 4.7 depicting the *Banking Server* context and the components it coordinates. Examples of *coordination events* in this system are:

```
login d with lmObj . login d
```

The context *method port* '*login*' is synchronized with the *method port* of the same name of the object '*lmObj*'.

```
lmObj . verifyPass d (newPass 1 2 3 4) with
    accObj1 . hasPass d (newPass 1 2 3 4)
```

The object *gate port* '*verifyPass*' is synchronized with the *method port* '*hasPass*' of the object '*accObj1*'.

Remember that *coordination events* correspond to the coordination of components living inside a context and as such may correspond to synchronizations between the local context and any of its internal components, or between internal components themselves.

Definition 4.4.10 *Required Service Expression, Provided Service Expression, Coordination Events*

Let $S = S^{\mathsf{A}} \cup S^{\mathsf{C}}$ be a set of sorts such that $S^{\mathsf{A}} \subseteq \mathbf{S}^{\mathbf{A}}$ and $S^{\mathsf{A}} \subseteq \mathbf{S}^{\mathbf{A}}$. Let us consider $A = (A_s)_{s \in S}$ and $L = \langle M, G \rangle$ and $I = \langle M', G' \rangle$ two pairs of S^*-sorted sets of context method and gate names. Also, let $IO = \langle M'', G'' \rangle$ be a pair of $S^{\mathsf{C}} \times S^*$-sorted sets of method and gate names and $O = (O_s)_{s \in S^{\mathsf{C}}}$ be a set of object identifiers. The required service object expressions set $ReqExpr_{A,L,I,IO,O}$ is built according to the following rules:

- $o.g(v_1, \ldots, v_n) \in ReqExpr_{A,L,I,IO,O}$ *for all* $g \in (G''_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in O_c$
- $g(v_1, \ldots, v_n) \in ReqExpr_{A,L,I,IO,O}$ *for all* $g \in (G'_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$
- $m(v_1, \ldots, v_n) \in ReqExpr_{A,L,I,IO,O}$ *for all* $m \in (M_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$

The provided service object expressions set $ProvExpr_{A,L,I,IO,O}$ is built as follows:

- $o.create \in ProvExpr_{A,L,I,IO,O}$ *for all* $o \in (O_s)_{s \in S^{\mathsf{C}}}$
- $o.destroy \in ProvExpr_{A,L,I,IO,O}$ *for all* $o \in (O_s)_{s \in S^{\mathsf{C}}}$
- $o.m(v_1, \ldots, v_n) \in ProvExpr_{A,L,I,IO,O}$ *for all* $m \in (M''_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in O_c$
- $m(v_1, \ldots, v_n) \in ProvExpr_{A,L,I,IO,O}$ *for all* $m \in (M'_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$
- $g(v_1, \ldots, v_n) \in ProvExpr_{A,L,I,IO,O}$ *for all* $g \in (G_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$
- $prov \ op \ prov' \in ProvExpr_{A,L,I,IO,O}$ *for all* $prov, prov' \in ProvExpr_{A,L,I,IO,O}$, $op \in \{//, .., \oplus\}$

Finally the coordination events $\mathbf{CE}_{A,L,I,IO,O}$ are defined in the following fashion:

- req ***with*** $prov \in \mathbf{CE}_{A,L,I,IO,O}$ *for all* $req \in ReqExpr_{A,L,I,IO,O}$, $prov \in ProvExpr_{A,L,I,IO,O}$

We can now formally introduce the concept of *coordination formula* which state under which conditions *coordination events* may happen. Namely, conditions are stated by algebraic equations involving the parameters of the port synchronizations involved in the event.

Definition 4.4.11 *Coordination formulas*

Let $\Sigma = \langle S, \leq, F \rangle$ be an order-sorted signature such that $S = S^{\mathsf{A}} \cup S^{\mathsf{C}}$ ($S^{\mathsf{A}} \subseteq \mathbf{S}^{\mathbf{A}}$ and $S^{\mathsf{C}} \subseteq \mathbf{S}^{\mathbf{C}}$). Let us consider a pair L of S^*-sorted methods and gates of a local context module, a pair I of S^*-sorted methods and gates of context modules *contained by the local context, a pair IO of* $S^{\mathsf{C}} \times S^*$*-sorted methods and gates of objects modules contained by the local context, an* S^{C}*-disjointly-sorted set of internal objects contained by the local context and a* S*-disjointly-sorted set of variables* X*. A* coordination formula *is a pair* $\langle Cond, CoordEv\rangle$*, where:*

- $Cond$ *is a set of equations over* Σ *and* X*;*
- $CoordEv \in \mathbf{CE}_{(T_{\Sigma,X}),L,I,IO,(T_{\Sigma,X})_s}$ *where* $s \in S^{\mathsf{C}}$

We also denote a behavioral formula $\langle Cond, CoordEv\rangle$ *by the expression*

$$Cond \Rightarrow CoordEv$$

Example 4.4.12 *Consider the following* coordination formula *in line 38 of figure 4.8 which is defined in the concrete syntax of CO-OPN:*

lmObj . verifyPass usr p ***with*** *accVar . hasPass u p;*

According to the abstract syntax in definition 4.4.11 this coordination formula can be split in the following components:

- *the* event *part: 'lmObj . verifyPass u p* ***with*** *accVar . hasPass u p'*
- *the* condition *part:* empty condition

Notice that in the example 'u' *and* 'p' *are variables,* 'lmObj' *is an object reference, and* 'accVar' *is a variable standing for any object reference of type* 'account'*.*

4.4.6 Context Module

Context Modules coordinate the interaction of a number of CO-OPN components. They are thus composed of other *contexts* and *objects* and include a number of coordination formulas involving the ports of the local context and/or the ports of the coordinated components.

Definition 4.4.13 *Context module*

Let Σ *be a set of ADT signatures and* Ω *be a set of class interfaces such that the global signature* $\Sigma_{\Sigma,\Omega} = \langle S, \leq, F\rangle$ *is complete. Let also* Ξ *be a set of context signatures. A context module is a quintuplet* $Md^{\mathsf{X}}_{\Sigma,\Omega,\Xi} = \langle \Xi^{\mathsf{X}}, O, C, X, \chi\rangle$ *where:*

- $\Xi^{\mathsf{X}} = \langle M, G\rangle$ *is a context module interface;*

- *$O = (O_s)_{s \in S^{\mathsf{C}}}$ is an S^{C}-sorted set of static objects and $S^{\mathsf{C}} \subseteq S$;*
- *$C = \{Md_i^{\mathsf{X}}\}_{i \in \{1,\ldots,n\}}$ is a family of context modules where $Md_i^{\mathsf{X}} = \langle \Xi_i^{\mathsf{X}}, O_i, C_i, X_i, \chi_i \rangle$, $\Xi_i^{\mathsf{X}} = \langle M, G \rangle_i$ and $\Xi_i^{\mathsf{X}} \in \Xi$;*
- *$X = (X_s)_{s \in S}$ is an S-sorted set of variables;*
- *χ is a set of coordination formulas over $T_{\Sigma,X}$, $\langle M, G \rangle$, $\bigcup_i \langle M, G \rangle_i$ $(1 \leq i \leq n)$, a pair of methods and gates of the classes objects O belong to, the objects O themselves and variables X.*

4.4.7 CO-OPN Specification

We can now present the notion of CO-OPN specification as a set of ADT, Class and Context modules built from a set of ADT signatures, Class interfaces and Context interfaces. The definition of CO-OPN specification we introduce in this thesis extends the one introduced by Biberstein in [1]. It also integrates the syntax of context modules introduced by Buffo in [2] in the typical *algebraic specification* syntactic style adopted in CO-OPN.

Definition 4.4.14 *CO-OPN Specification*

Let Σ be a set of ADT module signatures and Ω be a set of class module interfaces such that the global signature $\Sigma_{\Sigma,\Omega}$ is complete. Let also Ξ be a set of context module signatures. A CO-OPN specification consists of a set of ADT, Class and Context modules such that:

$$Spec_{\Sigma,\Omega,\Xi} = \left\{ (Md_{\Sigma,\Omega}^{\mathsf{A}})_i \,|\, 1 \leq i \leq n \right\} \cup \\ \left\{ (Md_{\Sigma,\Omega}^{\mathsf{C}})_j \,|\, 1 \leq j \leq m \right\} \cup \left\{ (Md_{\Sigma,\Omega,\Xi}^{\mathsf{X}})_k \,|\, 1 \leq k \leq o \right\}$$

Two dependency graphs can be constructed from a CO-OPN/$_{2c++}$ specification *Spec*. The first one consists of the dependencies within the algebraic part of the specification, i.e between the various ADT modules. The second dependency graph corresponds to the "*contains*" relationship between the *context* modules. Both these graphs are composed of the specification *Spec* and a binary relation over *Spec* denoted noted D_{Spec}^{A} for the algebraic dependency graph, and D_{Spec}^{X} for the *context* "*contains*" dependency graph. The relation D_{Spec}^{A} is constructed as follows: for any ADT module Md, Md' of *Spec* $(Md \neq Md')$, $(Md, Md') \in D_{Spec}^{\mathsf{A}}$ if and only Md uses some elements defined in the signature of Md'. As for the relation

D^{X}_{Spec}, it is constructed as follows: for any *context* module Md, Md' ($Md \neq Md'$), $(Md, Md') \in D^{\mathsf{X}}_{Spec}$ if and only if Md' is part of the *context* modules contained in Md.

A *well-formed* CO-OPN/$_{2c++}$ specification is thus a specification with two constraints concerning the dependencies between the modules which compose the specification. These hierarchical constraints are necessary for the theory of algebraic specifications and in order to allow the construction of the semantics of a CO-OPN/$_{2c++}$ specification, as we will show in the next chapter.

Definition 4.4.15 *Well-formed CO-OPN/$_{2c++}$ specification*

A CO-OPN/$_{2c++}$ specification Spec is well-formed *if and only if:*

1. *the algebraic dependency graph $\langle Spec, D^{\mathsf{A}}_{Spec}\rangle$ has no cycle;*
2. *the "contains" dependency graph $\langle Spec, D^{\mathsf{X}}_{Spec}\rangle$ has no cycle.*

4.5 Summary

We have started by providing an historical introduction to the CO-OPN language. From a formal point of view CO-OPN was developed in several iterations: a first *object based* CO-OPN version encapsulated Algebraic Petri Nets within static structures providing a set of services synchronized with the nets' transitions; a second CO-OPN$_{/2}$ object oriented version including *class templates*, *subtyping* and dynamic object creation and destruction; a third CO-OPN$_{/2c}$ version included coordination modules allowing the modeling of distribution. More recently some work was developed at the level of the semantics of the *coordination* and *composition* of CO-OPN components.

All the formal work on CO-OPN was relatively dispersed, with each of the versions introducing new notations which were not necessarily well integrated with the previous ones. In particular, the formalization of the CO-OPN$_{/2c}$ version did not take into consideration a good syntactic integration with the previous object oriented version and the semantics were defined in a transformational manner — not directly taking into consideration the new *coordination* and *composition* concepts of the language. The most recent work on the semantics of *coordination* and *composition* by Huerzeler [6] was done at an abstract level and was not directly applied to the formal definition of CO-OPN.

We have thus decided to introduce a new integrated and reviewed version of the CO-OPN specification language — which we have called CO-OPN$_{/2c++}$. In this chapter we introduce a new formal definition of the abstract syntax of the language. The new abstract syntax is based on the work of Buffo [32], which we have reviewed for better presentation, integration and bug correction. In particular we have introduced:

- distinct *object events* and *coordination events* which allowed us to later define a new *coordination* and *composition* semantics;
- a new syntax for *context module interfaces* and *context modules*, following the algebraic specification style used in CO-OPN$_{/2}$. This new notation also allowed introducing a methodology for formally adding new modules to the CO-OPN language, which we have used while formalizing *test intention* modules;
- a new definition of CO-OPN specifications, including *context modules.* We have also added a new constraint on the *well-formedness* of a CO-OPN specification, i.e. that there are no cycles in the "*contains*" dependency graph of the specification's context modules.

Within the chapter we have also introduced a complete CO-OPN specification of a *Banking Server* application, illustrating many of the features of the language. The only features not explored in the example are *subsorting*, *subtyping* and dynamic object *creation* and *destruction.* The *Banking Server* example will be used in the subsequent chapters in order to provide examples for the definitions of CO-OPN and SATEL.

Given the fact that this chapter is a revision of the abstract syntax of CO-OPN, some of the formal definitions present in this chapter are inspired or taken without change from [58] and [32].

Chapter 5

CO-OPN – Semantics

This section presents the semantical aspects of the CO-OPN$_{/2c++}$ formalism which are based on two notions: *order-sorted algebras* and *transition systems*.

First of all, we concentrate on order-sorted algebras as models of the data structures of a CO-OPN$_{/2c++}$ specification. Then, we introduce an essential element of the CO-OPN$_{/2c++}$ formalism, namely the order-sorted algebra of object identifiers, which is organized in a very specific way.

Afterwards, we present how the notion of transition system is used so as to describe a system modeled by a CO-OPN$_{/2c++}$ specification. Then, we provide all the inference rules which allow us to construct the transition system representing the semantics of a CO-OPN$_{/2c++}$ *context* module. The semantics of a CO-OPN$_{/2c++}$ specification *Spec* will correspond to the union of all the semantics of the *context* modules of *Spec* which are at the top of the *context* module "containment" hierarchy.

5.1 Algebraic Models of a CO-OPN$_{/2c++}$ Specification

Here we focus on the semantics of the algebraic dimension of a CO-OPN$_{/2c++}$ specification. Remember that an ADT signature can be deduced from each class interface of the specification. It is composed of a type, of a subtype relation, and of some operations required for the management of the object identifiers. We now provide the definition of the ADT module induced by each class module of the specification. Such an ADT module is composed of the induced ADT interface and of the formulas which determine the intended semantics of the operations.

Definition 5.1.1 *ADT module induced by a class module*

Let Spec be a well-formed CO-OPN$_{/2c{+}{+}}$ specification and $\leq$ be its global subsort/subtype relation. Let $Md^{\mathsf{C}} = \langle \Omega^{\mathsf{C}}, P, I, V, \Psi \rangle$ be a class module of Spec in which $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, G \rangle$. The ADT module induced by Md^{C} is denoted $Md^{\mathsf{C}}_{\Omega^{\mathsf{C}}} = \langle \Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}}, V_{\Omega^{\mathsf{C}}}, \Phi_{\Omega^{\mathsf{C}}} \rangle$ in which $\Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}} = \langle \{c\}, \leq^{\mathsf{C}}, F_{\Omega^{\mathsf{C}}} \rangle$, and where:

- $F_{\Omega^{\mathsf{C}}} = \{init_c : \rightarrow c,\ new_c : c \rightarrow c\} \cup$
  $\{sub_{c,c'} : c \rightarrow c',\ super_{c,c''} : c \rightarrow c'' \mid c' \leq c, c \leq c''\}$;
- $V_{\Omega^{\mathsf{C}}} = \{o_c : c,\ o_{c'} : c' \mid c' \leq c\}$;
- $\Phi_{\Omega^{\mathsf{C}}} = \{sub_{c,c'}(init_c) = init_{c'},$
  $sub_{c,c'}(new_c\, o_c) = new_{c'}(sub_{c,c'}\ o_c),$
  $super_{c',c}(init_{c'}) = init_c,$
  $super_{c',c}(new_{c'}\ o_{c'}) = new_c(super_{c',c}\ o_{c'}) \mid c' \leq c\}$

Each class module (see definition 4.4.9) defines a type and a subtype relation which are present in the ADT module induced by each class module presented in definition 5.1.1. On the one hand, each type (actually a sort) defines a carrier set which contains all the object identifiers of that type and, on the other hand, the global subtype relation imposes a specific structure over the carrier sets. Moreover, four operations are defined in each ADT module induced by each class module. These operations over the object identifiers are divided into two groups: the generators (the operations which build new values) and the regular operations. For each type c and c' of the specification these operations are as follows:

1. the generator $init_c$ corresponds to the first object identifier of type c;
2. the generator new_c returns a new object identifier of type c;
3. the operation $sub_{c,c'}$ maps the object identifiers of type c into the ones of type c', when $c' \leq c$;
4. the operation $super_{c',c}$ maps the object identifiers of types c' into the ones of type c, when $c' \leq c$;
5. as indicated by their names, $super_{c',c}$ is the inverse operation of $sub_{c,c'}$.

The presentation of a CO-OPN$_{/2c{+}{+}}$ specification $Spec$ — denoted $Pres(Spec)$ — consists of collapsing all the ADT modules of the specification and all the ADT modules which are induced by the class modules. Renamings are necessary to avoid name clashes between the various modules.

In the following text, we often denote the set of all sorts, types, methods, and places of a specification $Spec$ by, $Sorts(Spec)$, $Types(Spec)$, $Methods(Spec)$,

$Places(Spec)$, respectively. Note that $Sorts(Spec)$ represents the sorts defined in the ADT modules of the specification, while $Types(Spec)$ corresponds to the sorts (types) induced by the class modules.

The initial approach has been adopted throughout this work. The semantics of the algebraic dimension of a CO-OPN$_{/2c++}$ specification *Spec* is therefore defined as the semantics of the presentation of the specification. We denote this model $Sem(Pres(Spec))$.

The semantics of such a presentation is composed of two distinct parts. The first one consists of all the carrier sets defined by the ADT modules of the specification, i.e. the model of the algebraic dimension of the specification without considering the ADT modules induced by the class modules. The second part is what we previously called the *object identifier algebra.* This "sub-algebra" is constructed in a very specific way and plays an important role in our approach because it provides all the potential object identifiers as well as the operations required for their management.

Let $Sem(Pres(Spec)) = A$. The models of the ADT modules of the specification are usually denoted $\ddot{A}$, while the object identifier algebra defined by the ADT modules induced by the class modules of the specification is $\widehat{A}$. $\ddot{A}$ and $\widehat{A}$ are disjoint and $A = \ddot{A} \cup \widehat{A}$.

Intuitively, the idea behind the object identifier algebra of a specification is to define a set of identifiers for each type of the specification and to provide some operations which return a new object identifier whenever a new object has to be created. Moreover, these sets of object identifiers are arranged according to the subtype relation over these types. It means that two sets of identifiers are related by inclusion if their respective types are related by subtyping. In other words, the inclusion relation reflects the subtype relation. Furthermore, two operations mapping subtypes and supertypes are provided. The structure of the object identifier algebra and the operations mapping subtypes and supertypes can be used to determine, for example, if an object identifier belongs to a given type.

5.2 Management of Object Identifiers

Whenever a new class instance is created, a new object identifier must be assigned to it. This means that the system must know, for each class type and at any time, the last object identifier used, so as to be able to compute a new object identifier. Consequently, throughout its evolution, the system retains a function which returns the last object identifier used for a given class type. Moreover, another information has to be retained throughout the evolution of the system. This information consists of the objects that have been created and that are still alive, i.e. the object identifiers assigned to some class instances involved in the system at a given time. This second information is also retained by means of a function the role of which is to return, for every class type, a set of object identifiers which corresponds to the alive (or active) object identifiers.

For the subsequent development, let us consider a specification *Spec*, $A = \mathit{Sem}(\mathit{Pres}(\mathit{Spec}))$, and the set of all types of the specification $S^{\mathsf{C}} = \mathit{Types}(\mathit{Spec})$.

The partial function which returns, for each class, the last object identifier used, is a member of the set of partial functions:

$$\mathit{Loid}_{\mathit{Spec},A} = \{l : S^{\mathsf{C}} \to \widehat{A} \mid l(c) \in \widetilde{A}_c \text{ or is not defined}\}$$

in which $\widetilde{A}_c = \widehat{A}_c \setminus \cup_{c'<c} \widehat{A}_{c'}$ represents the proper object identifiers of the class type c (excluding the ones of any subtype of c). Such functions either return, for each class type, the last object identifier that has been used for the creation of the objects, or is undefined when no object has been created yet.

For every class type c in S^{C}, the computation of a new last object identifier function starting with an old one is performed by the family of functions $\{\mathit{newloid}_c : \mathit{Loid}_{\mathit{Spec},A} \to \mathit{Loid}_{\mathit{Spec},A} \mid c \in S^{\mathsf{C}}\}$ (new last object identifier) defined as:

$$(\forall c, c' \in S^{\mathsf{C}})(\forall l \in \mathit{Loid}_{\mathit{Spec},A}) \quad \mathit{newloid}_c(l) = l' \text{ such that}$$
$$l'(c') = \begin{cases} \mathit{init}_c^{\widehat{A}} & \text{if } l(c) \text{ is undefined and } c' = c, \\ \mathit{new}_c^{\widehat{A}}(l(c)) & \text{if } l(c) \text{ is defined and } c' = c, \\ l(c) & \text{otherwise.} \end{cases}$$

The second function retained throughout the evolution of the system returns the set of the alive objects of a given class. It is a member of the set of partial functions[1]:

$$\mathit{Aoid}_{\mathit{Spec},A} = \{a : S^{\mathsf{C}} \to C \mid C \subseteq \mathcal{P}(\widehat{A}),\ a(c) \in \mathcal{P}(\widetilde{A}_c)\}.$$

The creation of an object implies the storage of its identity and the computation of a new alive object identifiers function based on the old one. This is achieved by the family of functions $\{\mathit{newaoid}_c : \mathit{Aoid}_{\mathit{Spec},A} \times \widehat{A} \to \mathit{Aoid}_{\mathit{Spec},A} \mid c \in S^{\mathsf{C}}\}$ (new alive object identifiers) defined as:

$$(\forall c, c' \in S^{\mathsf{C}})(\forall o \in \widetilde{A}_c)(\forall a \in \mathit{Aoid}_{\mathit{Spec},A}) \quad \mathit{newaoid}_c(a, o) = a' \text{ such that}$$
$$a'(c') = \begin{cases} a(c) \cup \{o\} & \text{if } c' = c, \\ a(c) & \text{otherwise.} \end{cases}$$

[1]The notation $\mathcal{P}(A)$ represents the *power set* of a set A.

Both of those families of functions $newloid_c$ and $newaoid_c$ are used in the inference rules concerning the creation of new instances.

The set of functions $\{remaoid_c : Aoid_{Spec,A} \times \widehat{A} \rightarrow Aoid_{Spec,A} \mid c \in S^{\mathsf{C}}\}$ is the dual version of the $newaoid_c$ family in the sense that, instead of adding an object identifier, they remove a given object identifier and compute the new alive object identifiers function as follows:

$$(\forall c, c' \in S^{\mathsf{C}})(\forall o \in \widetilde{A}_c)(\forall a \in Aoid_{Spec,A}) \;\; remaoid_c(a, o) = a' \text{ such that}$$
$$a'(c') = \begin{cases} a(c) \setminus \{o\} & \text{if } c' = c, \\ a(c) & \text{otherwise.} \end{cases}$$

This family of functions is necessary when the destruction of class instances is considered.

We now define three operators and a predicate in relation with the last object identifier used and the alive object identifiers functions. These operators and this predicate are used in the inference rules of definition 5.4.6; they have been developed in order to allow simultaneous creation and destruction of objects. For the moment we only provide their formal definition; the explanations related to their meaning and their use is postponed until the informal description of the inferences rules in which they are involved. The first two operators are ternary operators which handle an original last object identifiers function and two other functions. The third binary operator and the predicate handle alive object identifiers functions.

$$\triangle\text{: } Loid_{Spec,A} \times Loid_{Spec,A} \times Loid_{Spec,A} \rightarrow Loid_{Spec,A} \text{ such that}$$
$$(\forall c \in S^{\mathsf{C}}) \; (l' \triangle_l l'')(c) = \begin{cases} l'(c) & \text{if } l'(c) \neq l(c) \wedge l''(c) = l(c), \\ l''(c) & \text{if } l'(c) = l(c) \wedge l''(c) \neq l(c), \\ l(c) & \text{otherwise.} \end{cases}$$

$$\triangleq : Loid_{Spec,A} \times Loid_{Spec,A} \times Loid_{Spec,A} \text{ such that}$$
$$(\forall c \in S^{\mathsf{C}}) \; (l' \triangleq_l l'')(c) = ((l(c) = l'(c) = l''(c)) \vee (l'(c) \neq l''(c)))$$

$$\cup : Aoid_{Spec,A} \times Aoid_{Spec,A} \rightarrow Aoid_{Spec,A} \text{ such that}$$
$$(\forall c \in S^{\mathsf{C}}) \; (a \cup a')(c) = a(c) \cup a'(c)$$

$$P : Aoid_{Spec,A} \times Aoid_{Spec,A} \times Aoid_{Spec,A} \times Aoid_{Spec,A} \text{ such that}$$

$$P(a_1, a_1', a_2, a_2') \iff$$

$$\begin{aligned}(\forall c \in S^{\mathsf{C}}) \quad & (((a_1(c) \cap ((a_2(c) \setminus a_2'(c)) \cup (a_2'(c) \setminus a_2(c)))) = \emptyset) \wedge \\ & ((a_1'(c) \cap ((a_2(c) \setminus a_2'(c)) \cup (a_2'(c) \setminus a_2(c)))) = \emptyset) \wedge \\ & ((a_2(c) \cap ((a_1(c) \setminus a_1'(c)) \cup (a_1'(c) \setminus a_1(c)))) = \emptyset) \wedge \\ & ((a_2'(c) \cap ((a_1(c) \setminus a_1'(c)) \cup (a_1'(c) \setminus a_1(c)))) = \emptyset))\end{aligned}$$

5.3 State Space

In the algebraic nets community, the state of a system corresponds to the notion of marking, that is to say a mapping which returns, for each place of the net, a multi-set of algebraic values. However, this current notion of marking is not suitable in the CO-OPN$_{/2c++}$ context. Remember that CO-OPN$_{/2c++}$ is a structured formalism which allows for the description of a system by means of a collection of entities organized in a hierarchical fashion. In fact, the definitional unit in a CO-OPN specification is the *context* module that possibly coordinates a collection of objects and *context* modules. Also, the collection of objects contained in each *context* module can dynamically increase or decrease in terms of number of entities. We are thus primarily interested in defining the state of a *context module*. In order to do this we will proceed in two steps. Firstly we will define the state of a set of objects which are instances of a set of *class* modules. We will then compose this object state with the state of a set of *context* modules in order to produce the state for an enveloping *context* module. As we will see, this compositional definition will also be useful while defining the semantics of a *context* module. For the subsequent development we often denote the set of ports of a *context* module or set of *context* modules by $Ports(Md)$. We will also denote the set of ports of the *class* modules contained in a specification *Spec* by $ClassPorts(Spec)$.

In terms of object management, the state of a context consists of three elements. The first two ones manage the object identifiers, i.e. a partial function to memorize the last identifiers used, and a second function to memorize which identifiers are created and alive. The third element consists of a *partial* function that associates a multi-set of algebraic values to an object identifier and a place. Such a partial function is undefined when the object identifier is not yet assigned to a created object.

Definition 5.3.1 *Marking, Definition Domain, State of an Object Collection*

Let Spec be a specification, $A = Sem(Pres(Spec))$, $S = sorts(Spec)$ and $P = Places(Spec)$. A marking *is a partial function $m : \widehat{A} \times P \to [A]$ such that if $o \in \widehat{A}$ and $p \in P_s$ with $s \in S$ then $m(o,p) \in [A]_s$. We denote the set of all* *markings over Spec and A by* $Mark_{Spec,A}$. *The* definition domain *of a marking* $m \in Mark_{Spec,A}$ *is defined as*

$$Dom_{Spec,A}(m) = \{(o,p) \mid m(o,p) \text{ is defined}, p \in P, o \in \widehat{A}\}.$$

A marking m *is denoted* $\perp$ *when* $Dom_{Spec,A}(m) = \emptyset$.

The state of an object collection *where the objects are instances of a set of* class *modules belonging to a specification Spec is denoted by* $ObjState_{Spec,A}$. *It corresponds to a triple* $\langle l,a,m\rangle \in Loid_{Spec,A} \times Aoid_{Spec,A} \times Mark_{Spec,A}$.

We now possess all the information required to define the state of a CO-OPN *context* module. In order to do that we need to add to the state of a collection of objects (see definition 5.3.1) the state of a collection of *context* modules contained by the enveloping *context* module.

Definition 5.3.2 *State of a CO-OPN Context*

Let Spec be a well-formed *CO-OPN specification,* $A = Sem(Pres(Spec))$ *and* $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi\rangle$ *be a CO-OPN* context *module belonging to Spec.*

The state *of* Md^{X} *is a pair* $\langle oState, cState\rangle \in CtxState_{Spec,A,Md^{\mathsf{X}}}$ *where:*

- $oState \in ObjState_{Spec,A}$
- $cState = \left\{\left(CtxState_{Spec,A,Md_i^{\mathsf{X}}}\right)_i\right\}_{i\in\{1,\ldots,n\}}$ *is a family of context states where* $Md_i^{\mathsf{X}} \in C$

Notice that definition 5.3.2 is recursive and the stop condition consists of an inner context that either contains only static objects and no contexts, or is empty. Also, the states of sub-contexts are ordered in the same fashion (by the notion of family) they are ordered in the *context* module specification (see definition 4.4.6). Finally, the state *oState* is dynamic in the sense that it can represent any number of instances of objects inside a given *context* module.

The notion of transition system is an essential element of the semantics of a CO-OPN$_{/2c++}$ specification. In the context of the *structured operational semantics* the semantics of a system is expressed by a relation involving a *state*, a *command* (or event) and the new *state* resulting from the effect of the *command* on the original state.

Let us introduce the notion of transition system for a context module, which we need to calculate the semantics of a CO-OPN specification.This transition system will have as states the *context* module state we have defined in 5.3.2 and as events the subset of *coordination* events (see definition 4.4.10) involving only methods and gates of the considered *context* module.

Definition 5.3.3 *Context Transition system*

Let Spec be a well-formed *CO-OPN specification, $A = Sem(Pres(Spec))$ and $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi\rangle$ a CO-OPN* context *module belonging to Spec. Let also $L = Ports(Md^{\mathsf{X}})$, $I = Ports(C)$ and $IO = ClassPorts(Spec)$. A* context transition system *over Spec, A and Md^{X} is a set of triples:*

$$TS_{Spec,A,Md^{\mathsf{X}}} \subseteq CtxState_{Spec,A,Md^{\mathsf{X}}} \times \mathbf{CE}_{A,L,\emptyset,\emptyset,\widehat{A}} \times CtxState_{Spec,A,Md^{\mathsf{X}}}.$$

The set of all transitions systems over Spec, A and Md^{X} is denoted $\mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$. A triple $\langle st, e, st'\rangle$ represents the occurrence of an event e between two states st and st' and is commonly written $st \xrightarrow{e}_{\mathsf{x}} st'$. Due to the generic nature of the definition of coordination events *(see definition 4.4.10) we use the $\emptyset$ parameter to avoid that events internal to the context appear in the* context transition system.

Let us now informally introduce some basic operators on markings and for the management of the object identifiers. These operators will be intensively used in the inference rules designed for the construction of the transition system associated to a given CO-OPN$_{/2c++}$ specification.

Informally, the *sum* of markings '+' adds the multi-set values of two markings and takes into account the fact that markings are partial functions. The *common markings* predicate '$\bowtie$' determines if two markings are equal for their common places. As for the *fusion* of markings '$m_1 \trianglelefteq m_2$', it returns a marking whose the values are those of m_1 and those of m_2 which do not appear in m_1.

Definition 5.3.4 *Sum of markings, common markings, fusion of markings*

Let Spec be a well-formed *CO-OPN specification and $A = Sem(Pres(Spec))$. Let also $S = Sorts(Spec)$ and $P = Places(Spec)$.*
The sum *of two markings is the function: $+ : Mark_{Spec,A} \times Mark_{Spec,A} \to Mark_{Spec,A}$ defined as follows:*

$$(\forall s \in S)\ (\forall p \in P_s)\ (\forall o \in \widehat{A})$$

$$(m_1 + m_2)(o,p) = \begin{cases} m_1(o,p) +^{[A_s]} m_2(o,p) & \text{if } (o,p) \in Dom(m_1) \cap Dom(m_2) \\ m_1(o,p) & \text{if } (o,p) \in Dom(m_1) \setminus Dom(m_2) \\ m_2(o,p) & \text{if } (o,p) \in Dom(m_2) \setminus Dom(m_1) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Common markings *is the predicate:* $\bowtie$: $Mark_{Spec,A} \times Mark_{Spec,A}$ *defined as follows:*

$$\begin{aligned} m_1 \bowtie m_2 \iff & \forall (o,p) \in \widehat{A} \times P \\ & (o,p) \in Dom(m_1) \cap Dom(m_2) \Longrightarrow m_1(o,p) = m_2(o,p) \end{aligned}$$

The fusion of two markings is the function: $\trianglelefteq$: $Mark_{Spec,A} \times Mark_{Spec,A} \rightarrow Mark_{Spec,A}$ *defined as follows:*

$$m_1 \trianglelefteq m_2 = m_3 \textit{ such that } (\forall o \in \widehat{A})$$

$$(m_3)(o,p) = \begin{cases} m_1(o,p) & \textit{if } (o,p) \in Dom(m_1) \\ m_2(o,p) & \textit{if } (o,p) \in Dom(m_2) \backslash Dom(m_1) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Let us also introduce an additional operator and a predicate for managing the part of a *context* state which deals with keeping the states of the contained *context* modules. Again, for the moment we will only provide their formal definition. They will be used in the inference rules to calculate the semantics of a *well-formed* CO-OPN specification and at that point in the text we will introduce their informal meaning. For the definition of these operators consider a specification *Spec*, $A = Sem(Pres(Spec))$ and a family of context modules states $\{Md_i^{\mathsf{X}}\}_{i \in \mathbb{N}}$.

$$\begin{aligned} \triangledown : & \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \times \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \times \\ & \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \rightarrow \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \text{ such that} \\ & S' \triangledown_S S'' = \begin{cases} \bigcup_i s_i \cup \bigcup_j s'_j \cup \bigcup_k s''_k & \text{where } s_i = s'_i = s''_i, s'_j \neq s_j \text{ and } s''_k \neq s_k \\ \text{undefined} & \text{otherwise} \end{cases} \\ & \text{for all } \{s_i, s_j, s_k\} \subseteq S, \{s'_i, s'_j\} \subseteq S' \text{ and } \{s''_i, s''_k\} \subseteq S'' \end{aligned}$$

$$\begin{aligned} Q : & \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \times \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \times \\ & \left\{ \left(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \right)_i \right\}_{i \in \mathbb{N}} \text{ such that} \end{aligned}$$

$$Q(S, S', S'') \iff (\forall s_i \in S, s'_i \in S', s''_i \in S'')\ (s_i \neq s'_i \wedge s_i = s''_i) \vee (s_i \neq s''_i \wedge s_i = s'_i)$$

5.4 Component Semantics

We will now concentrate on defining the semantics of the behavioral part of CO-OPN, which is given by the *class* and *context* modules. Remember that a CO-OPN$_{/2c{+}{+}}$ specification is composed of a set of *context* modules possibly coordinating several other *context* modules and *objects*. On their turn, the internal *context* modules may themselves contain other context modules as we have explained in section 4.4.6. It then seems reasonable to start by providing the semantics of a *context* module by composing the individual semantics of its components. This will allow us to define an inductive step of the definition of the full semantics of a CO-OPN specification.

We will thus proceed in a sequence of steps, which are the following:

1. Defining the semantics of the individual objects inside a *context* module assuming they do not interact with each other. This will be done by building the transition system of the set of possible objects which are instances of the *class* modules contained in a considered CO-OPN$_{/2c{+}{+}}$ specification. The transition system for the possible objects of a CO-OPN$_{/2c{+}{+}}$ specification will be described by a set of inference rules describing the operational semantics of those objects;

2. Adding to the transition system calculated in step 1 the semantics of the *context* modules coordinated by the *context* module we are considering. In fact we do not know these semantics (since we are in fact building the semantics of *context* modules), so we will consider them as known. We may then produce an integrated transition system where the total state includes the individual states of the objects and of the contained *context* modules;

3. Calculating the semantics of the composition of all the possible components inside a *context* module. This composition will be done taking into consideration the coordination expressions which are defined at the level of the enveloping *context* module;

4. Filtering the obtained transition system by only considering events which are accessible from the interface of the enveloping *context* module.

The present method for calculating the semantics of a *context* module is inspired from the work of Buffo in [2] and mostly from the work of Huerzeler in [6]. Huerzeler defines a generic framework for component-oriented formalisms such as CO-OPN$_{/2c{+}{+}}$. We partially reuse this framework, especially in what concerns the composition of the individual behaviors of the components inside a *context* module.

First of all, we provide some auxiliary definitions used in the subsequent construction of the semantics of *class* and *context* modules. Let us consider a specification *Spec*, $A = \mathit{Sem}(\mathit{Pres}(\mathit{Spec}))$, a class module $Md^{\mathsf{C}} = \langle \Omega^{\mathsf{C}}, P, I, X, \Psi \rangle$ of type c belonging to *Spec*. Let $S^{\mathsf{A}} = \mathit{sorts}(\mathit{Spec})$, $S^{\mathsf{C}} = \mathit{types}(\mathit{Spec})$, $\langle M, G \rangle = \mathit{classPorts}(\mathit{Spec})$, and Σ be the global signature of *Spec*.

The evaluation of a set of terms of $T_{[\Sigma],X}$ indexed by P for a given assignment σ and a given class instance o into the set of markings $\mathit{Mark}_{\mathit{Spec},A}$ is defined as:

$$\llbracket (t_p)_{p \in P} \rrbracket^{\sigma}_{o} = m \text{ such that } (\forall p \in \mathit{Places}(\mathit{Spec}))(\forall o' \in \widehat{A})$$
$$m(o', p) = \begin{cases} \llbracket t_p \rrbracket^{\sigma} \text{ if } o' = o \text{ and } p \in P, \\ \text{undefined otherwise.} \end{cases}$$

Such terms form, for example, a pre/post condition of a behavioral formula or an initial marking.

Another kind of evaluation required by the inference rules is the evaluation of an event which consists of the evaluation of all the arguments of the methods, but also the evaluation of the object identifier terms.

The synchronization evaluation $\llbracket\ \rrbracket^{\sigma} : \mathit{ObjSyncExpr}_{(T_{\Sigma,X}),\{c\},\langle M,G \rangle,(T_{\Sigma,X})_s} \rightarrow \mathit{ObjSyncExpr}_{A,\{c\},\langle M,G \rangle,\widehat{A}}$ is inductively defined as:

$$\begin{aligned}
\llbracket t.\mathit{create} \rrbracket^{\sigma} &= \llbracket t \rrbracket^{\sigma}.\mathit{create} \\
\llbracket t.\mathit{destroy} \llbracket^{\sigma} &= \llbracket t \rrbracket^{\sigma}.\mathit{destroy} \\
\llbracket t.m(v_1, \ldots, v_n) \rrbracket^{\sigma} &= \llbracket t \rrbracket^{\sigma}.m(\llbracket v_1 \rrbracket^{\sigma}, \ldots, \llbracket v_n \rrbracket^{\sigma}) \\
\llbracket t.g(v_1, \ldots, v_n) \rrbracket^{\sigma} &= \llbracket t \rrbracket^{\sigma}.g(\llbracket v_1 \rrbracket^{\sigma}, \ldots, \llbracket v_n \rrbracket^{\sigma}) \\
\llbracket \mathit{sync}\ \mathit{op}\ \mathit{sync}'' \rrbracket^{\sigma} &= \llbracket \mathit{sync} \rrbracket^{\sigma}\ \mathit{op}\ \llbracket \mathit{sync}' \rrbracket^{\sigma}
\end{aligned}$$

The *object event* evaluation $\llbracket\ \rrbracket^{\sigma} : \mathsf{OE}_{(T_{\Sigma,X}),\{c\},\langle M,G \rangle,(T_{\Sigma,X})_s} \rightarrow \mathsf{OE}_{A,\{c\},\langle M,G \rangle,\widehat{A}}$ naturally follows from definition 4.4.6 and is inductively defined as:

$$\begin{aligned}
\llbracket t.m(v_1, \ldots, v_n) \rrbracket^{\sigma} &= \llbracket t \rrbracket^{\sigma}.m(\llbracket v_1 \rrbracket^{\sigma}, \ldots, \llbracket v_n \rrbracket^{\sigma}) \\
\llbracket t.m(v_1, \ldots, v_n) \textbf{ with } \mathit{sync} \rrbracket^{\sigma} &= \llbracket t.m(v_1, \ldots, v_n) \rrbracket^{\sigma} \textbf{ with } \llbracket \mathit{sync} \rrbracket^{\sigma} \\
\llbracket \mathit{ev} \textbf{ with } \mathit{ev}' \rrbracket^{\sigma} &= \llbracket \mathit{ev} \rrbracket^{\sigma} \textbf{ with } \llbracket \mathit{ev}' \rrbracket^{\sigma}
\end{aligned}$$

for all:

- $\mathit{ev}, \mathit{ev}' \in \mathsf{OE}_{(T_{\Sigma,X}),\{c\},\langle M,G \rangle,(T_{\Sigma,X})_s}$ with $s \in S^{\mathsf{C}}$

- $sync \in ObjSyncExpr_{(T_{\Sigma,X}),\{c\},\langle M,G\rangle,(T_{\Sigma,X})_s}$ with $s \in S^{\mathsf{C}}$
- $t \in (T_{\Sigma,X})_s$ with $s \in S^{\mathsf{C}}$
- $m \in M_{s,s_1,\ldots,s_n}$ with $s \in S^{\mathsf{C}}$ and $s_i \in S^{\mathsf{A}}$
- $v_1, \ldots, v_n \in (T_{\Sigma,X})_s$ with $s \in S$
- synchronization operators $op \in \{//, .., \oplus\}$

Note that the evaluation of any term t of $(T_{\Sigma,X})_s$ with $s \in S^{\mathsf{C}}$ belongs to $\widehat{A}$ and then represents an object identifier. The evaluation of such terms, in the previous definition, is essential when data structures of object identifiers are considered.

We need a similar evaluation function for *coordination events.* Consider a context module $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi \rangle$ belonging to specification *Spec* having a set of local ports L, a set of internal context ports I and a set of internal object ports IO. Similarly to object event evaluation, the coordination event evaluation is derived from definition 4.4.10. For space and clarity reasons we will only provide in this text its signature. The formal definition is analogous the one for *object events*:

$$[\![\ ]\!]^{\sigma} : \mathbf{CE}_{(T_{\Sigma,X}),L,I,IO,(T_{\Sigma,X})_s} \to \mathbf{CE}_{A,L,I,IO,\widehat{A}} \ \text{ with } s \in S^{\mathsf{C}}$$

Finally, the satisfaction of a condition of a behavioral or coordination formula is defined as:

$$A, \sigma \models Cond \iff (Cond = \emptyset) \vee (\forall (t = t') \in Cond \,.\, A, \sigma \models (t = t')).$$

5.4.1 Class Modules

We now develop the *partial semantics* of a class module in a CO-OPN specification. We call it *partial* because at this point we only take into consideration the events induced by *behavioral formulas* and not the possible *simultaneous, sequential* or *alternative* synchronizations of those events. Also, we do not address the interaction between the possible object instances. This interaction will be later addressed when we will perform the composition induced by the synchronizations defined at the level of the enveloping *context module* by means of *coordination formulas.*

In figure 5.1 we depict the intuition behind the formal concept of *class semantics.* We wish to produce a transition system which will encompass all the possible instances of the classes present in a CO-OPN specification. The difficulty resides in the fact that the number of objects is dynamic given that each class implicitly possesses a "*static*" *create* and *destroy* method. Figure 5.1 presents a state of the

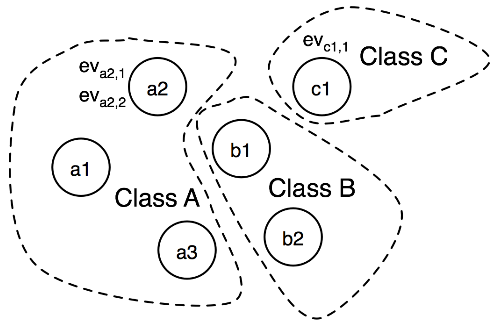


Figure 5.1: Class Semantics

transition system we wish to produce. The state is composed of six instances of classes *A, B* and *C*. We also include in the figure some events which are possible from that state. It is important to notice that the state of an *object collection* we have introduced in definition 5.3.1 is able to encompass a dynamic amount of *class instances*.

Let us now define the semantics of a single *class module*. The semantics of all *class modules* will be produced by uniting all transition systems obtained for all *class modules*.

Definition 5.4.1 *Partial semantics of a class module*

Let Spec be a specification and $A = Sem(Pres(Spec))$*. Let* $Md^{\mathsf{C}} = \langle \Omega^{\mathsf{C}}, P, I, X, \Psi \rangle$ *be a class module of Spec, where* $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, G \rangle$*. The semantics of* Md^{C} *is the* $PSem_{Spec,A}(Md^{\mathsf{C}})$ *transition system (noted* $\rightarrow_{\mathsf{cl}}$*) which is the least fixed point resulting from the application of the inference rules:* Class*,* Mono*,* Create*, and* Destroy *defined in the following rules:*

$$Class \quad \frac{\begin{array}{l} Event :: Cond \Rightarrow Pre \rightarrow Post \in \Psi,\ \exists \sigma : X \rightarrow A, \\ A, \sigma \models Cond,\ o \in a(c) \end{array}}{\langle l, a, [\![Pre]\!]^{\sigma}_{o} \rangle \xrightarrow{[\![Event]\!]^{\sigma}}_{\mathsf{cl}} \langle l, a, [\![Post]\!]^{\sigma}_{o} \rangle}$$

$$Create \quad \frac{\begin{array}{l} \exists \sigma : X \rightarrow A, \\ l' = newloid_c(l),\ a' = newaoid_c(a, o),\ o = l'(c),\ o \notin a(c) \end{array}}{\langle l, a, \bot \rangle \xrightarrow{o.create}_{\mathsf{cl}} \langle l', a', [\![I]\!]^{\sigma}_{o} \rangle}$$

$$Destroy \quad \frac{o \in a(c),\ a' = remaoid_c(a, o)}{\langle l, a, \bot \rangle \xrightarrow{o.destroy}_{\mathsf{cl}} \langle l, a', \bot \rangle}$$

$$\mathit{Mono}\ \frac{\langle l,a,m\rangle \xrightarrow{e}_{\mathsf{cl}} \langle l',a',m'\rangle}{\langle l,a,m+m''\rangle \xrightarrow{e}_{\mathsf{cl}} \langle l',a',m'+m''\rangle}$$

where $l,l' \in \mathit{Loid}_{\mathit{Spec},A}$, $a,a' \in \mathit{Aoid}_{\mathit{Spec},A}$, $m,m',m'' \in \mathit{Mark}_{\mathit{Spec},A}$, $o \in \widetilde{A}_c$ *and* $e \in \mathbf{OE}_{A,\{c\},\langle M,G\rangle,\widehat{A}}$.

The inference rules introduced in definition 5.4.1 can be informally formulated as follows:

- The *Class* rule generates the basic transitions that follow from the behavioral formulas of a class. For all the object identifiers of the class, for all last object identifier function l, and for all alive object identifier function a, a *firable* (or *enabled*) transition is produced provided that:

  1. there is a behavioral formula $\mathit{Event} :: \mathit{Cond} \Rightarrow \mathit{Pre} \rightarrow \mathit{Post}$ in the class;
  2. there exists an assignment $\sigma : X \rightarrow A$;
  3. all the equations of the global condition are satisfied ($A,\sigma \models \mathit{Cond}$);
  4. the object o has already been created and is still alive, i.e. it belongs to the set of alive objects of the class ($o \in a(c)$).

  The transition generated by the rule guarantees that there are enough values in the respective places of the object. The firing of the transition consumes and produces the values as established in the pre-set and post-set of the behavioral formula.

- The *Create* rule generates the transitions aimed at the dynamic creation of new objects provided that:

  1. for any last object identifier function l and any alive object identifier function a;
  2. a new last object identifier function is computed ($l' = \mathit{newloid}_c(l)$);
  3. a new object identifier o is determined for the class ($o = l'(c)$);
  4. this new object identifier must not correspond to any active object ($o \notin a(c)$).

  The new state of the transition generated by the rule is composed of the new last object identifier function l', an updated function a' in which the new object identifier has been added to the set of created objects of the class, and the initial marking $[\![I]\!]_o^\sigma$.

- The *Destroy* rule, aimed at the destruction of objects, is similar to the *Create* rule. The *Destroy* rule merely takes an object identifier out of the set of created objects, provided that the object is alive.
- The *Mono* rule (for monotonicity) generates all the firable transitions from the transitions already generated. This rule allows building the state so that we take into consideration all the possible markings of the objects that allow an event of the class to occur.

5.4.2 Composition Semantics for Context Modules

Let us now proceed to the composition of the semantics of the components (objects and context modules) inside a considered *context* module. This will be achieved by composing the individual semantics of those components given a set of coordination formulas defined at the level of the considered *context* module. As for the individual components, the semantics of the composition will also be given as a *transition system* where the states correspond to the states of all the involved components and the labels correspond to the events possible in that composition. Given a set of *objects* and *context* modules let us start by defining the auxiliary notion of *composed events.*

Definition 5.4.2 *Composed Gate Synchronization Expression, Composed Method Synchronization Expression, Composed Events*

Let Spec be a well-formed *CO-OPN specification,* $\Sigma = \langle S, \leq, F\rangle$ *be the global signature of Spec,* $A = Sem(Pres(Spec))$ *and* $C \subseteq S$ *be a set of class type names. Let also* $M = (M_{c,w})_{w\in S^*} \cup (M_w)_{w\in S^*}$ *be a set of class and context* method *ports and* $G = (G_{c,w})_{w\in S^*} \cup (G_w)_{w\in S^*}$ *be a set of class and context* gate *ports where* $c \in C$*.*

The composed gate synchronization expressions $GateExpr_{A,C,G}$ *are built as follows:*

- $g(v_1, \ldots, v_n) \in GateExpr_{A,C,G}$ *for all* $g \in (G_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$
- $o.g(v_1, \ldots, v_n) \in GateExpr_{A,C,G}$ *for all* $g \in (G_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in A_c$
- $exp\ op\ exp' \in GateExpr_{A,C,G}$ *for all* $exp, exp' \in GateExpr_{A,C,G}$, $op \in \{//, .., \oplus\}$

The composed method synchronization expressions $MethodExpr_{A,C,G}$ *are built analogously to* $GateExpr_{A,C,G}$*. The set of composed events* $CompEv_{A,C,M,G}$ *is recursively built with the sets* $GateExpr_{A,C,G}$ *and* $MethodExpr_{A,C,M}$ *in the following fashion:*

- $m(v_1, \ldots, v_n) \in CompEv_{A,C,M,G}$ *for all* $m \in (M_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$
- $m(v_1, \ldots, v_n)$ **with** $exp \in CompEv_{A,C,M,G}$ *for all* $m \in (M_{s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $exp \in GateExpr_{A,C,G}$
- $o.m(v_1, \ldots, v_n) \in CompEv_{A,C,M,G}$ *for all* $m \in (M_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in A_c$
- $o.m(v_1, \ldots, v_n)$ **with** $exp \in CompEv_{A,C,M,G}$ *for all* $m \in (M_{c,s_1,\ldots,s_n})$, $v_i \in A_{s_i}$, $o \in A_c$, $exp \in GateExpr_{A,C,G}$
- $ev\ op\ ev' \in CompEv_{A,C,M,G}$ *for all* $ev, ev' \in CompEv_{A,C,M,G}$, $op \in \{//, .., \oplus\}$

In definition 5.4.2 we build the set of all possible events while composing the behavior of a set of *class* and *context* modules. Notice that the dynamic aspect of class modules is taken into consideration by the fact that we consider the object identifiers (belonging to the object identifier algebras of the considered types) while building the synchronizations. *Composed events* are similar to *object events* (see definition 4.4.6) and *coordination events* (see definition 4.4.10), the difference being that in this case we consider all the possible events resulting from the interaction of *objects* and *context* modules.

Composition of Event Names

In order to compose a set of components we have to take into consideration the *coordination events* connecting those components. These connections allow linking the transition systems of the individual components in order to form the composed transition system. The calculation of the composed transition system involves two aspects: firstly, the individual components need to hold enough resources to satisfy the synchronization expressions in the *coordination events*; secondly, the labels of the composed transition system — which identify the *events* of the composed transition system — will be formed by composing the event names of the individual component transition systems using the *coordination events*. Let us start by the problem of composing event names.

Consider the set of components depicted in figure 5.2. The boxes represent arbitrary components (objects or *context* modules) with their method and gate ports. The dashed lines inside the components represent internal synchronizations between *method* and *gate* ports. Notice that for the sake of explanation we abstract the representation of the resources which would be associated to the places of the Petri Nets encapsulated by the components. The full lines between the components indicate the synchronizations connecting components. A possible composed event name from the example of figure 5.2 would be:

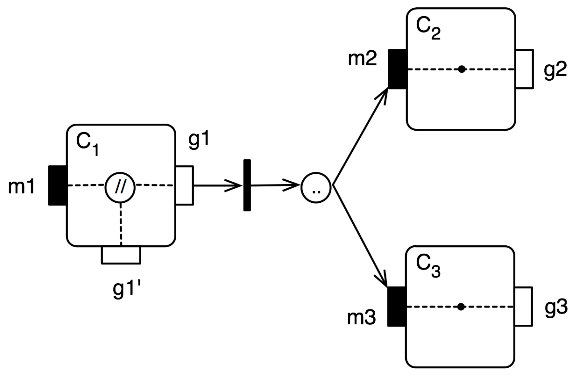


Figure 5.2: Component Composition

$$m1 \; with \; (g2 \,..\, g3) \; // \; g1'$$

Notice that the name of this event is built by performing a kind of transitive closure of the connections between components. Informally, in order to build the name of this event we need to compose some events from components C_1, C_2 and C_3 using the coordination event $g1 \; with \; (m2 \,..\, m3)$, as follows:

$$\left.\begin{array}{ll} C_1: & m1 \; with \; (g1 \,//\, g1') \\ C_2: & m2 \; with \; g2 \\ C_3: & m3 \; with \; g3 \\ Coordination: & g1 \; with \; (m2 \,..\, m3) \end{array}\right\} \; m1 \; with \; (g2 \,..\, g3) \,//\, g1'$$

The main difficulty amounts then to, given a method port synchronization, calculating the remainder gate synchronization taking into consideration the transitivity induced by the coordination events. In order to do this, we will start by building an exhaustive relation that, for a method synchronization and an event name involving that method synchronization allows us to know the remainder gate synchronization.

Definition 5.4.3 *Relation MethodEvRem*

Let Spec be a well-formed *CO-OPN specification, $\Sigma = \langle S, \leq, F \rangle$ be the global signature of Spec, $A = Sem(Pres(Spec))$ and $C \subseteq S$ be a set of class type names. Let also $M = (M_{c,w})_{w \in S^*} \cup (M_w)_{w \in S^*}$ be a set of class and context* method *ports and $G = (G_{c,w})_{w \in S^*} \cup (G_w)_{w \in S^*}$ be a set of class and context* gate *ports where $c \in C$. The relation $MethodEvRem_{A,C,M,G} \subseteq MethodExpr_{A,C,M} \times CompEv_{A,C,M,G} \times$*

$GateExpr_{A,C,G}$ is defined as follows. For readability we note $x \leftarrow y/_z$ for $\langle x, y, z\rangle$ in the relation:

$$ctxSimpleMethodSync \ \frac{m \in (M_{s_1,\ldots,s_n}), v_i \in A_{s_i}}{m(v_1,\ldots,v_n) \leftarrow m(v_1,\ldots,v_n)/_\epsilon}$$

$$ctxMethodSync \ \frac{m \in (M_{s_1,\ldots,s_n}), v_i \in A_{s_i}, z \in GateExpr_{A,C,G}}{m(v_1,\ldots,v_n) \leftarrow m(v_1,\ldots,v_n) \ \textbf{\textit{with}} \ z/_z}$$

$$objSimpleMethodSync \ \frac{m \in (M_{c,s_1,\ldots,s_n}), v_i \in A_{s_i}, o \in A_c}{o.m(v_1,\ldots,v_n) \leftarrow o.m(v_1,\ldots,v_n)/_\epsilon}$$

$$objSimpleMethodSync \ \frac{m \in (M_{c,s_1,\ldots,s_n}), v_i \in A_{s_i}, o \in A_c, z \in GateExpr_{A,C,G}}{o.m(v_1,\ldots,v_n) \leftarrow o.m(v_1,\ldots,v_n) \ \textbf{\textit{with}} \ z/_z}$$

$$compOp \ \frac{x \leftarrow y/_z, x' \leftarrow y'/_{z'}, z \neq \epsilon, z' \neq \epsilon, op \in \{//, .., \oplus\}}{x \ op \ x' \leftarrow y \ op \ y'/_{z \ op \ z'}}$$

$$compEmptyOp1 \ \frac{x \leftarrow y/_z, x' \leftarrow y'/_\epsilon, op \in \{//, .., \oplus\}}{x \ op \ x' \leftarrow y \ op \ y'/_z}$$

$$compEmptyOp2 \ \frac{x \leftarrow y/_z, x' \leftarrow y'/_\epsilon, op \in \{//, .., \oplus\}}{x' \ op \ x \leftarrow y' \ op \ y/_z}$$

where $x \in MethodExpr_{A,C,M}$, $y \in CompEv_{A,C,M,G}$ and $x \in GateExpr_{A,C,G}$.

The first four inference rules in definition 5.4.3 are trivial and state that: a method synchronization without a gate synchronization has no remaining gate synchronization; an method synchronization with a gate synchronization has that gate synchronization as remainder. Note that we treat both *object* and *context* module components. The "*compOp*" rule treats composed method synchronizations using the usual synchronizations operators and propagate those operators to the remainder. The last two inference rules treat the case where one of the method synchronizations has no remainder and does not take it into consideration in the composed remainder.

Definition 5.4.3 gives us all the possible event names in a composed system, but does not take into consideration the *coordination events*. Taking the relation *MethodEvRem*, let us now build a relation *GateEvRem* that, given a gate synchronization and an event name allows knowing the remainder gate synchronization that results from coordinating the gate synchronization with the event name.

Definition 5.4.4 *Relation GateEvRem*

Let Spec be a well-formed *CO-OPN specification,* $\Sigma = \langle S, \leq, F \rangle$ *be the global signature of Spec,* $A = Sem(Pres(Spec))$ *and* $C \subseteq S$ *be a set of class type names. Let also* $M = (M_{c,w})_{w \in S^*} \cup (M_w)_{w \in S^*}$ *be a set of class and context* method *ports and* $G = (G_{c,w})_{w \in S^*} \cup (G_w)_{w \in S^*}$ *be a set of class and context* gate *ports where* $c \in C$*. Finally let* $ce \in \mathbf{CE}_{A,L,I,IO,\widehat{A}}$ *be a set of coordination events. The relation* $GateEvRem_{A,C,M,G,ce} \subseteq MethodExpr_{A,C,M} \times CompEv_{A,C,M,G} \times GateExpr_{A,C,G}$ *is defined as follows. For readability we write* $x \Leftarrow y/_z$ *for* $\langle x, y, z \rangle$ *in the relation:*

$$sync \ \frac{g \ \textbf{\textit{with}} \ x \in ce, x \leftarrow y/_z}{g \Leftarrow y/_z}$$

$$compop \ \frac{v \Leftarrow y/_z, v' \Leftarrow y'/_{z'}, z \neq \epsilon, z' \neq \epsilon, op \in \{//, .., \oplus\}}{v \ op \ v' \Leftarrow y \ op \ y'/_{z \ op \ z'}}$$

$$compEmptyOp1 \ \frac{v \Leftarrow y/_z, v' \Leftarrow y'/_\epsilon, op \in \{//, .., \oplus\}}{v \ op \ v' \Leftarrow y \ op \ y'/_z}$$

$$compEmptyOp2 \ \frac{v \Leftarrow y/_z, v' \Leftarrow y'/_\epsilon, op \in \{//, .., \oplus\}}{v' \ op \ v \Leftarrow y' \ op \ y/_z}$$

$$enrich1 \ \frac{v \Leftarrow y/_z, z \neq \epsilon, op \in \{//, .., \oplus\}, v' \in GateExpr_{A,C,G}}{v \ op \ v' \Leftarrow y/_{z \ op \ v'}}$$

$$enrich2 \ \frac{v \Leftarrow y/_z, z \neq \epsilon, op \in \{//, .., \oplus\}, v' \in GateExpr_{A,C,G}}{v' \ op \ v \Leftarrow y/_{v' \ op \ z}}$$

$$emptyEnrich1 \ \frac{v \Leftarrow y/_\epsilon, op \in \{//, .., \oplus\}, v' \in GateExpr_{A,C,G}}{v \ op \ v' \Leftarrow y/_{v'}}$$

$$emptyEnrich2 \ \frac{v \Leftarrow y/_z, z \neq \epsilon, op \in \{//, .., \oplus\}, v' \in GateExpr_{A,C,G}}{v' \ op \ v \Leftarrow y/_{v'}}$$

where $x \in MethodExpr_{A,C,M}$*,* $y \in CompEv_{A,C,M,G}$ *and* $x \in GateExpr_{A,C,G}$*.*

The first inference rule in definition 5.4.4 states that if we coordinate a gate synchronization with an event name having a given remainder, the remainder of the coordination is the same. An application of this rule to the example presented in figure 5.2 can be seen in the following example:

$$\left.\begin{array}{ll} \mathit{Event}: & (m2\mathinner{..}m3)\ \mathit{with}\ (g2\mathinner{..}g3) \\ \mathit{Coordination}: & g1\ \mathit{with}\ (m2\mathinner{..}m3) \end{array}\right\}\ g1\ \mathit{with}\ (g2\mathinner{..}g3)$$

As in the *MethodEvRem* relation, the "*compOp*" rule in the definition of the *GateEvRem* relation allows composing several gate synchronizations coordinated with event names by using the known synchronization operators. As previously, the last argument of the relation includes the propagation of the operators to the remainder gate synchronization. The *"compEmptyOp1"* and *"compEmptyOp2"* also perform composition, but do not take into consideration empty remainders.

Finally, the last four *enrich* rules allow propagating additional gate synchronizations to the already formed coordinations of of gate synchronizations with events. The usefulness of this notion will become clearer in the text that follows. For the time being let us exemplify again with the composition shown in figure 5.2:

$$\left.\begin{array}{ll} \mathit{Coordinated\ event}: & g1\ \mathit{with}\ (g2\mathinner{..}g3) \\ \mathit{Gate\ Synchronization}: & g1' \end{array}\right\}\ g1//g1'\ \mathit{with}\ (g2\mathinner{..}g3)//g1'$$

At this point we have described a fashion of composing event names that we will use as labels while building the transition system representing the semantics of a composed system. We now need to associate the fashion in which the event names are built to the state of the composed system, i.e. to the available resources.

Partial Semantics of the Composed Modules

In section 5.4.1 we have introduced the semantics for a *class* module. With the set of rules in definition 5.4.1 we have the transition system representing the individual behavior of each of the objects instances of a given *class* module. We would now like to to build a *unified transition system* including not only all the possible transition systems of the potential individual objects inside a an enveloping context, but also the transition systems for the unified *contexts* modules inside that enveloping context. We call this transition system the *unified partial semantics* because it does not take yet into consideration compositional aspects.

In figure 5.3 we present the intuition behind the formal notion of *unified transition system*. We take transition system produced by all the *class modules* in a specification (see section 5.4.1) and augment it by adding the transition systems of a set of *context modules*. We do not yet take into consideration any interactions between any *context modules* or *class instances* and consider that they evolve independently. Figure 5.3 can be seen as a snapshot of the *unified transition system*, i.e

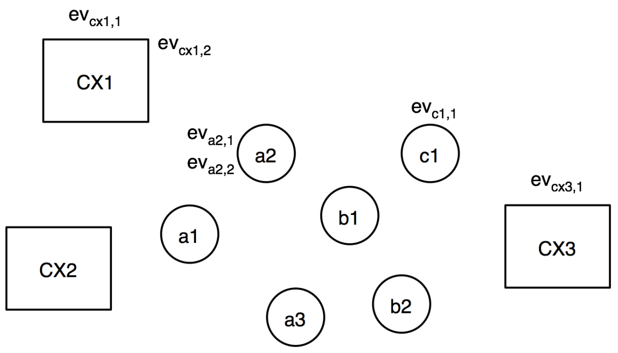


Figure 5.3: Unified Partial Semantics

it represents a state including both *context modules* and *class instances* as well as some *events* possible from that state.

Definition 5.4.5 *Unified Partial Semantics*

Let Spec be a well-formed *CO-OPN specification,* $A = Sem(Pres(Spec))$ *and* $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi\rangle$ *be a CO-OPN* context *module belonging to Spec. Let also* $S^{\mathsf{C}} = Types(Spec)$, $\langle M, G\rangle = Ports(C) \cup ClassPorts(Spec)$, $Md^{\mathsf{C}}_j \in Spec$ $(0 \le j \le n)$ *be a set of class modules and* $Md^{\mathsf{X}}_i \in C$ $(0 \le i \le m)$ *a set of context modules coordinated by* Md^{X}*. Finally let* $\bigcup_j PSem_{Spec,A}(Md^{\mathsf{C}}_j)$ *be the transition system (noted* $\rightarrow_{\mathsf{cl}}$*) which results from the union of the partial semantics of the* class *modules of Spec and* $Sem_{Spec,A}(Md^{\mathsf{X}}_i)$ *be the transition system (noted* $\rightarrow_{\mathsf{x_i}}$*) which represents the semantics of context module* $Md^{\mathsf{X}}_i \in C$.

The unified partial semantics *for* Md^{X} *is the transition system* $UPS_{Spec,A}(Md^{\mathsf{X}}) \subseteq CtxState_{Spec,A,Md^{\mathsf{X}}} \times CompEv_{A,C,M,G} \times CtxState_{Spec,A,Md^{\mathsf{X}}}$ *(noted* $\rightarrow_{\mathsf{u}}$*) defined by the following rules:*

$$includeClassEv \quad \frac{os \xrightarrow{e}_{\mathsf{cl}} os'}{\langle CS, os\rangle \xrightarrow{e}_{\mathsf{u}} \langle CS, os'\rangle}$$

$$includeContextEv \quad \frac{cs_i \xrightarrow{e}_{\mathsf{x_i}} cs'_i}{\langle CS^{i-} \cup \{cs_i\}, os\rangle \xrightarrow{e}_{\mathsf{u}} \langle CS^{i-} \cup \{cs'_i\}, os\rangle}$$

where

- $e, e' \in CompEv_{A,S^{\mathsf{C}},M,G}$, $op \in \{//, .., \oplus\}$
- $os, os' \in ObjState_{Spec,A}$, $cs_i, cs'_i \in CtxState_{Spec,A,Md^{\mathsf{X}}_i}$

- $CS \in \left\{\left(CtxState_{Spec,A,Md_i^{\mathsf{X}}}\right)_i\right\}_{i\in\mathbb{N}}$ *where* $Md_i^{\mathsf{X}} \in C$
- $CS^{k^-} \in \left\{\left(CtxState_{Spec,A,Md_i^{\mathsf{X}}}\right)_i\right\}_{i\in\mathbb{N}} \backslash \left\{CtxState_{Spec,A,Md_k^{\mathsf{X}}}\right\}$ *where* $Md_i^{\mathsf{X}} \in C$ *and* $k \in \mathbb{N}$

Notice that definition 5.4.5 performs a kind of "*cartesian product*" between the semantics of the *object collection* and the *context modules.*

Closure Operation

In the definition of *partial unified semantics* (see definition 5.4.5) we have considered the behavior of a system including a set of *class* and *context* modules. However we have considered both the class instances (*objects*) and the *context modules* separately. We now need a means of calculating the possible *simultaneous*, *sequential* and *alternative* behaviors given a transition system involving the behavior of a set of *objects* and *context* modules.

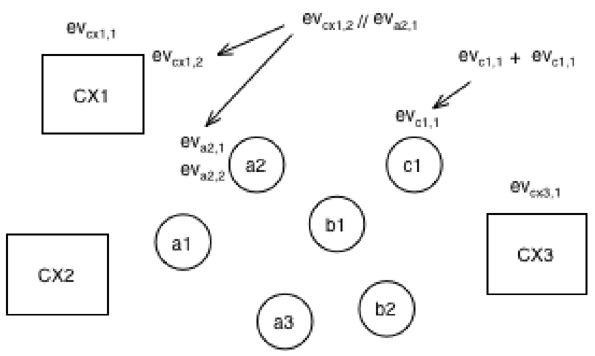


Figure 5.4: Closure Operation

Figure 5.4 can be seen as the "next step" after the snapshot of a state of the example *unified partial semantics* transition system that can be observed in figure 5.3. The intuition behind the *closure* operation corresponds to taking the possible *events* for a state of the *unified partial semantics* transition system and producing their possible synchronizations according to the typical operators. The possibility of synchronizing two *events* depends on the resource availability in the state we are considering. For the *simultaneous synchronization* of two *events* we demand that is no conflict between the resources required both events. For the *sequential synchronization* of two *events* we demand that the resources required by the second *event* are available after the first event occurs. For the *alternative* *synchronization* we demand that resources are available for either one or the other *event*.

Definition 5.4.6 *Closure Operation*

Let Spec be a specification, $A = Sem(Pres(Spec))$ *and* $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi \rangle$ *be a CO-OPN* context *module belonging to Spec. The* closure operation *consists of the function* $Closure : \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}} \to \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$ *such that* $Closure(TS)$ *is the least fixed point which results from the application on* TS *of the inference rules* Seq, Sim, Alt1 *and* Alt2 *which are defined as follows:*

$$Seq \quad \frac{\begin{array}{l} m'_1 \bowtie m_2, \\ \big\langle S, \langle l, a_1, m_1 \rangle \big\rangle \xrightarrow{e_1} \big\langle S', \langle l', a'_1, m'_1 \rangle \big\rangle,\ \big\langle \langle S', l', a'_2, m_2 \rangle \big\rangle \xrightarrow{e_2} \big\langle S'', \langle l'', a'_2, m'_2 \rangle \big\rangle \end{array}}{\big\langle S, \langle l, a, m_1 \unlhd m_2 \rangle \big\rangle \xrightarrow{e_1 \,..\, e_2} \big\langle S'', \langle l'', a'_2, m'_2 \unlhd m'_1 \rangle \big\rangle}$$

$$Sim \quad \frac{\begin{array}{l} l' \triangleq_l l'',\ P(a_1, a'_1, a_2, a'_2),\ Q(S, S', S''), \\ \big\langle S, \langle l, a_1, m_1 \rangle \big\rangle \xrightarrow{e_1} \big\langle S', \langle l', a'_1, m'_1 \rangle \big\rangle,\ \big\langle S, \langle l', a'_2, m_2 \rangle \big\rangle \xrightarrow{e_2} \big\langle S'', \langle l'', a'_2, m'_2 \rangle \big\rangle \end{array}}{\big\langle S, \langle l, a_1 \cup a_2, m_1 + m_2 \rangle \big\rangle \xrightarrow{e_1 \,//\, e_2} \big\langle S' \nabla_S S'', \langle l' \vartriangle_l l'', a'_1 \cup a'_2, m'_1 + m'_2 \rangle \big\rangle}$$

$$Alt1 \quad \frac{\big\langle S, \langle l, a, m \rangle \big\rangle \xrightarrow{e_1} \big\langle S', \langle l', a', m' \rangle \big\rangle}{\big\langle S, \langle l, a, m \rangle \xrightarrow{e_1 \oplus e_2} \big\langle S', \langle l', a', m' \rangle}$$

$$Alt2 \quad \frac{\big\langle S, \langle l, a, m \rangle \big\rangle \xrightarrow{e_1} \big\langle S', \langle l', a', m' \rangle \big\rangle}{\big\langle S, \langle l, a, m \rangle \big\rangle \xrightarrow{e_2 \oplus e_1} \big\langle S', \langle l', a', m' \rangle \big\rangle}$$

where

- m_1, m'_1, m_2, m'_2 *in* $Mark_{Spec,A}$ *and* l, l', l'' *in* $Loid_{Spec,A}$
- $a, a', a_1, a'_1, a_2, a'_2$ *in* $Aoid_{Spec,A}$ *and* e_1, e_2 *in* $CompEv_{A,S^{\mathsf{C}},M,G}$
- $S, S', S'' \in \big\{ \big(CtxState_{Spec,A,Md_i^{\mathsf{X}}} \big)_i \big\}_{i \in \mathbb{N}}$ *where* $Md_i^{\mathsf{X}} \in C$

The inference rules of definition 5.4.6 can be informally formulated as follows:

- The *Seq* rule infers the sequence of two transitions provided that the markings shared between m'_1 and m_2 are equal.

- The *Sim* rule infers the simultaneity of two transitions, provided that some constraints on the l and a functions are satisfied. The purposes of these constraints are:

  1. to avoid that an event can use a given object being created by the other event (i.e. which does not already exist);
  2. to avoid that an event can use a given object being destroyed by the other event (i.e. which does not exit any more).

  Informally, the operators defined in section 5.2 are used to:

  1. $l' \overset{\triangle}{=}_l l''$ avoids the conflicts when simultaneous creation is considered. Remember that the assignment of a new object identifier is handled by a different function for each type. Consequently, this characteristic does not permit the simultaneous creation of two objects of the same type.
  2. $l' \ \triangle_l \ l''$ combines the last object identifier functions according to the creations involved in e_1 and e_2;
  3. $a \cup a'$ makes merely the union of the a and a' for each type;
  4. the predicate $P(a_1, a'_1, a_2, a'_2)$ guarantees that the objects created or destroyed by the events e_1 do not appear in the upper tree related to the event e_2 and vice versa; more precisely, for each type c the active objects of $a_1(c)$ (and $a'_1(c)$) and the "difference" between $a_2(c)$ and $a'_2(c)$ have to be disjoint, as well as the active objects of $a_2(c)$ (and $a'_2(c)$) and the "difference" between $a_1(c)$ and $a'_1(c)$.

- The *Alt1* and *Alt2* rules provides all the alternative behaviors. Two rules are necessary for representing both choices of the alternative operator '$\oplus$'.

In consequence, several intuitive but important intended events can never occur in a system that is built by means of such formal system. These are :

1. the use of an object followed by the creation of this object;
2. the destruction of an object followed by the use of this object;
3. the creation (or destruction) of an object and the simultaneous use of this object;
4. the creation (or destruction) of an object and the simultaneous creation (or destruction) of another object of the same type;
5. the synchronization of the use of an object with the creation (or destruction) of this object;

6. the multiple creation of the same object;
7. the multiple destruction of the same object;
8. the destruction followed by the creation of the same object;

Compositional Semantics

We have introduced with the *closure* operation (see definition 5.4.6) a fashion of calculating the behaviors induced by the synchronizations of elementary behaviors. However we have still to calculate the behaviors induced by the *coordination* events implied by the *coordination formulas* declared at the level of the enveloping *context module*. In order to this we now need the *composition of event names* we have previously described in section 5.4.2. In particular we are interested in the *GateEvRem* relation (noted $x \Leftarrow y/_z$) we have introduced in definition 5.4.4.

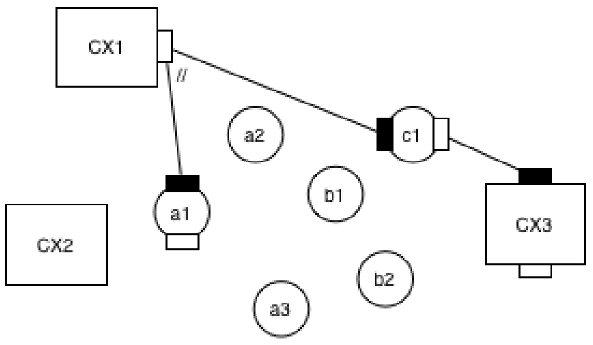


Figure 5.5: Composition Solution

Figure 5.5 depicts the intuition behind the notion of calculating the *composition solution* of a *transition system* given a set of *coordination events*. The figure presents a state of a transition system including the integrated behaviors of *class instances* and *context modules*. For clarity reasons we omit the *events* which are possible from this particular state. The *coordination events* are depicted as the lines connecting *gate ports* to *method ports*. With the *GateEvRem* relation are able to calculate all the possible *composed event names* resulting from the transitivity induced by *coordination events*. From those *composed event names* we are only interested in those for which there are sufficient available resources.

Definition 5.4.7 *Composition Solution*

Let Spec be a specification, $A = Sem(Pres(Spec))$ and $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi \rangle$ be a CO-OPN context *module belonging to Spec. Let also $S^{\mathsf{C}} = Types(Spec)$, $L =$*

$Ports(Md^{\mathsf{X}})$, $I = Ports(C)$, $IO = ClassPorts(Spec)$ and $\langle M, G\rangle = I \cup IO$. Finally let $TS \in \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$ be a transition system (noted $\rightarrow_{\mathsf{c}}$) and $ce \in \mathbf{CE}_{A,L,I,IO,\widehat{A}}$ be a set of coordination events. The composition solution *consists of the function $Comp : \mathbf{CE}_{A,L,I,IO,\widehat{A}} \times \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}} \rightarrow \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$ such that $Comp_{ce}(TS)$ (noted $\rightarrow_{\mathsf{s}}$) is the least fixed point which results from the application of the following rules:*

$$include\ \frac{s \xrightarrow{e}_{\mathsf{c}} s'}{s \xrightarrow{e}_{\mathsf{s}} s'}$$

$$rest\ \frac{\begin{array}{c} l' \triangleq_l l'',\ P(a_1, a_1', a_2, a_2'),\ Q(S, S', S''),\ x \Leftarrow y/_z \in GateEvRem_{A,S^{\mathsf{C}},M,G,ce}, \\ \langle S, \langle l, a_1, m_1\rangle\rangle \xrightarrow{m\ \boldsymbol{with}\ x}_{\mathsf{s}} \langle S', \langle l', a_1', m_1'\rangle\rangle,\ \langle S, \langle l', a_2, m_2\rangle\rangle \xrightarrow{y}_{\mathsf{s}} \langle S'', \langle l'', a_2', m_2'\rangle\rangle \end{array}}{\langle S, \langle l, a_1 \cup a_2, m_1 + m_2\rangle\rangle \xrightarrow{m\ \boldsymbol{with}\ z}_{\mathsf{s}} \langle S' \nabla_S S'', \langle l' \vartriangle_l l'', a_1' \cup a_2', m_1' + m_2'\rangle\rangle}$$

$$noRest\ \frac{\begin{array}{c} l' \triangleq_l l'',\ P(a_1, a_1', a_2, a_2'),\ Q(S, S', S''),\ x \Leftarrow y/_\epsilon \in GateEvRem_{A,S^{\mathsf{C}},M,G,ce}, \\ \langle S, \langle l, a_1, m_1\rangle\rangle \xrightarrow{m\ \boldsymbol{with}\ x}_{\mathsf{s}} \langle S', \langle l', a_1', m_1'\rangle\rangle,\ \langle S, \langle l', a_2, m_2\rangle\rangle \xrightarrow{y}_{\mathsf{s}} \langle S'', \langle l'', a_2', m_2'\rangle\rangle \end{array}}{\langle S, \langle l, a_1 \cup a_2, m_1 + m_2\rangle\rangle \xrightarrow{m}_{\mathsf{s}} \langle S' \nabla_S S'', \langle l' \vartriangle_l l'', a_1' \cup a_2', m_1' + m_2'\rangle\rangle}$$

where

- $m \in (M_{s_1,\dots,s_n})$
- *m_1, m_1', m_2, m_2' in $Mark_{Spec,A}$ and l, l', l'' in $Loid_{Spec,A}$*
- *$a, a', a_1, a_1', a_2, a_2'$ in $Aoid_{Spec,A}$*
- *$S, S', S'' \in \left\{\left(CtxState_{Spec,A,Md_i^{\mathsf{X}}}\right)_i\right\}_{i \in \mathbb{N}}$ where $Md_i^{\mathsf{X}} \in C$*

The similarity between the *rest* and *noRest* rules in definition 5.4.7 and the *sim* rule in definition 5.4.6 is not fortuitous. We want to produce the composed events only in the case where there are sufficient resources so that the synchronization of all the parts of the composition is allowed. This is where the $x \Leftarrow y/_z$ relation comes into play by "testing" if the 'y' event is possible in order to produce the composed event. The difference between the *rest* and the *noRest* rules resides in the fact that with the former we have a remaining *gate call* in the composed *event*, while the later there is no remaining *gate call*.

Context Semantics

At this point we have all the definitions necessary to calculate the semantics of the components of a *context module* (*class instances* and *context modules*) as well as their composition. We are however missing the interaction of the semantics of the *components* with the interface of the enveloping *context module*. This interaction is also defined by *coordination formulas* and implies: the synchronization of the *method ports* of the enveloping *context module* with the *method ports* of the *components*; the synchronization of the *gate ports* of the *components* with the *gate ports* of the enveloping *context module*.

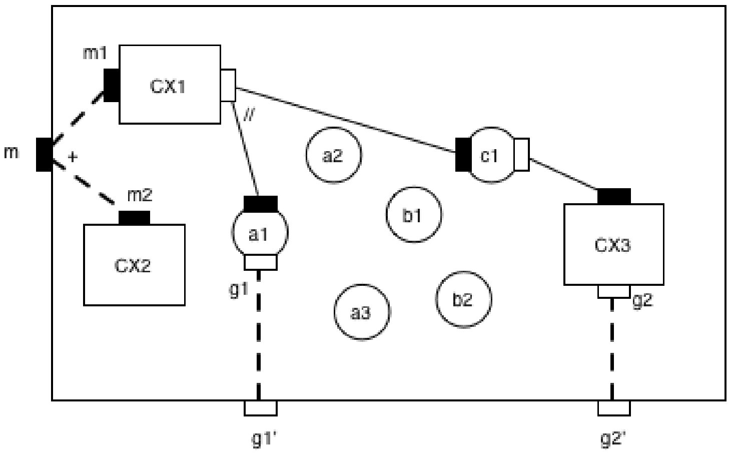


Figure 5.6: Context Filter

We call this additional operation *filtering* and an intuition for its semantics can be observed in figure 5.6. Notice that the mapping of the ports can be more than one-to-one as is exemplified by the '*m* ***with*** *m1*⊕*m2*' synchronization. This example points out that the *filtering* operation restricts the possible *events* for the *components* inside the enveloping *context module*.

Before providing the formal definition of the *filter* function let us introduce the possible *coordination events* available for a *context module*.

Definition 5.4.8 *Coordination events*

Let Spec be a well-formed *CO-OPN specification,* $\Sigma = \langle S, \leq, F \rangle$ *be the global signature of Spec and* $A = Sem(Pres(Spec))$ *be a* Σ*-algebra. Let also* χ *be a set of coordination formulas over a set* X *of S-sorted variables. Finally let* σ *be an assignment of* X *in* A*. The set of coordination expressions* $CoordEv_{A,\chi,X}$ *is built as follows:*

$$\forall \sigma, \forall (cond \Rightarrow coordExp) \in \chi \ . \ A, \sigma \models cond \Rightarrow [\![coordExp]\!]^{\sigma} \in CoordEv_{A,\chi,X}$$

Definition 5.4.9 *Context Filter*

Let Spec be a specification, $A = Sem(Pres(Spec))$ and $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi \rangle$ be a CO-OPN context *module belonging to Spec. Let also $S^{\mathsf{C}} = Types(Spec)$, $L = Ports(Md^{\mathsf{X}})$, $I = Ports(C)$, $IO = ClassPorts(Spec)$, $\langle M, G \rangle = L$ and $\langle M', G' \rangle = I \cup IO$. Finally let $TS \in \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$ be a transition system (noted $\rightarrow_{\mathsf{c}}$) and $CE \in \mathbf{CE}_{A,L,I,IO,\widehat{A}}$ be a set of coordination events. The* context filter *consists of the function $Filter : \mathbf{CE}_{A,L,I,IO,\widehat{A}} \times \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}} \rightarrow \mathbf{TS}_{Spec,A,Md^{\mathsf{X}}}$ such that $Filter_{cf}(TS)$ (noted $\rightarrow_{\mathsf{f}}$) is the least fixed point which results from the application of the following rules:*

$$methodSync \quad \frac{m \ \textbf{\textit{with}}\ mexp \in CE, \ s \xrightarrow{mexp}_{\mathsf{c}} s'}{s \xrightarrow{m}_{\mathsf{f}} s'}$$

$$methodGateSync \quad \frac{\begin{array}{l} m \ \textbf{\textit{with}}\ mexp \in CE, \\ gexp' = gateF_{CE}(gexp), \ s \xrightarrow{mexp \ \textbf{\textit{with}}\ gexp}_{\mathsf{c}} s' \end{array}}{s \xrightarrow{m \ \textbf{\textit{with}}\ gexp'}_{\mathsf{f}} s'}$$

$$compOp \quad \frac{\begin{array}{l} \{m \ \textbf{\textit{with}}\ mexp, \ m' \ \textbf{\textit{with}}\ mexp'\} \subseteq CE, \\ gexp'' = gateF_{CE}(gexp), \ gexp''' = gateF_{CE}(gexp'), \\ s \xrightarrow{mexp \ \textbf{\textit{with}}\ gexp \ op \ mexp' \ \textbf{\textit{with}}\ gexp'}_{\mathsf{c}} s' \end{array}}{s \xrightarrow{m \ \textbf{\textit{with}}\ gexp'' \ op \ m' \ \textbf{\textit{with}}\ gexp'''}_{\mathsf{f}} s'}$$

$$compEmptyOp1 \quad \frac{\begin{array}{l} \{m \ \textbf{\textit{with}}\ mexp, \ m' \ \textbf{\textit{with}}\ mexp'\} \subseteq CE, \\ gexp'' = gateF_{CE}(gexp), \ s \xrightarrow{mexp \ \textbf{\textit{with}}\ gexp \ op \ mexp'}_{\mathsf{c}} s' \end{array}}{s \xrightarrow{m \ \textbf{\textit{with}}\ gexp'' \ op \ m'}_{\mathsf{f}} s'}$$

$$compEmptyOp2 \quad \frac{\begin{array}{l} \{m \ \textbf{\textit{with}}\ mexp, \ m' \ \textbf{\textit{with}}\ mexp'\} \subseteq CE, \\ gexp'' = gateF_{CE}(gexp), \ s \xrightarrow{mexp \ op \ mexp' \ \textbf{\textit{with}}\ gexp}_{\mathsf{c}} s' \end{array}}{s \xrightarrow{m \ op \ m' \ \textbf{\textit{with}}\ gexp''}_{\mathsf{f}} s'}$$

where

- $m, m' \in (M_{s_1,\ldots,s_n})$
- $mexp, mexp' \in MethodExpr_{A,S^{\mathsf{C}},M'}$ *and* $gexp, gexp' \in GateExpr_{A,S^{\mathsf{C}},G'}$
- $gexp'', gexp''' \in GateExpr_{A,S^{\mathsf{C}},G}$

Let us finally introduce the *semantics* for a given *context module.* The semantics is calculated as a composition of the *Closure* (see definition 5.4.6), *Comp* (see definition 5.4.7) and $Filter$ functions (see definition 5.4.9) applied to the *unified transition system* obtained from the *class modules* and *context modules* coordinated by that *context module*.

Definition 5.4.10 *Context Semantics*

Let Spec be a specification, $A = Sem(Pres(Spec))$ and $Md^{\mathsf{X}} = \langle \Xi^X, O, C, X, \chi\rangle$ be a CO-OPN context *module belonging to Spec. The semantics of Md^{X} — noted $Sem_{Spec,A}(Md^{\mathsf{X}}) \in \mathbf{\mathsf{TS}}_{Spec,A,Md^{\mathsf{X}}}$ — is calculated as follows:*

$$Sem_{Spec,A}(Md^{\mathsf{X}}) = Filter_{ce}(Closure(Comp_{ce}(Closure(UPS_{Spec,A}(Md^{\mathsf{X}})))))$$

where $ce = CoordEv_{A,\chi,X}$.

Note that the definitions of *context semantics* and *unified partial semantics* (see definition 5.4.5) are mutually recursive, thus allowing the calculation of the semantics of the coordinated *context modules*. As we have previously stated, the *base case* for the recursion consists of *context modules* that are either empty or only contain *class instances*. Notice also that we apply the $Closure$ function twice in the equation in definition 5.4.10, once *before* and once *after* the application of the $Comp$. The invocation *before* is due to the fact that the composition can only be calculated if all the synchronized elementary behaviors already present in the transition system — due to the fact that coordination events may involve synchronizing with composed behaviors. The invocation *after* is due to the fact that the $Comp$ function introduces a new set of events in the transition system which may also be synchronized by the typical operators.

5.5 CO-OPN Specification Semantics

We can now define the semantics of a CO-OPN/$_{2c++}$ specification. Intuitively the semantics of such a specification consists of the semantics of the *topmost context module* in the *containment* hierarchy. However, it is possible to define several *topmost context modules* in the same specification. We thus define the semantics of a CO-OPN/$_{2c++}$ specification as the union of all the semantics of all the *topmost context modules* in that specification.

Definition 5.5.1 *Semantics of a CO-OPN/$_{2c++}$ Specification*

Let Spec be a specification and $A = Sem(Pres(Spec))$. The semantics of Spec is noted $Sem_A(Spec)$ and is defined as follows:

$Sem_A(Spec) = \bigcup_i Sem_{Spec,A}(Md_i^{\mathsf{X}})$, such that Md_i^{X} is the first element of the partial order $\sqsubset_i \subseteq D_{Spec}^{\mathsf{X}}$ $(1 \leq i \leq n)$.

In definition 5.5.1 we use the notion of *partial order* in order to find the *topmost context modules*. The possible *partial orders* are calculated on the D_{Spec}^{X} relation we have introduced in section 4.4.7. The *topmost context modules* correspond to the first elements of the existing *partial orders* in D_{Spec}^{X}.

5.6 Summary

In this chapter we have progressively defined the semantics of a *well-formed* CO-OPN/$_{2c++}$ specification. We have started by the definition of the *algebraic models* of a CO-OPN specification which consist of both the models of the ADT modules of the specification and a particular kind of models called the *object identifier algebra*. The object identifier algebra has two purposes: on the one hand it allows handling object identifiers in a dynamic fashion; on the other hand it includes mechanisms for casting an object to either a *subtype* or a *supertype*. We have then introduced a number of sets of functions for manipulating *object identifiers*. These sets of functions are later used in order to model the state of a CO-OPN specification. They include the calculations of the *last object identifier* for a given type, the set of *alive objects* for a given type and the new set of *alive object* after an object has been *created* or *destroyed*. All the definitions of the *algebraic models* of a CO-OPN specification and of the functions for manipulating *object identifiers* were directly taken without change from [58].

We have then proceeded to the definition of the *state* of a CO-OPN context. This definition of state is inspired from the previous definition in [58] which concerned a dynamic network of objects. We have introduced a new notion of state which is recursively defined in order to include the states of the coordinated *contexts*. In other words, the state of each *context* includes the state of a dynamic network of objects and the states of the *contexts* it coordinates. We have also defined the notion of *context transition system* which corresponds to a typical transition system where the states are CO-OPN context *states* and the transitions are events of the *context module*.

In order to define the semantics of a CO-OPN specification we have started by defining the semantics of a *context module*. Our definition is inspired from the work of Huerzeler [6] which studies the semantics of a set of component evolving concurrently and collaborating by being synchronized. The framework introduced by Huerzeler was too abstract to be directly applied to CO-OPN specifications as it considered very simplified components. A large part of our work was then to adapt the existing approach to the hierarchical definition of CO-OPN components, which can be both *objects* or *contexts*.

The semantics of a CO-OPN *context module Ctx* is calculated in a sequence of steps:

- We first calculate the semantics of a set of CO-OPN classes involved in the definition of *Ctx* by producing the transition system reflecting the elementary behaviors the classe's instances. The set of inference rules allowing this calculation is the same as the one presented in [58];
- We then perform a kind of "*cartesian product*" between the transition system calculated in the previous step and the transition systems representing the semantics of each of the *context modules* coordinated by *Ctx*. The operation produces a new transition system called the *unified partial semantics*, the *states* of which encompass the states of all the components of *Ctx* and the *behaviors* of. which include the behaviors of all the components of *Ctx*;
- A *closure* function is then applied to the *unified partial semantics* of *Ctx* producing a new transition system including new behaviors which are possible synchronizations of the behaviors of *Ctx*'s components. The definition of the closure operation is inspired from [58] but was extended to take into consideration the state of *context modules*. A number of particular operators were developed for that purpose;
- The transition system calculated in the previous step is then used in conjunction with the *coordination formulas* of *Ctx*. The purpose is to produce a new transition system including the behaviors resulting from coordinating the components of *Ctx*. The method is inspired from [6] but we were obliged to completely review the definitions in order to adapt them to CO-OPN;
- A new *closure* operation is applied to the transition system defined in the previous step. The purpose is to recalculate the behaviors resulting from synchronizing the coordinated behaviors calculated in the previous step;
- The behaviors present in the transition system calculated in the previous step are filtered taking into consideration the interface of *Ctx*.

Finally we have introduced the semantics of a *well-formed* CO-OPN$_{/2c++}$ specification *Spec* as the union of the semantics of all the *topmost* context modules of the specification. We define a *topmost* context modules of the specification as all the first elements of the partial orders possible on the "*contains*" binary relation between the context modules of *Spec*.

Chapter 6

Testing and CO-OPN State of the Art

In this chapter we will present the testing background our approach is based on. Previously to our work, Cécile Péraire and Stéphane Barbey have written their Ph.D. theses [4, 5] on model-based test case generation from CO-OPN$_{/2}$ specifications. Their approach was based on the BGM testing theory by Bernot, Gaudel and Marre [3]. While BGM uses as specification language *algebraic specifications*, CO-OPN$_{/2}$ is a formalism mixing algebraic specifications, *Petri Nets* and concepts of *Object Orientation*. This said, Péraire and Barbey had to adapt BGM in order to keep the same theoretical framework.

We will start by presenting the original BGM theory and the way in which it frames test selection. After this we will introduce the adaptation of BGM to CO-OPN$_{/2}$, which includes some important changes in the assumptions originally made in BGM, in particular in terms of the test formalism. We will finalize the chapter by pointing out some deficiencies on the work done by Péraire and Barbey and proposing improving the state-of-the-art notably at the level of the usability of the approach and the associated methodology.

6.1 BGM Theory

The BGM theory is a kind of *Model-Based Testing* which aims at producing tests for SUT's based on models which are *algebraic specifications.* In the following presentation we will not concentrate on the details of the theory concerning algebraic specifications (which can be found in [30]), but will rather provide a generalization of the framework which is applicable to any sufficiently formal specification language. With the term *sufficiently formal* we mean that a model in that specification language should define the SUT's expected behavior in a fashion that tests for that SUT may be computed in an automatic (or semi-automatic) fashion. This following presentation of the BGM theory is inspired by the work of Gaudel [41] and Péraire [4].

The traditional view on testing which was initially stated by Myers in the book "The Art of Software Testing" [21] is the following:

"Testing is the process of executing a program with the intent of finding errors."

However, given that a program (or SUT) has typically an infinite amount of behaviors, we can also state that:

"Proving that a system does what it is intended to do by testing it is impossible."

In fact, *proving* that an SUT does what it is intended to do is an activity that pertains to the domain of techniques such as *model checking* or *theorem proving* which we have briefly mentioned in chapter 1. The BGM theory, however, sees the testing activity as *proving* that the SUT behaves as the specification predicts. This proof is accomplished by executing the SUT with a *practicable* test set derived from the specification. By *practicable* we understand that the test set should be executable in a finite "reasonable" amount of time. This may sound contradictory as the necessary test set to perform such a proof is in general infinite, given the infinite amount of behaviors present in the specification. Despite, we may claim that the proof holds if the used test set has the following properties:

- *validity*: if a test set accepts an SUT, then that SUT is correct; in other words, no incorrect SUTs are accepted by the test set.
- *unbiasedness*: if an SUT is correct, then the test set accepts it; in other words, no correct SUTs are rejected by the test set.

The success of the approach relies then on selecting from an SUT's specification a *valid* and *unbiased* test set — also called a *pertinent* test set — which is also *practicable*. BGM proposes performing this selection by stating well understood *hypothesis* about the behavior of the SUT — typically "*1 to n*" or "*m to n*" generalizations of certain quantifiable aspects of the behavior of the SUT. Using these hypothesis, it is possible to reduce the initial *exhaustive* and possibly infinite test set obtained from the specification. By devising a number of such generalizations about the SUT the test engineer may reach a *pertinent* (*valid* and *unbiased*) and *practicable* test set — but only if the generalizations contained in those reduction hypothesis about the SUT are true, which may not always be possible to prove.

One of the main advantages of the BGM theory is that it relies on the quality of the *reduction hypothesis* made by the test engineer in order to devise a test set that "proves" that an SUT is correct according to its specification, or in the failing case, discovers implementation errors. The approach mimics the traditional test set construction where, in order to build a "reasonably sized" test set, a test engineer uses his/her experience to generalize a number of selected behaviors to the full behavior of the SUT.

Notice that it would also be possible to state *hypothesis* about the behaviors contained in the specification rather than about those contained in the SUT. However, given that the specification is normally described at a more abstract level than the SUT and/or that the specification may be incomplete regarding the SUT, it is more semantically meaningful for a test engineer to state the hypothesis about the object that is being tested.

Formalization of the Approach

Let us now introduce a formalization for the BGM theory. The formalization is useful not only to state the theory in a clear fashion, but also in order to provide a precise notation for the subsequent development of this thesis.

In the text that follows we will use the following notations: *Spec*, the class of all specifications written in the considered specification language; $Prog$, the class of all SUTs written in the chosen implementation language; $Test$, the class of all test sets that can be produced using the chosen test language; *Hyp*, the class of all reduction hypothesis we can write in the chosen hypothesis language.

Definition 6.1.1 *Pertinence (Validity and Unbiasedness)*

Let $P \in Prog$ be system under test (SUT), $SP \in Spec$ be a specification for that SUT and $T_{SP} \in Test$ be a set of tests derived from SP. Let also $\models \subseteq (Prog \times Spec)$ be satisfaction relation between SUTs and specifications and $\models_O \subseteq (Prog \times Test)$ be an oracle satisfaction relation between SUTs and test sets. If T_{SP} is a pertinent *test set, then the following equivalence holds:*

$$P \models SP \Leftrightarrow P \models_O T_{SP}$$

Definition 6.1.1 concentrates both the concepts of *validity* and *unbiasedness.* As we have seen before, *validity* stands for the fact that "*if a test set accepts an SUT, then that SUT is correct*" or, in error detection terms, any *error* in an SUT will be detected by a *valid* test set. *Validity* corresponds to the *right to left* direction of the equivalence in definition 6.1.1:

$$P \models_O T_{SP} \Rightarrow P \models SP$$

On the other hand, *unbiasedness* stands for the fact that "*if an SUT is correct, then the test set accepts it*" or, in error detection terms, no errors will be detected in correct SUT by an *unbiased* test set. *Unbiasedness* corresponds to the *left to right* direction of the equivalence in definition 6.1.1:

$$P \models SP \Rightarrow P \models_O T_{SP}$$

As was noted by Weyuker and Ostrand in [59], an obvious test set that guarantees both *validity* and *unbiasedness* is the one that covers all the behaviors expressed in the specification. Given a specification SP, we will note $Exhaust_{SP}$ the exhaustive test set that contains all the behaviors modeled by SP. Checking the satisfaction of the $Exhaust_{SP}$ test set by an SUT implementing SP corresponds in fact to a proof of correctness of the SUT with respect to SP. However, as we have previously mentioned, a test set covering all of the specification's behavior is usually infinite, making it not *practicable* in the general case.

Hypothesis for Test Selection

In order to tackle the fact that the exhaustive test set is typically infinite, the BGM theory proposes stating hypothesis that generalize the correct implementation of certain behaviors of the SUT to the correct implementation of the whole SUT. By taking those generalizations into consideration we are able to reduce the size of the exhaustive test set needed to cover all of the specification's behavior.

Let us consider the example of a software controller for a simplified Drink Vending Machine (DVM) depicted in figure 6.1. Assume the machine sells five kinds of drinks: *d1* and *d2* costing 1CHF, *d3* costing 2CHF and *d4* and *d5* costing 3CHF. A user wanting to buy a drink would insert a number of CHF coins in the DVM and would select the desired drink. If the inserted money is enough, the DVM distributes the drink and possibly some change. Otherwise the DVM alerts the user for the fact that the inserted money is not enough.

Even in such a small system as the simplified DVM controller one may find a very large, if not infinite, amount of behaviors. For example, one could be able to insert a very large number of coins in the DVM before selecting the drink. While testing the DVM, we could envisage a test engineer stating the following hypothesis

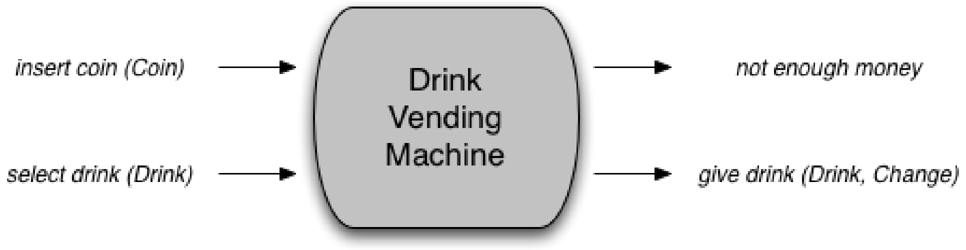


Figure 6.1: Simplified Drink Vending Machine

about its correctness:

"If the DVM works well while buying one drink, then the DVM works well for buying any available drink."

This hypothesis corresponds to a "*1 to n*" generalization of the behavior of the DVM, where we generalize the correction of the implementation of the choice of one drink to the correction of the implementation of the choice of all drinks. In the BGM theory a "*1 to n*" generalization is called a *uniformity hypothesis* and can be formalized in the following fashion:

Definition 6.1.2 *Uniformity Hypothesis*

Let $f \in Test$ be a test and v be a variable of f of a type T (of the chosen test language), ranging over a domain D. A uniformity hypothesis *on the domain D for an SUT $P \in Prog$ is the following assumption:*

$$\forall t_0 \in D \; . \; (P \models f(t_0) \Rightarrow (\forall t_1 \in D \; . \; P \models f(t_1))$$

In definition 6.1.2 we mention the existence of a variable $v \in D$ in test f. By this slight abuse of notation we mean to indicate that f corresponds to a test set resulting from instantiating v to all its possible values in D. Note that if we consider the above hypothesis example about the DVM, variable t_0 corresponds to the drink chosen to represent the whole set of available drinks.

It is relevant to mention that in the work of Gaudel [41] and Péraire [4] the types in the testing language are algebraic types. For the sake of generality and simplicity of the presentation, in definition 6.1.2 we have relaxed that assumption. We simply consider the test language includes a number of types to describe the possible dimensions of a test case.

Another possible hypothesis about the correctness of the DVM would be:

"If the DVM works well for sequences of coin insertions up until 10 coins, then the DVM works well for any number of coin insertions."

In this case the hypothesis corresponds to an "*m to n*" generalization of the correctness of the behavior of the DVM. This kind of hypothesis in the BGM theory is called a *regularity hypothesis* and can be formalized as follows:

Definition 6.1.3 *Regularity Hypothesis*

Let $f \in Test$ be a test and v be a variable of f of a type T of the chosen test language, ranging over a domain D. Let also α be an interest function taking as argument a value of a type T and returning a natural number. A regularity hypothesis *for a program $P \in Prog$ is the following assumption:*

$$((\forall t_0 \in D \ . \ (\alpha(t_0) \leq k \Rightarrow P \vDash f(t_0)) \Rightarrow (\forall t_1 \in D \ . \ P \vDash f(t_1))$$

Coming back to the hypothesis about the insertion of coins, t_0 corresponds to all the insertions of coins up until 10 — the natural k number which results in applying the complexity function α to t_0. Normally complexity functions return some sort of syntactically measurable quantity present in the values of the testing language types.

As we have previously seen, in the BGM theory we are only interested in *valid* and *unbiased* test sets. However, if we are using a series of hypothesis about the SUT in order build up the *pertinent* test set, we need to review the definition of *pertinence* we have provided in 6.1.1.

Definition 6.1.4 *Pertinence assuming Hypothesis about the SUT*

Let $T_{SP} \in Test$ be a test set obtained by assuming a set of hypothesis $H \subseteq Hyp$ about a SUT $P \in Prog$ with specification $SP \in Spec$. Let also $\vDash \subseteq (Prog \times Spec)$ be a satisfaction relation between SUTs and specifications and $\vDash_O \subseteq (Prog \times Test)$ be an oracle satisfaction relation between SUTs and test sets. T_{SP} is a pertinent *test set if the following implication holds:*

$$H \Rightarrow (P \vDash SP \Leftrightarrow P \vDash_O T_{SP})$$

We can then imagine the test selection activity as: 1) the act of devising a number of *uniformity* and/or *regularity* hypothesis about the behavior of the SUT; 2) given a specification SP of the SUT, applying the selected hypothesis to the exhaustive test set $Exhaust_{SP}$ in order to reduce it to a *practicable* size while keeping *pertinence*.

In this line of thought, it seems natural to think about any pertinent test set for a given SUT as attached to a set of hypothesis about that same SUT. More formally, let us introduce the notion of *testing context*.

Definition 6.1.5 *Testing Context*

Let $H \subseteq Hyp$ be a set of hypothesis about an SUT $P \in Prog$, $SP \in Spec$ a specification of P and T_{SP} a set of test cases extracted from SP. A testing context for P and SP is a pair $\langle H, T_{SP} \rangle$ where T_{SP} is pertinent if P satisfies hypothesis H.

Given the notion of *testing context* presented in 6.1.5, it becomes useful to define a *preorder* relation between testing contexts.

Definition 6.1.6 *Context preorder relation*

Let $TC = \langle H, T \rangle$ and $TC' = \langle H', T' \rangle$ be two testing contexts for a given SUT $P \in Prog$. We say that TC' refines *TC (noted $TC \lesssim TC'$) if and only if the following conditions hold:*

- *The hypothesis H' are stronger than the hypothesis H:*

$$H' \Rightarrow H$$

- *If the hypothesis H' about P hold, then T' detects as many errors as T:*

$$H' \Rightarrow (P \models_O T' \Rightarrow P \models_O T)$$

- *The test set T' is contained in the test set T:*

$$T' \subseteq T$$

We can now state the following theorem:

Theorem 6.1.7 *Preservation of Pertinence*

Assume a specification $SP \in Spec$ of an SUT $P \in Prog$ and two test contexts $\langle H, T_{SP} \rangle$ and $\langle H', T'_{SP} \rangle$ such that $\langle H, T_{SP} \rangle \lesssim \langle H', T'_{SP} \rangle$. If $\langle H, T_{SP} \rangle$ is pertinent then $\langle H', T'_{SP} \rangle$ is pertinent.

Proof*: trivial by construction.*

We can now use the idea of *refinement* presented in 6.1.6 in order to define a methodology for selecting *pertinent* testing contexts for a specification SP of an SUT P. This can be done starting with the testing context $\langle H_{min}, Exhaust_{SP}\rangle$ — where H_{min} corresponds to the empty set of hypothesis — and producing a sequence of refinements by applying increasingly stronger hypothesis about P. The goal is to eventually reach a *pertinent* test set with a reasonable size. This process can be observed in figure 6.2, where the test selection by refinement goes as follows:

$$\langle H_{min}, Exhaust_{SP}\rangle \lesssim \ldots \lesssim \langle H^i, T^i_{SP}\rangle \lesssim \langle H^j, T^j_{SP}\rangle \lesssim \ldots \lesssim \langle H, T_{SP}\rangle$$

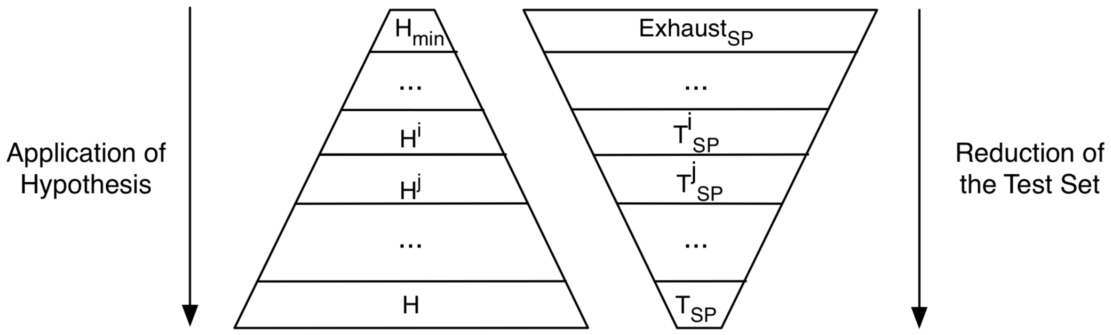


Figure 6.2: Test Selection by Context Refinement

Notice that if we refine the testing context $\langle H_{min}, Exhaust_{SP}\rangle$, the *unbiasedness* property is guaranteed by the fact that each refinement is a subset of the *exhaustive* test set. Given that the exhaustive test set is *unbiased* by construction (since it is derived from the specification), a subset of the exhaustive test set cannot discover any more errors than the ones described in the specification and is thus unbiased too. In fact, the only fashion in which it would be possible to refine an *unbiased* testing context into a biased one would be to add tests covering behaviors not present in the specification — which is by definition impossible.

On the other hand, *validity* is only guaranteed if the hypothesis about the SUT which are introduced at each step of the refinement hold. Since the hypothesis are generalizations of the correctness of the behavior of the SUT, the danger is that the test engineer states hypothesis which are too strong. If the hypothesis do not hold it may happen that *validity* is not kept because the obtained test set does not cover all relevant behaviors as stated in the specification. It then becomes possible for an SUT to satisfy the test set even if it does not correctly implement the specification.

Coming back to the DVM example, in order to produce a *pertinent* and *practicable* test set we would formalize the uniformity and regularity hypothesis we have stated above — using an appropriate hypothesis language — and would use them to refine the exhaustive test set derived from the DVM specification. The order of the application of the hypothesis is not relevant, as long as the SUT satisfies them.

Up until now we have only discussed hypothesis stated by the test engineer about the SUT. However, given that the specification is expressed in a formal fashion, it is possible to automatically explore that formal definition to complement the human hypothesis. Imagine we have a specification of the DVM where both the behaviors of *inserting money* and of *selecting a drink* are modeled. Clearly, in order to model the selection of the drink it is necessary to model three distinct behaviors:

- the buyer hasn't inserted enough coins and cannot buy the selected drink;
- the buyer has inserted just enough coins for the selected drink, gets the drink but no change;
- the buyer has inserted too many coins for the selected drink, gets the drink and some change.

Given that these behaviors have to be specified by some formal conditions, we could envisage exploring the semantics of those conditions in order to find uniformity domains for the operation selecting the drink. In particular for the *select drink* operation it would be interesting to choose only one drink per behavior stated in the items above. Imagine the user has inserted 2CHF in the machine. Then it would be possible to test the three possible behaviors of drink selection by choosing one drink per behavior. The possible sets of drinks to select for testing the three domains would be $\{d1, d3, d5\}$, $\{d2, d3, d4\}$, $\{d1, d3, d4\}$ or $\{d2, d3, d5\}$. Notice that we are using the specification to discover domains where the behavior of the SUT in similar. We can thus apply uniformity hypothesis to those domains in order to choose one drink per behavior. It would then be possible to choose only one of the sets of drinks above in order to test the *select drink* behavior after having inserted the 2CHF.

The hypothesis discovered by using the specification are a special type of *uniformity hypothesis* and are named in BGM *uniformity hypothesis with subdomain decomposition*. They correspond to a kind of *white box* testing of the specification since symbolic techniques (as we have seen in chapter 2) are used in order to expose the subdomains.

6.2 Adaptation of the BGM theory to CO-OPN$_{/2}$

In order to apply the BGM theory to CO-OPN$_{/2}$ in a practical fashion we have to tackle several problems. Firstly, we have to choose a test language for CO-OPN$_{/2}$. Then, the exhaustive test set for a CO-OPN$_{/2}$ specification has to be built. Finally, there needs to be some sort of mechanism with which we can express hypothesis about the correctness of the SUT in order to reduce the exhaustive test set for a CO-OPN$_{/2}$ specification to a *pertinent* and *practicable* one.

In this section we will start by introducing the test formalism adopted for CO-OPN$_{/2}$ specifications. The original BGM theory [3] was devised having in mind *algebraic specifications*. In this setting, both the specifications and the test cases are expressed using algebraic equations where the left and the right side of the equation are terms of the same sort. For CO-OPN$_{/2}$ specifications Pérarire and Barbey have chosen to use as test formalism the HML (Hennessy-Milner Logic) [35] temporal logic. Given the fact that CO-OPN$_{/2}$ specifications represent state-based SUTs with possible hidden state changes, we need a test formalism suitable to provide an observational equivalence relation between specifications and SUTs. As we will show HML is a suitable formalism to provide this equivalence relation. It also proposes an interesting syntax for the expression of test cases.

6.2.1 CO-OPN$_{/2}$ equivalence

As we have seen in chapter 5, a CO-OPN$_{/2}$ specification has a transition system semantics. Also, as explained in [1], the CO-OPN$_{/2}$ equivalence corresponds to the *bisimulation* relation between the transition systems denoting the semantics of CO-OPN$_{/2}$ models. Bisimulation [60] is interesting in the context of testing because it disregards state comparison and only concentrates on knowing if two systems always allow the same events after having executed the same prefix starting from an initial state. Given that the state of an SUT is not directly observable (remember we are doing black-box testing), bisimulation offers an observational equivalence relation to allow the comparison. On the other hand, the internals of an SUT are irrelevant as long as it behaves as the specification predicts.

Definition 6.2.1 *Strong bisimulation*

A strong bisimulation *between two transition systems TS_1, TS_2 is the relation $R \subseteq State(TS_1) \times State(TS_2)$ such that:*

- *if $st_1 \, R \, st_2$ and $st_1 \xrightarrow{e} st'_1 \in TS_1$ then there is $st_2 \xrightarrow{e} st'_2 \in TS_2$ such that $st'_1 \, R \, st'_2$;*

- *if* $st_1 \, R \, st_2$ *and* $st_2 \xrightarrow{e} st'_2 \in TS_2$ *then there is* $st_1 \xrightarrow{e} st'_1 \in TS_1$ *such that* $st'_1 \, R \, st'_2$*;*
- $st_1^{init} \, R \, st_2^{init}$

We say that TS_1 *and* TS_2 *are* strongly bisimilar*, if there exists such a non-empty relation* R*, and we denote this by* $TS_1 \underline{\Leftrightarrow} TS_2$*.*

This said, we can now formally define for CO-OPN$_{/2}$ a notion of satisfaction between a specification and an SUT we have generically introduced in section 6.1. In the text that follows we will use the function $Event(SP)$ which returns all the events of a CO-OPN$_{/2}$ specification SP.

Definition 6.2.2 *Satisfaction relation between SUTs and CO-OPN$_{/2}$ specifications*

Let $P \in Prog$ *be an SUT and* $SP \in Spec$ *be a CO-OPN$_{/2}$ specification. Let also* $G(P) = \langle Q_1, Event(P), \rightarrow_1, i_1 \rangle$ *be a transition system denoting the semantics of* P *and* $G(SP) = \langle Q_2, Event(SP), \rightarrow_2, i_2 \rangle$ *be a transition system denoting the semantics of* SP*. Assuming there is a one-to-one morphism between the signatures of* P *and* SP*, the satisfaction relationship* $\models \subseteq Prog \times Spec$ *is defined as follows:*

$$P \models SP \Leftrightarrow G(P) \underline{\Leftrightarrow} G(SP)$$

In the context of BGM, the first step towards building a test set is to build an exhaustive test set from a CO-OPN$_{/2}$ specification. Definition 6.2.2 is interesting because it provides us with a formal basis to extract that exhaustive test set. In fact, the exhaustive test set would cover all the scenarios interesting to establish the strong bisimulation equivalence between a specification and an SUT. As we will show in the next section, HML is an appropriate formalism for expressing those test sets.

Finally, notice that in definition 6.2.2 one of the conditions is that there exists a one-to-one morphism between the signatures of P and SP. This is a necessary condition to establish strong bisimulation as it allows comparing events occurring in the specification and events occurring in the SUT. In the remainder of this thesis we will assume this hypothesis is true for the systems we are testing.

6.2.2 Test Language - The HML Formalism

HML is a simple temporal logic built to express properties of processes. Let us start by establishing the syntax of possible HML formulas for a given CO-OPN$_{/2}$ specification.

Definition 6.2.3 *Syntax of HML_{SP}*

Let $SP \in Spec$ be a CO-OPN$_{/2}$ specification, $G(SP) = \langle Q, Event(SP), \rightarrow, i\rangle$ a transition system denoting the semantics of SP and $Event(SP)$ the set of all events of SP. The syntax of the HML formulas for SP — noted HML_{SP} — is built as follows:

- $T \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\neg f) \in HML_{SP}$
- $f, g \in HML_{SP} \Rightarrow (f \wedge g) \in HML_{SP}$
- $f \in HML_{SP} \Rightarrow (\langle e\rangle f) \in HML_{SP}$ *where* $e \in Event(SP)$

Notice that the syntax of HML_{SP} includes any possible formula involving the events of the specification SP. The semantics of HML_{SP} is defined in terms of the satisfaction of the formulas of HML_{SP} by the transition system denoting the semantics of specification SP.

Definition 6.2.4 *Semantics of HML_{SP}*

Let $SP \in Spec$ be a CO-OPN$_{/2}$ specification and $G(SP) = \langle Q, Event(SP), \rightarrow, i\rangle \in \Gamma$ a transition system denoting the semantics of SP and a state $q \in Q$, the satisfaction relation $\models_{HML_{SP}} \subseteq \Gamma \times Q \times HML_{SP}$ is defined as:

- $G, q \models_{HML_{SP}} T$
- $G, q \models_{HML_{SP}} (\neg f) \Leftrightarrow G, q \not\models_{HML_{SP}} f$
- $G, q \models_{HML_{SP}} (f \wedge g) \Leftrightarrow G, q \models_{HML_{SP}} f$ *and* $G, q \models_{HML_{SP}} g$
- $G, q \models_{HML_{SP}} (e\langle f\rangle) \Leftrightarrow \exists e \in Event_{SP}$ *such that* $q \xrightarrow{e} q' \in \rightarrow$ *and* $G, q' \models_{HML_{SP}} f$

In definition 6.2.3 we have introduced the syntax of HML formulas for a CO-OPN$_{/2}$ specification SP having a given set of events. In definition 8.3.3 we gave those HML formulas a semantics by establishing the notion of satisfaction of an HML formula by a transition system representing the semantics of SP. Notice that by introducing this notion of satisfiability we are able to distinguish HML formulas which correspond to acceptable behaviors of SP from HML formulas which correspond to unacceptable behaviors (those formulas that are *not* satisfied by the transition system). We can now define the notion of a *test case* as an HML formula with an associated logic value stating whether or not the formula corresponds to an acceptable behavior. We can also extrapolate the notion of *test case* to the notion of *test set* for a CO-OPN$_{/2}$ specification.

Definition 6.2.5 *Test Set for a CO-OPN$_{/2}$ Specification*

Let $SP \in Spec$ be a CO-OPN$_{/2}$ specification and $G(SP) = \langle Q, Event(SP), \rightarrow, i\rangle$ a transition system representing the semantics of SP. Given a set of formulas $F \subseteq HML_{SP}$, the test set *generated from F is defined as follows:*

$$\begin{aligned}Test_{SP}(F) = \{\langle f, Result\rangle \in F \times \{true, false\} \mid \\ (G(SP), i \models_{HML_{SP}} f \text{ and } Result = true) \text{ or} \\ (G(SP), i \not\models_{HML_{SP}} f \text{ and } Result = false)\}\end{aligned}$$

Typically in model based testing only tests covering valid behaviors are chosen. Tests covering invalid behaviors are also relevant as they can uncover errors of comission, rather than errors of omission. These kinds of tests can be seen as *robustness* testing.

6.2.3 Exhaustive Test Set for a CO-OPN$_{/2}$ Specification

Having formally stated the notion of *test set* for a CO-OPN$_{/2}$ specification in definition 6.2.5, we can now more precisely introduce the *oracle satisfaction* we have informally stated in 6.1. Given that both the semantics of the specification and of the SUT are represented by transition systems, the *oracle satisfaction* happens when the HML formulas that compose the test set have the same semantics in both transition systems.

Definition 6.2.6 *Oracle satisfaction*

Let $P \in Prog$ be a program and $SP \in Spec$ be a CO-OPN$_{/2}$ specification. Let also $G(P) = \langle Q, Event(P), \rightarrow, i\rangle$ be a transition system representing the semantics of P. Assuming there is a one-to-one morphism between the signatures of P and SP, the satisfaction relation $\models_O \subseteq Prog \times Test$ is defined as follows for a given test set T_{SP} for specification SP:

$$\begin{aligned}(P \models_O T_{SP}) \Leftrightarrow (\forall \langle Formula, Result\rangle \in T_{SP} \\ (G(P), i \models_{HML_{SP}} Formula \text{ and } Result = true) \text{ or} \\ (G(P), i \not\models_{HML_{SP}} Formula \text{ and } Result = false)\}\end{aligned}$$

A pertinent test set (see definition 6.1.1) to establish bisimulation observational equivalence for a specification SP corresponds to all HML_{SP} formulas and their corresponding semantics in the transition system denoting the semantics of SP.

Definition 6.2.7 *Exhaustive Test Set for a CO-OPN$_{/2}$ Specification*

Let $SP \in Spec$ be a CO-OPN$_{/2}$ specification. The exhaustive test set $Exhaust_{SP}$ is defined as follows:

$$Exhaust_{SP} = Test_{SP}(HML_{SP})$$

Note that in definition 6.2.7 of the *exhaustive test set* we include not only possible behaviors of the specification, but also behaviors that should *not* happen. Negative behaviors are introduced by the semantics of the '$\neg$' HML operator. In fact, both positive and negative formulas are necessary to establish the *bisimulation equivalence* between two transition systems by using the HML equivalence. The equivalence (also called full agreement) between *bisimulation equivalence* and *HML equivalence* was proved by Hennessy and Milner in [35], but for image finite transition systems. This means that for a given state of the transition system, only a finite number of transitions is possible. We will keep this — not too restrictive assumption — throughout this thesis.

From the full agreement between the *bisimulation equivalence* and *HML equivalence* we can extract the following result applied to model based testing from CO-OPN$_{/2}$ specifications. This result rejoins the initial definition 6.1.1 of *pertinent test set* and proves that we can use HML to build the *exhaustive test set* for a CO-OPN$_{/2}$ specification.

Theorem 6.2.8 *Full agreement between CO-OPN$_{/2}$ equivalence and Oracle satisfaction*

Let $P \in Prog$ be an object-oriented system under test, $SP \in Spec$ its specification and $Exhaust_{SP}$ an exhaustive test set obtained from SP. We have:

$$(P \models SP) \Leftrightarrow (P \models_O Exhaust_{SP})$$

Proof*: can be found in [4].*

6.3 Practical Test Selection

In practice the test selection mechanism proposed by Péraire and Buffo consists of a sequential application of constraint predicates to the variables of an HML formula.

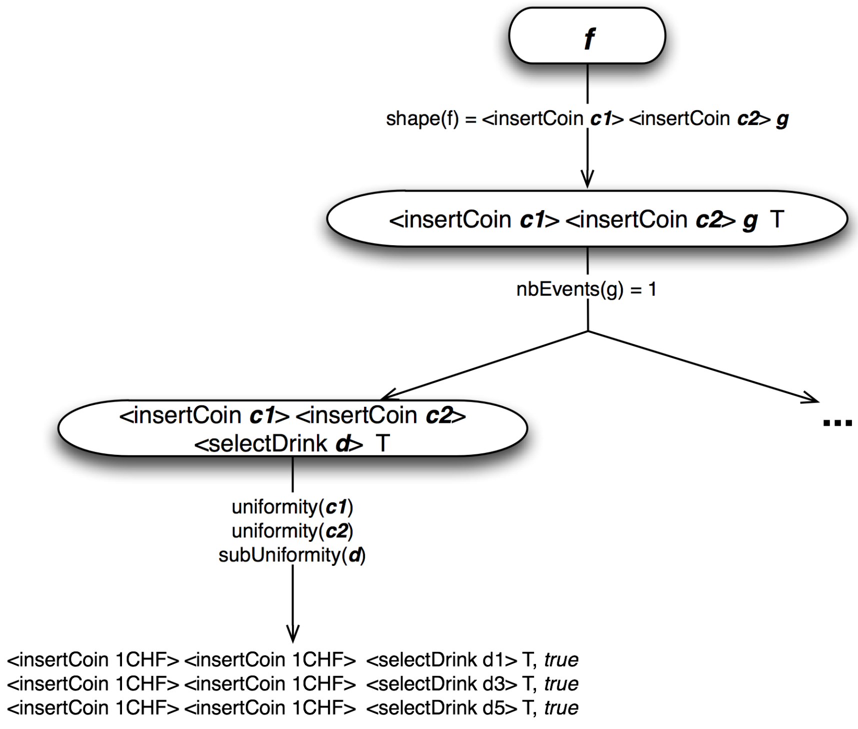

f
shape(f) = <insertCoin c1> <insertCoin c2> g
<insertCoin c1> <insertCoin c2> g T
nbEvents(g) = 1
<insertCoin c1> <insertCoin c2> <selectDrink d> T
...
uniformity(c1)
uniformity(c2)
subUniformity(d)
<insertCoin 1CHF> <insertCoin 1CHF> <selectDrink d1> T, true
<insertCoin 1CHF> <insertCoin 1CHF> <selectDrink d3> T, true
<insertCoin 1CHF> <insertCoin 1CHF> <selectDrink d5> T, true

Figure 6.3: Test Selection example using Peraire and Buffo's Contraint Language

Figure 6.3 describes an example of test selection using the technique of Péraire and Buffo. In order to select *test cases* for the DVM SUT we have presented in figure 6.1 we start by declaring a variable '*f*' which can be instantiated to any HML formula over the operations of the specification. At this moment we have the exhaustive test set.

The first reduction step (at the top of the figure) applies a '*shape*' predicate to variable '*f*', thus forcing the prefix of any test set to start by two '*insertCoin*' operations. Notice that the parameters of the *insertCoin*' operations are variables of type *money* (in bold font), and '*g*' is a variable of type HML.

The second reduction step applies a '*nbEvents*' predicate to the '*g*' variable, reducing the available test cases to those having three events. A possible formula resulting from the application of this predicate in the one presented on the inferior left side of figure 6.3. Many other formulas would be possible and they would correspond to other test cases having three events.

The last reduction step consists of performing *uniformity* and *subUniformity* hypothesis about the remaining variables. Notice that the *subUniformity* predicate chooses three behaviors for the '*selectDrink*' operation — when two coins were introduced in the DVM — as we have informally specified in section 6.1: choosing a drink which cost is inferior to 2CHF at distributing the drink and change; choosing a drink which cost is 2CHF and distributing the drink with no change; choosing a drink which cost is superior to 2CHF and issuing a "*not enough money*" message.

Several other predicates other than the ones presented in figure 6.3 were developed and are described in [4].

In terms of formalization of the approach Péraire and Barbey have presented in [4, 5] an abstract syntax and a semantics for their approach. The semantics of the dynamic part of the *test case* calculation — the one involving the *subUniformity* predicate which selects among the possible behaviors of a CO-OPN$_{/2}$ *method* at a given state — is stated operationally in Prolog. This operational semantics is based on the *unfolding* method proposed by Marre in [30]. Briefly, the decomposition consists of finding several possible calls of the CO-OPN$_{/2}$ method chosen for unfolding by analyzing the conditions that allow it to fire. All these conditions are built using operations expressed as positive conditional axioms which means they can be translated into Prolog. One instantiation of the variables for each of the positive conditional axiom is then enough to expose sub-domains of that operation. If the positive conditional axioms are defined by additional conditions, those conditions can also be unfolded increasing the precision of the behaviors found for the CO-OPN$_{/2}$ method being unfolded.

6.4 Criticism of the current BGM/CO-OPN testing approach

In this section we will provide the motivation for developing the test language SATEL we introduce in this thesis. SATEL is a natural evolution of the previous work on test generation from CO-OPN$_{/2}$ specifications we have introduced in the present chapter. The main goal while developing SATEL was to provide a fashion of stating hypothesis about an SUT which goes in the direction of *test construction*, rather than *test selection*. By *test construction* we mean that the test engineer *builds* tests to cover certain understood sub-behaviors of the SUT, rather than selecting them from the '*pool*' of all available behaviors. While the holistic *test selection* approach proposed by Péraire and Barbey works for SUTs with a small amount of behaviors which are easily understood and generalizable (see the *industrial production cell* case study [4]), the '*divide and conquer*' approach is more reasonable while testing SUTs exhibiting complex behavior.

Nonetheless, we wish to keep the *validity* and *unbiasedness* of the *test set* constrained using SATEL. This can be achieved under certain assumptions and allows us to stay in the same BGM theoretical framework we have introduced in section 6.1.

In the text that follows we will introduce some additional and more detailed reasons for introducing SATEL.

6.4.1 Observability issues

In [4] Péraire describes the *Co-opnTest* tool developed to implement the test selection activity based on the constraint language we have described in section 6.3. In the case study it is clear that the observation of the SUT during the testing activity causes a major problem. The work of Péraire is based on CO-OPN$_{/2}$, where a specification event corresponds to a method call. The technique is thus not adapted to testing systems that produce output messages during computation as these messages are difficult to model at the level of the specification. The same problem is present in the test selection for the DVM we have presented in section 6.3, where the events include no outputs (drink and change distribution, *not enough money*). In these situation the only observation that can be performed after issuing each method call in the test is the *blocking* or *non-blocking* of the SUT. In order to cope with the difficulty of effectively observing the state of the SUT two possibilities arise:

- Extending the specification (and consequently the SUT) with *observer* methods that allow checking that the SUT is in a given state. These kind of methods cannot change the state of the SUT. This is a moderately interesting solution as it implies building models and SUTs with more functionalities than strictly required, which not only places a burden on the analysis and development phases as it can also induce additional errors;

- If the SUT produces output messages (which is the general case), delegate to the test driver the task of associating those output messages to the corresponding method calls. This was done in the *industrial production cell* case study in [4], but under the assumption that there is a *determinism* between input and output messages of the SUT. This is a very restrictive assumption, as most of the times an SUT reacts to an input message in several ways, depending on its internal state. In particular, if an SUT cannot execute a required command and issues an error output message, the decision if an error was uncovered or not by the test can be better taken if the error message is observed — rather than just relying on a subsequent blockage of the SUT.

As we have explained in section 4.2, CO-OPN$_{/2c++}$ includes the notion of *gate*. A *gate* can be seen as either a required service by a CO-OPN component, or, in case the service is required from the environment, as an *ouput*. Given that in the CO-OPN$_{/2c++}$ version events may synchronize a *method* with a *gate* call, it becomes then natural to consider a '*required* ***with*** *provided*' event (see definition 4.4.10) as an input/output pair. In this way we can directly model with CO-OPN$_{/2c++}$ the observations available in the SUT, partially avoiding the problems we have mentioned in the items above. We say *partially* because the SUTs may be *active* rather than *reactive*, and *observer* methods may still be required.

6.4.2 Expressivity and usability of the Constraint language

In their thesis, Péraire and Barbey define the theoretical context and a practical framework for selecting test sets from CO-OPN$_{/2}$ specifications by successively applying constrains on the exhaustive test set — the possible set of HML formulas where the predicates are the available method calls from the specification. As we have explained in section 6.3, the constraints correspond to predicates that select formulas having certain syntactic properties at the HML or at the method name levels. There is however a syntactic layer of the HML formulas involving *data values* and *object references*, which corresponds to the parameters of the method calls. In their work, Péraire and Barbey constrain method parameters either by stating *uniformity hypothesis* or *uniformity hypothesis with subdomain decomposition*[1]. In the former case the coverage of the behaviors expressed in the specification is very low due to the fact that only one of the values of the domain is chosen; in the latter case the coverage is good, as one value is chosen per possible behavior stated in the CO-OPN$_{/2}$ specification. It is however not possible to state *regularity hypothesis* about data values. We feel this is a missing feature as the *test engineer* may have additional information about the SUT that is either not specified, or cannot be deduced from the specification using the operational methods to uncover the equivalence subdomains.

Another shortcoming we find in the previously proposed language is tied to the fact that, in the practical world, when testing SUTs exhibiting complex behavior, test cases are *built*, rather than obtained by *reducing* the exhaustive test set. Also, operationally it is in general not possible to reduce the exhaustive test set as it typically involves an infinite number of HML formulas. In this sense the constraint language proposed by Péraire and Buffo is awkward to use from a methodological point of view. This is due to the fact that, in order to test multiple behaviors, the only possibility is to build a very large conjunction of HML formulas and applying

[1]Due to the fact that the operational mechanism for uncovering equivalence domains is specific to algebraic specifications, Péraire and Barbey did not study its application to object references. We also leave this study undone in this thesis and assume the *test engineer* can only state *uniformity hypothesis* about method parameters which are object references.

constraints to its variables — replicating the *test selection by context refinement* in figure 6.2. Although adapted to algebraic specifications where each behavior of the SUT is specified by a set of equations, this methodology is not suitable for model-based testing from CO-OPN specifications.

This problem is made clear by the *industrial production cell* case study presented in [4] where, in order to produce a test set for a given component, several intermediate sets of hypothesis are expressed. Each set of hypothesis is aimed at testing a particular behavior of the component and the final test set corresponds to the union of all intermediate test sets.

While *building* test cases seems to contradict the *reduction* principle of the BGM theory that allows maintaining *validity*, under certain *decomposition hypothesis validity* can be kept. Taking the idea of *building* test sets further, it would also be profitable to a *test engineer* to introduce mechanisms allowing *composition*, *reuse* and possibly *parameterization* of test sets.

6.4.3 Unit / Integration / System Testing

The work on test generation developed by Péraire and Barbey was based on CO-OPN$_{/2}$ which allows the specification of object networks where the objects communicate by means of synchronizations. Accordingly, the notion of the limits of the specification in this context is somewhat vague, as any method available in any object can be called. Péraire and Barbey have dealt with this issue by introducing the notion of the *focus* of the test, which limits the region of interest of the specification that models the SUT. The *focus* can then be an object or a set of objects. Within the selected object(s), the set of available methods can also be selected. The remaining parts of the system not covered by the *focus* can either be replaced by tested implementations of the components, or *stubs*.

As can be seen by the *industrial production cell* case study, this kind of methodology can be successfully applied to the testing of small distributed and concurrent systems. However, we question the usability of such methodology while testing *application software* [61] (as opposed to *system software*), which is normally built modularly and in a hierarchical fashion. We feel that in this case the testing framework can largely benefit from the *context modules* that have been introduced with CO-OPN$_{/2c}$ (see section 4.4.6). *Context modules* may serve as clear interfaces to encapsulate the functionality under test. In that same line of thought, they can be used to ease the definition of *unit*, *integration* and *system* testing in our framework by clearly defining borders between components at the same and at different hierarchical layers.

6.4.4 Formalization of the Approach

A formalization of the semantics of the approach by Péraire and Barbey was presented in [4, 5]. In our view this formalization is incomplete, as it is only addressed at the *static* part of test selection, leaving the dynamic part to the operational Prolog mechanism performing the unfolding. The Prolog mechanism allows the symbolic exploration of CO-OPN$_{/2}$ specifications by encoding both CO-OPN$_{/2}$ semantics and the unfolding mechanism in the same program. Unfolding as explained in section 6.3 can then be performed on both the operators stating the conditions that enable Petri Net transitions and the conditions of the specification itself.

Although operationally interesting, this pragmatic view on the semantics of sub-domain decomposition does not allow a holistic vision on test selection. We think a study in the area involving formal mathematical techniques will provide a deeper insight on the subject — by allowing a clearer view on the subject that the one introduced by the Prolog program developed during the previous work.

6.5 Summary

In this chapter we have provided a state of the art of model based testing from CO-OPN specifications. This second state of the art motivates our work and complements chapter 2 which explores model based testing in general.

We have started by providing an account of the BGM (Bernot, Gaudel, Marre) theory of testing which establishes a formal framework for test selection from any specification formalism[2]. BGM proposes reducing an initial *exhaustive* and possibly infinite test set for a given specification by formulating a number of well understood hypothesis about how to test the SUT. If the hypothesis hold, then the reduced test set will have the property of being *pertinent*, which means that '*the reduced test set will not find errors in a* correctSUT' and '*the reduced test set will find all errors in an* SUT'. Clearly, since testing is a practical activity it becomes necessary that the reduced test set is small enough to be applied to the SUT in a reasonable amount of time.

We have then discussed how the BGM framework was applied in the context of CO-OPN$_{/2}$ specifications. We start by establishing the concept of *equivalence* between CO-OPN$_{/2}$ specifications which is the *bisimulation* between the transition systems denoting the semantics of those specifications. We then define the *exhaustive test set* for a CO-OPN specification SP. This set consists of all HML formulas built using the possible events of SP and annotated with a *true* or *false* logic value.

[2]although most of the results are given in terms of *algebraic specifications*.

The logic value states if a particular HML formula is a *valid* or *invalid* behavior (or property) of SP. An important result ends this section, proving that HML is sufficiently expressive to built an *exhaustive* and thus *pertinent* test set for CO-OPN$_{/2}$ specifications.

Practical test selection using the BGM framework and CO-OPN$_{/2}$ is then discussed. Péraire and Barbey have approached the problem of reducing the *exhaustive test set* by: defining a language of HML formulas with variables representing several the dimensions of a test case; defining a *constraint language* for the variables in those formulas. The purpose of the constraint language is to allow the expression of hypothesis about how to test the SUT. In operational terms test selection is performed using Prolog, using the unfolding technique introduced by Marre in [30]. This technique allows to dynamically find sub-domains for CO-OPN method calls, based on the conditions that allow methods to fire. By using the *subUniformity* predicate in the *constraint language* the test engineer can force this dynamic calculation of the sub-domains — thus expressing additional hypothesis about the SUT by using the semantics of the behavior of the SUT expressed in the specification.

Finally we motivate our work for introducing a new testing language, extending and modifying the *constraint language*. Partially the motivation is due to the fact that a new version of CO-OPN — CO-OPN$_{/2c++}$ which we have introduced in section 4.4 and chapter 5 — allows better modeling of *outputs* as well as modular and hierarchic specifications. We have decided to profit from the new version of CO-OPN to develop a new test language emphasizing test set *construction* (rather than selection), *reuse* and *composition*. We are also interested in defining *unit*, *integration* and *system* testing in the CO-OPN context. Finally, we establish the motivation for performing a formal study on *sub-domain decomposition* which was empirically approached by the previous work.

Chapter 7

SATEL – Introduction and Abstract Syntax

In chapter 6 we have presented previous work on model-based testing from CO-OPN specifications. We have also shown in section 6.4.3 that that work can be improved in several points, among which:

- it tries to closely replicate the BGM theoretical approach, not taking into consideration that, methodologically, it is very difficult for a test engineer to devise a number of reduction hypothesis producing a *pertinent test set* — while starting from the exhaustive *test set* for a CO-OPN specification (see definition 6.2.7);
- CO-OPN$_{/2}$ is not ideal to model reactive systems as *outputs* cannot be directly specified;
- no mechanisms exist for *reusability* and *composition* of *test sets*;
- the lack of structure and hierarchy in CO-OPN$_{/2}$ specification makes it difficult to isolate parts of the specification for integration testing.

In the current chapter we will present SATEL (Semi-Automatic TEsting Language), a language designed to improve the current state of the art of model-based testing from CO-OPN specifications. SATEL allows expressing *test intentions* about CO-OPN specifications. With the notion of *test intention* we keep the idea of applying a set of reduction hypothesis in the previous work, but we narrow those hypothesis to subsets of the behavior of the SUT.

Figure 7.1 depicts how test intentions relate to an SUT. The lower outer ellipse represents the full behavior of the SUT and the interior smaller ellipses represent

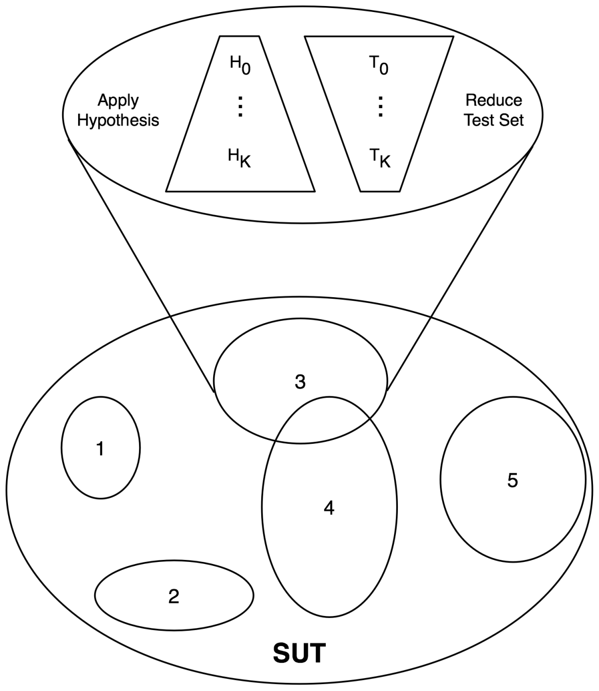


Figure 7.1: *Test Intentions* and the SUT

parts of that behavior. Each test intention corresponds to a set of reduction hypothesis for one of those parts of the behavior, as can be seen in the schema inside the upper ellipse of figure 7.1 representing the test set reduction process. If we take the example of the *Banking server* presented in section 4.3, we could for instance wish to only test the *login* behavior part of the system. If we assume 1000 users of the system, an exhaustive test set would imply 1000 test cases for the registered users plus a number of test cases for unregistered ones. A uniformity hypothesis on the user would state that testing the *login* behavior with only one registered and one unregistered user is enough and would reduce the original 1000+ test cases to two.

The advantage of this approach is that our test sets deduced from a given test intention are still *pertinent* while considering the sub-functionality they aim at testing. Also, given that different test intentions can cover different functionalities expressed in the model, we have also designed the language in a way that test intentions can be reused and composed.

7.1 Overview of SATEL

SATEL allows stating test intentions by constraining variables that represent measurable dimensions of the part of the SUT to be tested — following the ideas previously developed by Buchs, Péraire and Barbey [4, 5]. However, the new notion of *test intention* and the use of CO-OPN$_{/2c++}$ implies important differences between SATEL and the previous constraint language (see section 6.4.3), chiefly among which:

- All the possible operations of the SUT are seen as the events allowed by a single CO-OPN *context*. This allows taking advantage of the notion of context present in CO-OPN$_{/2c++}$ in order to represent a clear interface between the SUT and its environment;
- In CO-OPN$_{/2c++}$ events are extended by *gates*. SATEL also uses this augmented notion of event which facilitates observing the SUT by pairing *inputs* and *outputs*. In the context of testing, we will subsequently use the terms *stimulation* for SUT input and *observation* for SUT output;
- New mechanisms are defined to allow a finer grain while defining *Regularity* and *Uniformity* hypothesis. In particular, additional language constructs were developed to allow constraining *stimulations*, *observations* and their respective parameters;
- Test intention modules are introduced as legitimate CO-OPN$_{/2c++}$ modules.

7.1.1 Regularity and Uniformity Hypothesis in SATEL

A *test intention* is written as a set of *Hml formulas* with variables, which in the subsequent text we will call *execution patterns*. The variables correspond to three structural dimensions of a test case, namely:

- the ***shape of the execution paths***;
- the ***shape of stimulation/observation pairs*** inside an execution path;
- the ***shape of the parameters*** of stimulations or observations.

The domains of these variables are obtained from the signature of the CO-OPN model of the SUT.

A *test intention* is thus written as a set of partially instantiated execution patterns, where the variables present in those patterns are by default universally quantified. All the combined instantiations of the variables will produce a (possibly infinite) number of test cases. If no conditions are imposed on the variables they remain universally quantified, which means we obtain the exhaustive test sets for the behavior covered by the *test intention*. By constraining the domains of the variables present in an execution pattern by using *regularity* and *uniformity* hypothesis, the test engineer is able to reduce the exhaustive test set per *test intention*.

Regularity hypothesis

As explained in the following points, for each kind of domain we have used a number of functions and predicates that allow modeling regularity hypothesis.

- **Shape of execution paths**: variables representing the shape of execution paths are constrained by using functions and predicates that discriminate the of shape Hml formulas. We propose the following integer functions: *nbEvents* — number of events in an Hml formula; *depth* — length of the deepest branch of an Hml formula; *nbOccur* — number of occurrences of a given method in an Hml formula. We also propose the following boolean functions: *sequence* — *true* if the Hml formula contains no *and* operators; *positive* — *true* if the Hml formula contains no *not* operators; *trace* — *true* if the Hml formula contains no *and* or *not* operators;

- **Stimulations/Observations**: *stimulations* and *observations* are obtained from the signature of method and gate ports and using the typical $\{//, .., \oplus\}$ synchronization operators. We propose a single integer function *nbSynchro* that returns the number of events in the stimulation or observation;

- **Parameters of stimulations or observations**: given the fact that these variables represent values defined by the signature of CO-OPN ADT modules, we use algebraic equations in order to constrain the elements from those sets. If we take the example of figure 4.3, a possible algebraic constraint would be *(a = newPassword 1 2 3 4) = true*, which would limit an algebraic variable called a to the only possible value of *(newPassword 1 2 3 4)*.

The integer functions return a value measuring certain kinds of complexity of the structural dimensions of a test case. Predicates over integers (*equals*, *different*, *greater than*, *smaller than*, *greater or equal*, *smaller or equal*) are then used to constrain variables representing those dimensions. For example, the condition *nbEvents(f)<5* represents a regularity hypothesis over an Hml formula variable f, stating that it cannot be instantiated into an execution path which has more than five events. Boolean functions as well as algebraic equations are directly used as predicates in SATEL.

In fact, the kind of predicates we have presented to model regularity hypothesis can also be used to model uniformity hypothesis, if we apply them in a way to limit the instantiation of the variable to a single value. However, the value will be found in a deterministic fashion.

The predicates for constraining the *shape of execution paths* are the same as in [4]. The remaining ones were introduced by our work.

Uniformity Hypothesis

Uniformity and *uniformity with sub-domain decomposition* hypothesis are expressed in SATEL much in the same way as they are in the previous work on test generation from CO-OPN specifications. We use the unary predicates *uniformity* and *subUniformity* that have as parameter a SATEL variable of any type. While the *uniformity* predicate chooses one single random value from the variable's domain, the *subUniformity* predicate chooses values from the variable's domain that allow testing the possible behaviors of the CO-OPN method calls involved in the test intention.

As an example of applying a *uniformity with sub-domain decomposition* hypothesis, consider the following execution pattern which is part of a test intention for the *Banking Server* described in section 4.3:

```
<deposit(10) with null> <(withdraw amount) with g> T
```

In this execution pattern, *amount* is a variable of the ADT *integer* type and *g* is a variable of type *observation*. By applying a *uniformity with sub-domain decomposition* hypothesis to variable *amount*, we would produce the following *positive* test cases by decomposing the several behaviors of the *withdraw* operation:

⟨`<deposit(10) with null> <withdraw(5) with null> T`, *true*⟩
⟨`<deposit(10) with null> <withdraw(15) with noMoneyLeft> T`, *true*⟩

In fact the *subuniformity* predicate allows choosing two values for the *amount* variable, one for each fire condition of the method *withdraw* — see in appendix B the class *Account* where we have defined two axioms for *withdraw* method with the complementary conditions ($b >= amount$) = *true* and ($b >= amount$) = $false$. We choose one value satisfying each of those conditions, hence covering the two possible behaviors of method *withdraw*.

7.1.2 Declaring test intention rules

We will now briefly introduce how to declare test intentions using the concrete syntax of SATEL. This can be done by defining *test intention rules*, each rule having the form:

```
[ condition => ] inclusion
```

In the *condition* part of the rule (which is optional) the test engineer is able to express conditions over variables. In the *inclusion* part of the rule the test engineer can express that a given execution pattern is included in a named test intention. Assuming given a CO-OPN context to test, consider the following rule written in SATEL's concrete syntax (where f is a variable over the shape of execution paths):

```
nbEvents(f) < 5 => f in SomeIntention;
```

This rule would produce all possible test cases for that context that include a number of events inferior to 5. These test cases would become part of the test set generated by the test intention *SomeIntention*.

On the other hand it is possible to declare multiple rules for the same test intention. Let us add to the previous rule the following one, where *aMethod* and *aGate* are respectively stimulations and observations without any variables:

```
Hml({aMethod with aGate} T) in SomeIntention;
```

The set of test cases produced by *SomeIntention* would now become the one produced by the first rule united with the one produced by the second rule. In fact only one test case is produced by the second rule given that there are no variables in the execution pattern '*Hml<aMethod **with** aGate> T*'.

An interesting feature of the language is that it allows reusing rules by composition, as well as recursion between rules. Consider the following set of rules where f and g are variables over the shape of execution paths and f . g in the third rule means f *concatenated* to g:

```
            Hml({aMethod with aGate} T) in AnotherIntention
           Hml({aMethod' with aGate'} T) in AnotherIntention
f in AnotherIntention, g in AnotherIntention => f .  g in AnotherIntention
```

These rules would produce an infinite amount of test cases which include sequences of the stimulation/observation pairs <*aMethod **with** aGate*> and <*aMethod' **with*** *aGate'*> in any order and in any length. In fact, the third rule for *AnotherIntention* chooses non-deterministically any two test cases generated by any rule of the test intention and builds a new test case based on their concatenation.

7.2 SATEL example – testing the *Banking Server*

In figure 7.2 we provide a full example of usage of our test intention language for defining a test intention module for the Banking Server system we defined in section 4.3. The example is written in the concrete syntax of SATEL which can be found in appendix E.

The test intention module *TestBanking* acts over the *BankingServer* context (as defined in the *Focus* field) and defines several distinct test intentions declared in the fields *Intentions*. The types for those variables are declared in the *Variables* field.

Let us now describe the axioms involved in the definitions of the test intentions present in the *TestBanking* test intention module of figure 7.2. This will provide the reader with an informal notion of the semantics of SATEL.

- ***nWrongPins*** (lines 18–20) makes use of recursion in order to build (possibly infinite) sequences of erroneous introductions of passwords by user '*d*' (we assume a user '*d*' exists in the SUT and his password is different from '*newPassword 0 0 0 0*'. Two axioms are used in order to define this test intention: the one in line 18 is used to provide a base case for the recursion, stating that the empty execution pattern — represented by '*T*' — is part of the set of execution patterns needed to cover the *nWrongPins* functionality.

  The axiom in line 20 defines that any concatenation of an execution pattern for *nWrongPins* with the '*<loginUserPass(newString(d), newPin(0 0 0 0)) with badPassword>*' event is also in the set of execution patterns necessary to cover the *nWrongPins* functionality. These two axioms recursively build an infinite set of execution patterns containing increasingly longer sequences of wrong password insertion events.

  A particularity of this test intention is that it is defined in the *Body* section of the test intention module. Test intentions declared inside the *Body* section are *auxiliaries* for building other test intentions and will not directly produce test cases, as this is done only for test intentions declared in the *Interface* section.

- **insertPasswords** (line 22) uses the previously defined execution pattern *nWrongPins* in order to build all possible test cases for behaviors of the *insertPassword* operation. Given that the system will block a user after insertion of 3

```
 0 TestIntentionSet TestBanking Focus Banking;
     Interface
 2     Intentions
         insertPasswords;
 4       withdrawMoney;
         parallelLogin;
 6
     Body
 8     Intentions
         nWrongPins;
10
       Use
12       Natural;
         String;
14       Pin;
         Challenge;
16
       Axioms
18       T in nWrongPins;

20       path in nWrongPins => path . Hml({insertPassword(d, newPin(0 0 0 0)) with
             badPassword} T) in nWrongPins;

22       path in nWrongPins , nbEvents(path) < 4 => Hml({login(d) with askChallenge
             (chal)} T) . path . Hml({insertPassword(d,convertChal(chal)) with null
             } T) in insertPasswords;

24       subUniformity(am) | path in insertPasswords , nbEvents(path) = 3 => path
             . Hml({deposit(d,100) with null} {withdraw(d,am) with obs} T) in
             withdrawMoney;

26       sequence(path), nbEvents(path) <= 3 => Hml({login(d) // login(e) with obs}
              T) . path in parallelLogin;

28     Variables
         am : natural;
30       chal : string;
         obs : primitiveObservation;
32       path : primitiveHml;

34 End TestBanking;
```

Figure 7.2: Test Intentions for the Banking Server

wrong passwords, we use a variable representing an execution pattern from the *nWrongPins* while limiting it to a maximum of 3 events. We concatenate it on the left with a *login* event and on the right with an *insertPassword* event. In this way we are able to test behaviors including none, one or two password mistakes — which should reach a logged in state for user *d* — and three password mistakes — which should reach a blocked state for user *d*.

- **withdrawMoney** (line 24) builds test cases for the behaviors of the *withdraw* operation. Variable *path* of type Hml includes an execution pattern defined in the *insertPasswords* test intention with three events — the last one being the correct password insertion. After these events a *deposit* of value 100 can be executed and the following *withdraw* event includes a variable *am* over which there is a uniformity with sub-domain decomposition hypothesis. This hypothesis will find two values for variable *am*: one under 100 (successful withdrawal) and another over 100 (unsuccessful withdrawal).

- **parallelLogin** (line 26) builds test cases for the simultaneous login of two users (notice the usage of the // operator). It is provided in the text as a way of demonstrating that SATEL allows the testing of concurrency. The event where the two users are logged in is followed by an Hml variable *path* representing an execution path limited to a maximum of three events in sequence (meaning no *and* operators are allowed in the Hml formulas *path* will be instantiated into).

7.3 Abstract Syntax of SATEL

The abstract syntax of SATEL will be described in three steps — in the first step we will formalize the syntax of *execution patterns* as Hml temporal logic formulas. As previously explained, these formulas will contain variables representing the various dimensions of an SUT's behavior.

In the second step we will formalize the abstract syntax of *test intention axioms*. This language allows expressing conjunctions of constraints over the variables present in execution patterns.

In the third step we will formalize the abstract syntax of *test intention modules* which is integrated with the abstract syntax of the CO-OPN$_{/2c++}$ language we have described in section 4.4.

Let us remind the reader that we consider the universes I, M and G corresponding to the set of test intention, method and gate names.

7.3.1 Abstract Syntax of Execution Patterns

We will build the abstract syntax of execution patterns in a bottom-up fashion. As previously explained in the text, execution patterns correspond to Hml formulas with variables. Given a CO-OPN context module, we will successively build the set of *event argument* terms, the set of *event* terms (divided in *stimulation* and *observation* terms) and the set of *Hml formula* terms.

Definition 7.3.1 *Event argument terms*

Let $\Sigma = \langle S, \leq, F \rangle$ be a global signature and X be an S-sorted set of variables. The set of event argument terms *over Σ with sort $s \in S$ is the set $(T_{\Sigma,X})_s$ with $s \in S$.*

Note that S includes both the sort names of the algebraic part of SP and the types of the classes of SP. In this way we build terms that may include data, object identifiers, or both.

Example 7.3.2 *Assume the Banking Server example from section 4.3 has a global signature $\Sigma = \langle S, \leq, F \rangle$. Assume also an S-sorted set X of variables. In line 20 of figure 7.2 the arguments* d *and '*newPin(0 0 0 0)*' of the '*insertPassword*' method are* algebraic terms *of type $(T_{\Sigma,X})_{user}$ and $(T_{\Sigma,X})_{pin}$ respectively. In line 24 of the same figure* am *is a variable belonging to $X_{natural}$ which is also a term of $(T_{\Sigma,X})_{natural}$.*

We will now build the set of *event terms* for a given CO-OPN context. We do this in two parts — in the first part we define the syntax of *stimulation terms* and in the second the syntax of *observation terms*. While stimulation terms are method synchronizations composed using the simultaneous ('//') synchronization operator, observation terms are method synchronizations composed using all the synchronization operators . A stimulation/observation pair corresponds to an event in the context for which the test intentions are written.

Definition 7.3.3 *Stimulation terms*

Let $\Sigma = \langle S, \leq, F \rangle$ be a global signature and $M = (M_w)_{w \in S^*}$ an S^*-sorted set of method names of **M**. *Let also X be the union of the disjoint X_{Stm} and the S-sorted X_S sets of variables. The terms of $Stim_{\Sigma,M,X}$ are built as follows:*

- $x \in Stim_{\Sigma,M,X}$ *for all* $x \in X_{Stm}$
- $m(t_1, \ldots, t_n) \in Stim_{\Sigma,M,X}$ *for all* $m_{s_1,\ldots,s_n} \in M$ *and for all* $t_i \in (T_{\Sigma,X_S})_{s_i}$ $(1 \leq i \leq n)$
- $stm \; // \; stm' \in Stim_{\Sigma,M,X}$ *for all* $stm, stm' \in Stim_{\Sigma,M,X}$

The second item of definition 7.3.3 defines all possible method calls of a given context. The arguments of those method calls may be either algebraic terms or variables. The third item defines synchronizations of method calls using the usual operators.

Example 7.3.4 *Assume the Banking Server context module in figure 4.7 has an interface* $\Xi = \langle M, G \rangle$ *and is part of a CO-OPN specification having a global signature* $\Sigma = \langle S, \leq, F \rangle$*. Assume also a set* X *of S-sorted and stimulation variables. In line 24 of figure 7.2 the* 'withdraw(d,am)' *expression is a* stimulation term *of* $Stim_{\Sigma,M,X}$.

Also in line 26 of figure 7.2 the 'login(d) // login(e)' *expression is a stimulation term which consists of a synchronization of method calls.*

Definition 7.3.5 *Observation terms*

Let $\Sigma = \langle S, \leq, F \rangle$ *be a global signature and* $G = (G_w)_{w \in S^*}$ *an* S^**-sorted set of gate names of* G*. Let also* X *be the union of the disjoint* X_{Obs} *and the (S-sorted)* X_S *sets of variables. The terms of* $Obsrv_{\Sigma,G,X}$ *are built as follows:*

- $x \in Obsrv_{\Sigma,G,X}$ *for all* $x \in X_{Obs}$
- $g(t_1, \ldots, t_n) \in Obsrv_{\Sigma,G,X}$ *for all* $g_{s_1,\ldots,s_n} \in G$ *and for all* $t_i \in (T_{\Sigma,X})_{s_i}$ $(1 \leq i \leq n)$
- $obs\ op\ obs' \in Obsrv_{\Sigma,G,X}$ *for all* $obs, obs' \in Obsrv_{\Sigma,G,X}$ *and for all* $op \in \{//, .., \oplus\}$

Example 7.3.6 *Assume the Banking Server context module in figure 4.7 has an interface* $\Xi = \langle M, G \rangle$ *and is part of a CO-OPN specification having a global signature* $\Sigma = \langle S, \leq, F \rangle$*. Assume also a set* X *of S-sorted and stimulation variables. In line 22 of figure 7.2 the* 'askChallenge(chal)' *expression is an* observation term *of* $Obsrv_{\Sigma,G,X}$ *where* $askChallenge_{char} \in G$ *and* $chal$ *is a variable of* $X_{challengeTable}$.

Finally we will define the syntax of the *execution patterns*, which are the set of Hml formula terms for a given CO-OPN context module. Besides including variables over algebraic, stimulation and observation terms, execution patterns also include variables over Hml formulas (execution paths) and the concatenation operator ('.'), which will be used to "glue" Hml formulas together. We will do this in two steps — first definition of Hml formulas and only then the execution patterns.

Definition 7.3.7 *Hml formulas with variables*

Let $\Sigma = \langle S, \leq, F \rangle$ *be a global signature,* $M = (M_w)_{w \in S^*}$ *an* S^**-sorted set of method names of* M *and* $G = (G_w)_{w \in S^*}$ *an* S^**-sorted set of gate names of* G*. Let also* X *be the union of the disjoint* X_{Hml}*,* X_{Stm}*,* X_{Obs} *and the (S-sorted)* X_S *sets of variables. The Hml formulas with variables* $Hml_{\Sigma,M,G,X}$ *are defined as follows:*

- $x \in Hml_{\Sigma,M,G,X}$ *for all* $x \in X_{Hml}$

- $T \in Hml_{\Sigma,M,G,X}$
- $(\neg f) \in Hml_{\Sigma,M,G,X}$ *for all* $f \in Hml_{\Sigma,M,G,X}$
- $(f \wedge g) \in Hml_{\Sigma,M,G,X}$ *for all* $f, g \in Hml_{\Sigma,M,G,X}$
- $(\langle s \rangle\, f) \in Hml_{\Sigma,M,G,X}$ *for all* $s \in Stim_{\Sigma,M,X}$
- $(\langle s, o \rangle\, f) \in Hml_{\Sigma,M,G,X}$ *for all* $s \in Stim_{\Sigma,M,X}$ *and for all* $o \in Obsrv_{\Sigma,G,X}$

Definition 7.3.8 *Execution patterns*

Let $\Sigma = \langle S, \leq, F \rangle$ *be a global signature,* $M = (M_w)_{w \in S^*}$ *an* S^**-sorted set of method names of* **M** *and* $G = (G_w)_{w \in S^*}$ *a* S^**-sorted set of gate names of* **G**. *Let also* X *be the union of the disjoint* X_{Hml}*,* X_{Stm}*,* X_{Obs} *and the (S-sorted)* X_S *sets of variables. The execution patterns* $Pat_{\Sigma,M,G,X}$ *are defined as follows:*

- $th \in Pat_{\Sigma,M,G,X}$ *for all* $th \in Hml_{\Sigma,M,G,X}$
- $th\,.\,th' \in Pat_{\Sigma,M,G,X}$ *for all* $th, th' \in Hml_{\Sigma,M,G,X}$

Example 7.3.9 *Assume the Banking Server context module in figure 4.7 has an interface* $\Xi = \langle M, G \rangle$ *and is part of a CO-OPN specification having a global signature* $\Sigma = \langle S, \leq, F \rangle$. *Assume also a set* X_{Hml} *of variables. In line 20 of figure 7.2 the* 'path . Hml(insertPassword(d, newPin(0 0 0 0)) with badPassword T)' *expression corresponds to the execution pattern* 'p ath . Hml(<insertPassword(d, newPin(0 0 0 0)) with badPassword> T)' *of* $Pat_{\Sigma,M,G,X}$ *where* f *is a variable belonging to* X_{Hml}.

7.3.2 Abstract Syntax of Test Intention Axioms

We will now define the abstract syntax of test intention axioms. As was already explained in section 7.1.2 a test intention axiom is composed of a *condition* and an *inclusion* part. The *inclusion* part holds an execution pattern — more precisely an Hml formula term (see definition 7.3.8) — and a *test intention* name. The *condition* part includes a conjunction of predicates that allow stating test hypothesis by constraining the variables present in the execution pattern in the *inclusion* part.

The abstract syntax of test intention axioms will be presented in a bottom-up fashion. We will start by defining the *arithmetic* and *boolean* terms of the language which were briefly introduced in section 7.1.1.

Arithmetic and Boolean terms

Besides manipulating algebraic data types (ADT), stimulations, observations and Hml formulas, SATEL includes two other data types which are the *integers* and the *booleans*. In order to provide the abstract syntax for the *arithmetic* and *boolean* terms of SATEL we assume defined the following syntactic sets:

- $Bool = \{true, false\}$
- $Num = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

Definition 7.3.10 *Arithmetic terms*

Let $\Sigma = \langle S, \leq, F\rangle$ be a global signature, $M = (M_w)_{w \in S^*}$ an S^*-sorted set of method names of $\mathbf{M}$ *and $G = (G_w)_{w \in S^*}$ an S^*-sorted set of gate names of* $\mathbf{G}$*. Let also X be the union of the X_{Hml}, X_{Stm}, X_{Obs}, X_S and X_{Num} disjoint sets of variables where X_S is S-sorted. The arithmetic terms $ATerm_{\Sigma,M,G,X}$ are defined as follows:*

- $n \in ATerm_{\Sigma,M,G,X}$ *for all* $n \in Num$
- $x \in ATerm_{\Sigma,M,G,X}$ *for all* $x \in X_{Num}$
- $depth(pat) \in ATerm_{\Sigma,M,G,X}$
- $nbEvents(pat) \in ATerm_{\Sigma,M,G,X}$
- $nbOccur(m_{s_1,\ldots,s_n}, pat) \in ATerm_{\Sigma,M,G,X}$ *for all* $m_{s_1,\ldots,s_n} \in M$
- $nbSynchro(stm) \in ATerm_{\Sigma,M,G,X}$
- $tn\, op_{Num}\, tn' \in ATerm_{\Sigma,M,G,X}$ *for all* $tn, tn' \in ATerm_{\Sigma,M,G,X}$*, where* $op_{Num} \in \{+, -, *, /\}$

where $pat \in Pat_{\Sigma,M,G,X}$, $stm \in Stim_{\Sigma,M,X}$ and $obs \in Obsrv_{\Sigma,G,X}$

Example 7.3.11 *Assume the Banking Server context module in figure 4.7 has an interface $\Xi = \langle M, G\rangle$ and is part of a CO-OPN specification having a global signature $\Sigma = \langle S, \leq, F\rangle$. Assume also a set X of variables. In line 20 of figure 7.2 the* 'nbEvents(f)' *and* '4' *expressions are both arithmetic terms of $ATerm_{\Sigma,M,G,X}$ where* 'f' *is a variable belonging to X.*

Definition 7.3.12 *Boolean terms*

Let $\Sigma = \langle S, \leq, F \rangle$ be a global signature, $M = (M_w)_{w \in S^*}$ an S^*-sorted set of method names of **M** *and $G = (G_w)_{w \in S^*}$ an S^*-sorted set of gate names of* **G**. *Let also X be the union of the X_{Hml}, X_{Stm}, X_{Obs}, X_S and X_{Bool} disjoint sets of variables where X_S is S-sorted. The boolean terms $BTerm_{\Sigma,M,G,X}$ are defined as follows:*

- $b \in BTerm_{\Sigma,M,G,X}$ *for all* $b \in Bool$
- $x \in BTerm_{\Sigma,M,G,X}$ *for all* $x \in X_{Bool}$
- $sequence(pat) \in BTerm_{\Sigma,M,G,X}$
- $positive(pat) \in BTerm_{\Sigma,M,G,X}$
- $trace(pat) \in BTerm_{\Sigma,M,G,X}$

where $pat \in Pat_{\Sigma,M,G,X}$, $stm \in Stim_{\Sigma,M,X}$ *and* $obs \in Obsrv_{\Sigma,G,X}$

Example 7.3.13 *Assume the Banking Server context module in figure 4.7 has an interface $\Xi = \langle M, G \rangle$ and is part of a CO-OPN specification having a global signature $\Sigma = \langle S, \leq, F \rangle$. Assume also a set X of variables. In line 26 of figure 7.2 the* 'sequence(f)' *expression is a boolean term of $BTerm_{\Sigma,M,G,X}$ where* 'f' *is a variable belonging to X.*

Test Intention Axioms

Let us now define the abstract syntax for test intention axioms. We will start by defining *atomic conditions* which form the *condition* part of a test intention axiom.

Definition 7.3.14 *Inclusion Conditions, Variable Quantifier Conditions, Variable Constraint Conditions and Atomic Conditions*

Let $\Sigma = \langle S, \leq F \rangle$ be a global signature, $M = (M_w)_{w \in S^*}$ an S^*-sorted set of method names of **M** *and $G = (G_w)_{w \in S^*}$ an S^*-sorted set of gate names of* **G**. *Let also X_{Hml}, X_{Stm}, X_{Obs}, X_S, X_{Bool} and X_{Num} be six disjoint sets of variables where X_S is S-sorted. Finally let I be a set of test intention names of* **I**.

The Inclusion Conditions $In_{\Sigma,M,G,X,I}$ *are:*

- $t\, in\, k \in In_{\Sigma,M,G,X,I}$ *for all* $t \in X_{Hml}, k \in I$

The Variable Quantifier Conditions $Qt_{\Sigma,M,G,X,I}$ *are:*

- $subunif(x) \in Qt_{\Sigma,M,G,X,I}$ *for all* $x \in X_{Hml} \cup X_{Stm} \cup X_S$
- $unif(x) \in Qt_{\Sigma,M,G,X,I}$ *for all* $x \in X_{Hml} \cup X_{Stm} \cup X_S$

The Variable Constraint Conditions $Cons_{\Sigma,M,G,X,I}$ *are:*

- $tn\ op_{Num}\ tn', \in Cons_{\Sigma,M,G,X,I}$ *for all* $tn, tn' \in ATerm_{\Sigma,M,G,X}$ *where* $op_{Num} \in \{==, <>, <, >, <=, >=\}$
- $tb\ bop_{Bool}\ tb' \in Cons_{\Sigma,M,G,X}$ *for all* $tb, tb' \in BTerm_{\Sigma,M,G,X}$ *where* $bop_{Bool} \in \{and, or\}$
- $uop_{Bool}\ tb \in Cons_{\Sigma,M,G,X}$ *for all* $tb \in BTerm_{\Sigma,M,G,X}$*, where* $uop_{Bool} \in \{not\}$
- $th = th', \in Cons_{\Sigma,M,G,X,I}$ *for all* $th, th' \in Pat_{\Sigma,M,G,X}$
- $ts = ts', \in Cons_{\Sigma,M,G,X,I}$ *for all* $ts, ts' \in Stm_{\Sigma,M,X}$
- $to = to', \in Cons_{\Sigma,M,G,X,I}$ *for all* $to, to' \in Obs_{\Sigma,G,X}$
- $tn = tn', \in Cons_{\Sigma,M,G,X,I}$ *for all* $tn, tn' \in ATerm_{\Sigma,M,G,X}$
- $tb = tb' \in Cons_{\Sigma,M,G,X,I}$ *for all* $tb, tb' \in BTerm_{\Sigma,M,G,X}$
- $t = t' \in Cons_{\Sigma,M,G,X,I}, \exists s \in S$ *such that* $t, t' \in (T_\Sigma)_s \cup (X_S)_s$

Finally, the Atomic Conditions $AC_{\Sigma,M,G,X,I}$ *are:*

$$In_{\Sigma,M,G,X,I} \cup Qt_{\Sigma,M,G,X,I} \cup Cons_{\Sigma,M,G,X,I}$$

Notice that the last six items of the definition of *Variable Constraint Conditions* (see definition 7.3.14) are equalities between terms of the same kind. The equalities allow defining additional constraints on variables.

Example 7.3.15 *Assume the* Banking *context module in figure 4.7 has an interface* $\Xi = \langle M, G\rangle$ *and is part of a CO-OPN specification having a global signature* $\Sigma = \langle S, \leq, F\rangle$*. Assume also a set* X *of variables and a set* I *of test intention names. In line 24 of figure 7.2 the '*subUnif(am)*', '*f in insertPasswords*' and '*nbEvents(f) = 3*' expressions are atomic conditions of* $AC_{\Sigma,M,G,X,I}$ *where '*am*' and '*f*' are variables belonging to* X*.*

Definition 7.3.16 *Axiom condition*

Let $\Sigma = \langle S, \leq F\rangle$ be a global signature, $M = (M_w)_{w\in S^*}$ an S^*-sorted set of method names of **M** *and $G = (G_w)_{w\in S^*}$ an S^*-sorted set of gate names of* **G**. *Let also X_{Hml}, X_{Stm}, X_{Obs}, X_S, X_{Bool} and X_{Num} be six disjoint sets of variables where X_S is S-sorted. Given a a set $I \subseteq$* **I** *of test intention names, the conditions $Cond_{\Sigma,M,G,X,I}$ are defined by the powerset of atomic conditions $AC_{\Sigma,M,G,X,I}$ as follows:*

$$Cond_{\Sigma,M,G,X,I} = \mathcal{P}(AC_{\Sigma,M,G,X,I})$$

Example 7.3.17 *Assume the* Banking *context module in figure 4.7 has an interface $\Xi = \langle M, G\rangle$ and is part of a CO-OPN specification having a global signature $\Sigma = \langle S, \leq, F\rangle$. Assume also a set X of variables and a set I of test intention names. In line 22 of figure 7.2 the '*f in nWrongPins , *$nbEvents(f) < 4$' expression is an axiom condition in SATEL's concrete syntax. In the abstract syntax presented in definition 7.3.16 this axiom condition corresponds to the set* {f in nWrongPins , *$nbEvents(f) < 4$*} *of $Cond_{\Sigma,M,G,X,I}$ where* f *is a variable belonging to X.*

We now have the elements to define the abstract syntax of complete test intention axioms, which we have informally introduced in section 7.1.2.

Definition 7.3.18 *Test intention axiom*

Let $\Sigma = \langle S, \leq F\rangle$ be a global signature, $M = (M_w)_{w\in S^*}$ a S^*-sorted set of method names of **M** *and $G = (G_w)_{w\in S^*}$ a S^*-sorted set of gate names of* **G**. *Let also X_{Hml}, X_{Stm}, X_{Obs}, X_S, X_{Bool} and X_{Num} be six disjoint sets of variables where X_S is S-sorted. Given a a set* **I** *of test intention names, the axioms $Axiom_{\Sigma,M,G,X,I}$ are defined as the following cartesian product:*

$$Axiom_{\Sigma,M,G,X,I} = Cond_{\Sigma,M,G,X,I} \times Pat_{\Sigma,M,G,X} \times I$$

Example 7.3.19 *Assume the* Banking *context module in figure 4.7 has an interface $\Xi = \langle M, G\rangle$ and is part of a CO-OPN specification having a global signature $\Sigma = \langle S, \leq, F\rangle$. Assume also a set X of variables and a set I of test intention names. Consider the following test intention axiom in line 20 of figure 7.2:*

f in nWrongPins =>f . Hml({insertPassword(d, newPin(0 0 0 0)) with badPassword} T) in nWrongPins;

In the abstract syntax presented in definition 7.3.16 this expression corresponds to following triplet, where f *is a variable belonging to X:*

$\big\{$\{f in nWrongPins\}, f . <insertPassword(d, newPin(0 0 0 0)) with badPassword T>, nWrongPins$\big\} \in Axiom_{\Sigma,M,G,X,I}$

7.3.3 Test Intention Modules

Let us define the notion of *test intention module*. These modules are orthogonal to the remaining CO-OPN modules in the sense that *ADT*, *class* or *context* modules are used to *model* a system, while *test intention* modules are used to *verify* it. Nonetheless, these two dimensions of the construction of the model of a system are intertwined because the signature and behavioral part of the model — described by the *ADT*, *class* and *context* modules — is used as a way of syntactically building test intentions and also as a means of producing an *oracle* for the Hml formulas produced from those test intentions.

As happens with *ADT*, *class* and *context* modules, *test intention* modules also include the notion of *interface* where the services provided to the exterior of the module are listed. In order to allow inter-module test intention compositionality, it becomes necessary to expose to other *test intention* modules the set of test intentions declared in a particular *test intention* module.

Definition 7.3.20 *Test intention module interface*

A test intention module interface Γ *is a set of test intention names* $I \subseteq \mathbf{I}$.

Example 7.3.21 *The interface* $\Gamma_{TestBanking}$ *for the* TestBanking *context module in figure 7.2 is the set* $\{insertPasswords, withdrawMoney, parallelLogin, nWrongPins\}$.

A *test intention* module is then composed of an *interface*, but also includes: the interface of the *context* module over which the test intentions are stated; a set of axioms declaring test intentions.

Definition 7.3.22 *Test Intention Module*

Let $\mathbf{\Sigma}$ *be a set of ADT module signatures and* $\mathbf{\Omega}$ *be a set of class module interfaces such that the global signature* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle S, \leq, F \rangle$ *is complete. Let also* Ξ *be a set of context module interfaces and* $\mathsf{\Gamma}$ *a set of test intention module interfaces. A test intention module is a sextuplet* $Md^{\mathsf{T}}_{\mathbf{\Sigma},\mathbf{\Omega},\Xi,\mathsf{\Gamma}}$=$\langle \Gamma, Focus, \Lambda, X \rangle$ *where:*

- Γ *is a test intention module interface;*
- $Focus \in \Xi$ *(where* $Focus = \langle M, G \rangle$*) is the interface of the context module under test;*
- $\Lambda \subseteq Axiom_{\Sigma,M,G,X,I}$ *is a set of test intention axioms, where* $I = \Gamma \cup \bigcup_{1 \leq i \leq n} \Gamma_i$ $(\Gamma_i \in \mathsf{\Gamma})$*;*

- *X is the union of the disjoint X_{Hml}, X_{Stm}, X_{Obs}, X_{Num}, X_{Bool} and X_S sets of variables where X_S is S-sorted.*

Notice that while building Λ in definition 7.3.22 we use a set I of test intention names which includes the test intention names of the interface of the module itself (Γ) and the test intention names imported from other modules ($\mathsf{\Gamma}$).

Example 7.3.23 *Let $\mathsf{\Sigma}$ be a set of ADT module signatures and $\mathsf{\Omega}$ be a set of class module interfaces used in the* Banking Server *specification. Let also $\Xi_{Banking}$ be a context module interface and $\Xi = \{\Xi_{Banking}\}$. The* TestBanking *test intention module in figure 7.2 corresponds to the abstract syntax $(Md^{\mathsf{T}}_{\mathsf{\Sigma},\mathsf{\Omega},\Xi,\varnothing})_{TestBanking} = \langle \Gamma_{TestBanking}, \Xi_{Banking}, \Lambda_{TestBanking}, X_{TestBanking} \rangle$, where:*

- *$\Gamma_{TestBanking}$ is defined in example 7.3.21;*
- *$\Lambda_{TestBanking}$ is partially defined in example 7.3.17;*
- *$X_{TestBanking} = X_{Hml} \cup X_{Obs} \cup X_S$, where $X_{Hml} = \{f\}$, $X_{Obs} = \{obs\}$, $(X_S)_{string} = \{chal\}$ and $(X_S)_{natural} = \{am\}$;*

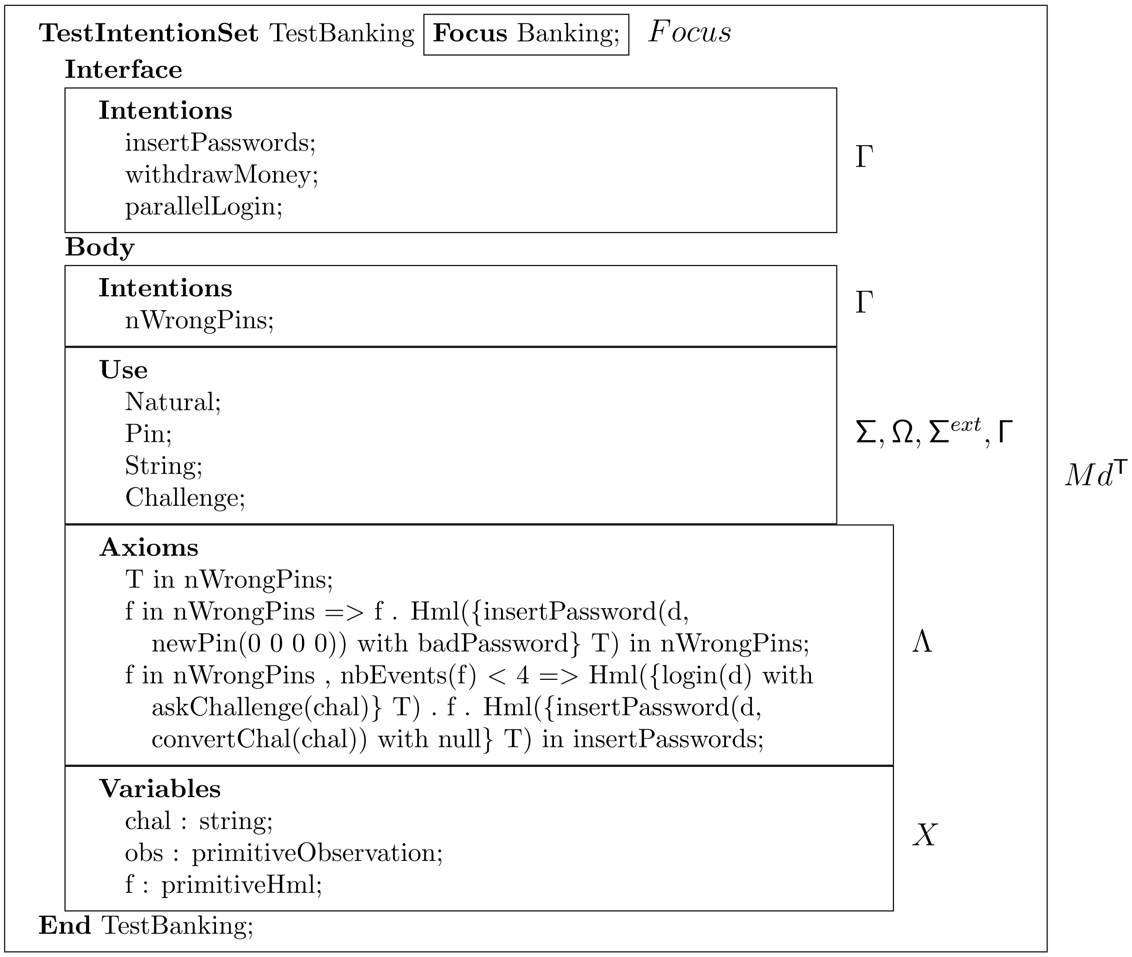
Relation Between the Abstract and Concrete Syntax of a Test Intention Module

Figure 7.3 presents a partial version of the *TestBanking* test intention module (see figure 7.2) in order to clarify the relation between the abstract and the concrete syntax of a test intention module. A particularity of the concrete syntax is that all the ADT, Class and Test Intention imported modules — represented by the $\mathsf{\Sigma}, \mathsf{\Omega}$ and $\mathsf{\Gamma}$ module interfaces in definition 7.3.22 — are collapsed in the *Use* section. Also, the context module over which the test intention module acts — represented by the test intention module interface Ξ in definition 7.3.22 — is declared following the *Focus* keyword.

The full concrete syntax of *Test Intention Modules* is defined in BNF syntax in appendix E.

7.3.4 CO-OPN and SATEL Specification

We can now extend the notion of CO-OPN specification we have presented in definition 4.4.15 to the notion of *CO-OPN and SATEL specification.* In order to do that we add to the previous definition of CO-OPN specification a set of *test intention* modules.



```
TestIntentionSet TestBanking  Focus Banking;
  Interface
    Intentions
      insertPasswords;
      withdrawMoney;
      parallelLogin;
  Body
    Intentions
      nWrongPins;
    Use
      Natural;
      Pin;
      String;
      Challenge;
    Axioms
      T in nWrongPins;
      f in nWrongPins => f . Hml({insertPassword(d,
        newPin(0 0 0 0)) with badPassword} T) in nWrongPins;
      f in nWrongPins , nbEvents(f) < 4 => Hml({login(d) with
        askChallenge(chal)} T) . f . Hml({insertPassword(d,
        convertChal(chal)) with null} T) in insertPasswords;
    Variables
      chal : string;
      obs : primitiveObservation;
      f : primitiveHml;
End TestBanking;
```

Figure 7.3: Relation Between the Abstract and Concrete Syntax of a Test Module

Definition 7.3.24 *CO-OPN and SATEL Specification*

Let Σ be a set of ADT module signatures and Ω be a set of class module interfaces such that the global signature $\Sigma_{\Sigma,\Omega}$ is complete. Let also Ξ be a set of context module signatures and Γ be a set of test intention module signatures. A CO-OPN and SATEL specification consists of a set of ADT, Class, Context and Test Intention modules such that:

$$Spec_{\Sigma,\Omega,\Xi,\Gamma} = \left\{ (Md^{\mathsf{A}}_{\Sigma,\Omega})_i \,|\, 1 \le i \le n \right\} \cup \left\{ (Md^{\mathsf{C}}_{\Sigma,\Omega})_j \,|\, 1 \le j \le m \right\} \cup \left\{ (Md^{\mathsf{X}}_{\Sigma,\Omega,\Xi})_k \,|\, 1 \le k \le o \right\} \cup \left\{ (Md^{\mathsf{T}}_{\Sigma,\Omega,\Xi,\Gamma})_l \,|\, 1 \le l \le p \right\}$$

where $Spec_{\Sigma,\Omega,\Xi}$ is a well formed *CO-OPN$_{/2c++}$ specification (see definition 4.4.15).*

The *specification* definition in 7.3.24 allows us to reuse entirely the structural and behavioral model of a CO-OPN specification, while defining a new orthogonal SATEL *verification* layer that acts over *context* modules. Also, given that test intention modules make use of each other — in the form of the Γ parameter in $Md^{\mathsf{T}}{}_{\Sigma,\Omega,\Xi,\Gamma}$ — this definition allows compositional test specifications.

7.4 Summary

In this chapter we have introduced the *test intention* language SATEL and its abstract syntax. We start by describing how to write test intentions, using SATEL's concrete syntax. SATEL is inspired by the previous work on test selection from CO-OPN specifications, but allows explicitly covering sub-behaviors of the SUT by means of *test intentions*. A *test intention* expresses a set of constraints over *variables* present in an *execution pattern* — which is a concatenation of Hml formulas including variables of testing types. A *test intention* can use other *test intentions* in its definition. It is also possible to define a *test intention* recursively.

We have introduced a number of constraints for each type of variable in SATEL, i.e. the *shape of the execution paths*, the *shape of stimulation or observation pairs* and the *shape of the parameters* of stimulations or observations. The constraints for the *shape of stimulation or observation pairs* were taken from the previous work on test generation from CO-OPN specifications, while the constraints for the remaining types were defined by our work. Another type of constraints are *uniformity* or *subUniformity* predicates which can be applied to any type of variable of SATEL.

We have also introduced a formal abstract syntax for SATEL. The abstract syntax of the language was defined in the following steps:

- we start by successively building the possible terms of the language, in the following order: *stimulation* and *observation* parameters, *stimulations and observations*, *Hml formulas with variables*, *execution patterns*. The order is necessary as the terms of each type *use* the terms of the previous type in their construction;

- we then build an accessory set of *arithmetic* and *boolean* terms which we are necessary in order to build a set of *atomic conditions* over the variables present in *execution patterns*;

- *test intention* axioms are built by using *atomic conditions*, *execution patterns* and *test intention names*;

- we finally introduce the notion of *test intention module interface* and *test* *intention* module, following the same abstract syntax style as was used for the CO-OPN$_{/2c++}$ ADT, *Class* and *Context* modules (see section 4.4);

In order to have syntactic coherence and to facilitate the building of a semantics for SATEL we have introduced a new kind of CO-OPN specification, called the *CO-OPN and SATEL* specification. A *CO-OPN and SATEL* specification consists of a *well-formed* CO-OPN$_{/2c++}$ specification extended with *test intention* modules.

Chapter 8

SATEL – Semantics

8.1 Algebraic models for a CO-OPN and SATEL Specification

Let us introduce the notion of *global SATEL signature*. This concept closely resembles the *global signature* in definition 4.4.4 given that we extend it with the new sorts induced by SATEL: *Execution patterns*, *Hml formulas*, *Stimulations*, *Observations*, *Integer* and *Boolean*. This definition will allow us to have an algebraic approach to SATEL, integrating it in this way with previous definitions for the syntax and semantics of CO-OPN.

Definition 8.1.1 *Global SATEL Signature*

Let Σ be a set of ADT module signatures, Ω be a set of class module interfaces such that the global signature $\Sigma_{\Sigma,\Omega} = \langle S, \leq, F \rangle$ is complete. Let also $\Xi = \bigcup_{1 \leq i \leq n} \Xi_i$ be a set of context module interfaces such that $\Xi_i = \langle M_i, G_i \rangle$.The global SATEL signature *over Σ, Ω and Ξ (noted $\Sigma^{Sat}_{\Sigma,\Omega,\Xi}$) is the triplet $\langle S', \leq', F' \rangle$ where:*

- $S' = S \cup \bigcup_{1 \leq i \leq n} Pat_{\Xi_i} \cup \bigcup_{1 \leq i \leq n} Hml_{\Xi_i} \cup \bigcup_{1 \leq i \leq n} Stim_{\Xi_i} \cup \bigcup_{1 \leq i \leq n} Obsrv_{\Xi_i} \cup \{Num, Bool\}$
- $\leq' = \leq \bigcup_{1 \leq i \leq n} (Pat_{\Xi_i}, Pat_{\Xi_i}) \cup \bigcup_{1 \leq i \leq n} (Hml_{\Xi_i}, Hml_{\Xi_i}) \cup \bigcup_{1 \leq i \leq n} (Stim_{\Xi_i}, Stim_{\Xi_i}) \cup \bigcup_{1 \leq i \leq n} (Obsrv_{\Xi_i}, Obsrv_{\Xi_i}) \cup \{(Num, Num), (Bool, Bool)\}$
- $F' = F \cup \bigcup_{1 \leq i \leq n} F_{Pat_{\Xi_i}} \cup \bigcup_{1 \leq i \leq n} F_{Hml_{\Xi_i}} \cup \bigcup_{1 \leq i \leq n} F_{Stim_{\Xi_i}} \cup \bigcup_{1 \leq i \leq n} F_{Obsrv_{\Xi_i}} \cup F_{Num} \cup F_{Bool}$

where:

- $F_{Pat_{\Xi_i}} = \big\{\_ : Hml_{\Xi_i} \rightarrow Pat_{\Xi_i},\ . : Hml_{\Xi_i}, Pat_{\Xi_i} \rightarrow Pat_{\Xi_i}\big\}$
- $F_{Hml_{\Xi_i}} = \big\{T :\rightarrow Hml_{\Xi_i},\ <\_>\_ : Stim_{\Xi_i}, Hml_{\Xi_i} \rightarrow Hml_{\Xi_i},$
  $<\_,\_>\_ : Stim_{\Xi_i}, Obsrv_{\Xi_i}, Hml_{\Xi_i} \rightarrow Hml_{\Xi_i},$
  $not : Hml_{\Xi_i} \rightarrow Hml_{\Xi_i},\ and : Hml_{\Xi_i}, Hml_{\Xi_i} \rightarrow Hml_{\Xi_i}\big\}$
- $F_{Stim_{\Xi_i}} = \big\{m_{s_1,\ldots,s_n}(s_1,\ldots,s_n) :\rightarrow Stim_{\Xi_i},$
  $// : Stim_{\Xi_i}, Stim_{\Xi_i} \rightarrow Stim_{\Xi_i} \,|\, m_{s_1,\ldots,s_n} \in M_i \wedge s_1 \ldots s_n \in S\big\}$
- $F_{Obsrv_{\Xi_i}} = \big\{g_{s_1,\ldots,s_n}(s_1,\ldots,s_n) :\rightarrow Obsrv_{\Xi_i},$
  $// : Obsrv_{\Xi_i}, Obsrv_{\Xi_i} \rightarrow Obsrv_{\Xi_i},\ .. : Obsrv_{\Xi_i}, Obsrv_{\Xi_i} \rightarrow Obsrv_{\Xi_i},$
  $\oplus : Obsrv_{\Xi_i}, Obsrv_{\Xi_i} \rightarrow Obsrv_{\Xi_i} \,|\, g_{s_1,\ldots,s_n} \in G_i \wedge s_1 \ldots s_n \in S\big\}$
- $F_{Num} = \big\{0 :\rightarrow Num,\ succ : Num \rightarrow Num,$
  $+ : Num, Num \rightarrow Num,\ - : Num, Num \rightarrow Num,$
  $* : Num, Num \rightarrow Num,\ / : Num, Num \rightarrow Num,$
  $nbEvents : Hml_{\Xi_j} \rightarrow Num,\ depth : Hml_{\Xi_j} \rightarrow Num,$
  $nbOccur : \big(M_j, Hml_{\Xi_j}\big) \rightarrow Num,$
  $nbSynchro : Stim_{\Xi_j} \rightarrow Num \,|\, 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$
- $F_{Bool} = \big\{\{true, false\} :\rightarrow Bool,\ not : Bool \rightarrow Bool,$
  $and : Bool, Bool \rightarrow Bool,\ or : Bool, Bool \rightarrow Bool$
  $sequence : Hml_{\Xi_j} \rightarrow Bool,\ positive : Hml_{\Xi_j} \rightarrow Bool,$
  $trace : Hml_{\Xi_j} \rightarrow Bool \,|\, 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$

Notice that in definition 8.1.1 we only provide the signatures for the sorts induced by SATEL and we do not establish equations. The following sections describe particular algebras for the sorts introduced in definition 8.1.1 of a *global SATEL signature*. These algebras provide the semantics of the several functions we use to constrain SATEL variables. We build the algebras separately for each family of terms, namely *Execution Patterns*, *Hml*, *Stimulations*, *Observations*, *Arithmetic* and *Boolean*.

8.1.1 Algebras for Stimulation, Observation and Hml terms

Definition 8.1.2 *Algebras for Stimulation, Observation and Hml terms*

Let $\Sigma = \langle S, \leq, F\rangle$ be a complete global signature *and Ξ a context module interface such that $\Xi = \langle M, G\rangle$. Let also A be a Σ-algebra.*

Let us define $Sig_{Stim}(\Sigma, \Xi) = \langle S', \leq', F'\rangle$ as the signature obtained by extending Σ as follows:

- $S' = S \cup \{Stim_{\Xi}\}$

- $\leq' = \leq \cup \{(Stim_\Xi, Stim_\Xi)\}$
- $F' = F \cup \big\{m_{s_1,\ldots,s_n}(s_1,\ldots,s_n) :\rightarrow Stim_\Xi,$
  $// : Stim_\Xi, Stim_\Xi \rightarrow Stim_\Xi \,|\, m_{s_1,\ldots,s_n} \in M \wedge s_1 \ldots s_n \in S\big\}$

Let $\Sigma' = Sig_{Stim}(\Sigma, \Xi)$*. The model for* Σ' *corresponds to the* $Term_{\Sigma',Stim_{\Xi_i}}(A)$ *algebra and is noted* $Sem_A(Sig_{Stim}(\Sigma, \Xi))$*.*

We define the $Sig_{Obsrv}(\Sigma, \Xi)$ *signature and its respective model* $Sem_A(Sig_{Obsrv}(\Sigma, \Xi))$ *analogously.*

Let us define $\Sigma''' = Sig_{Hml}(\Sigma, \Xi) = \langle S''', \leq''', F''' \rangle$ *as the signature obtained by extending* $\Sigma' = Sig_{Stim}(\Sigma, \Xi) = \langle S', \leq', F' \rangle$ *and* $\Sigma'' = Sig_{Obsrv}(\Sigma, \Xi) = \langle S'', \leq'', F'' \rangle$ *as follows:*

- $S''' = S' \cup S'' \cup \{Hml_\Xi\}$
- $\leq''' = \leq' \cup \leq'' \cup \{(Hml_\Xi, Hml_\Xi)\}$
- $F''' = F' \cup F'' \cup \big\{T :\rightarrow Hml_\Xi,\ <\_>\_ : Stim_\Xi, Hml_\Xi \rightarrow Hml_\Xi,$
  $<\_,\_>\_ : Stim_\Xi, Obsrv_\Xi, Hml_\Xi \rightarrow Hml_\Xi,$
  $not : Hml_\Xi \rightarrow Hml_\Xi,\ and : Hml_\Xi, Hml_\Xi \rightarrow Hml_\Xi\big\}$

The model for the Σ''' *signature is the* $Term_{\Sigma''',Hml_\Xi}\big(Sem_A(\Sigma') \cup Sem_A(\Sigma'')\big)$ *algebra — noted* $Sem_A(Sig_{Hml}(\Sigma, \Xi))$*.*

8.1.2 Algebra for Execution Pattern terms

Definition 8.1.3 *Algebra for Execution Pattern terms*

Let Σ *be a* complete global signature *and* Ξ *a context module interface such that* $\Xi = \langle M, G \rangle$*. Let also* A *be a* Σ*-algebra.*

Let us define $\Sigma' = Sig_{Pat}(\Sigma, \Xi) = \langle S', \leq', F' \rangle$ *as the signature obtained by extending* $\Sigma'' = Sig_{Hml}(\Sigma, \Xi) = \langle S, \leq, F \rangle$ *(see definition 8.1.2) as follows:*

- $S' = S \cup \{Pat_\Xi\}$
- $\leq' = \leq \cup \{(Pat_\Xi, Pat_\Xi)\}$
- $F' = F \cup \big\{\_ : Hml_\Xi \rightarrow Pat_\Xi,\ . : Hml_\Xi, Pat_\Xi \rightarrow Pat_\Xi\big\}$

Let $(A', F^{A'}) = Sem_A(\Sigma'')$. *The algebra for* Σ' *is the* S'*-sorted set* $A' \cup A_{Pat_\Xi}$ *where* $A_{Pat_\Xi} = A'_{Hml_\Xi}$ *and the family of functions* $F^{A'} \cup \{ \_^{A_{Pat_\Xi}}_{Hml_\Xi, Pat_\Xi}, \ \cdot^{A_{Pat_\Xi}}_{Hml_\Xi\, Pat_\Xi, Pat_\Xi} \}$ *where:*

Definition 8.1.4 $\_^{A_{Pat_\Xi}}_{Hml_\Xi, Pat_\Xi} : A'_{Hml_\Xi} \to A_{Pat_\Xi}$

- $\_^{A_{Pat_\Xi}}\, T^{A_{Hml_\Xi}} = T^{A_{Pat_\Xi}}$
- $\_^{A_{Pat_\Xi}}\, (\neg f) = \neg(\_^{A_{Pat_\Xi}}\, f)$
- $\_^{A_{Pat_\Xi}}\, (f \wedge g) = (\_^{A_{Pat_\Xi}}\, f) \wedge (\_^{A_{Pat_\Xi}}\, g)$
- $\_^{A_{Pat_\Xi}}\, (\langle stm, obs \rangle f) = \langle stm, obs \rangle (\_^{A_{Pat_\Xi}}\, f)$

for all $f, g \in A'_{Hml_\Xi}$, $stm \in A'_{Stim_\Xi}$, $obs \in A'_{Obsrv_\Xi}$

Definition 8.1.5 $\cdot^{A_{Pat_\Xi}}_{Hml_\Xi\, Pat_\Xi, Pat_\Xi} : A'_{Hml_\Xi} \times A_{Pat_\Xi} \to A_{Pat_\Xi}$

- $T^{A_{Pat_\Xi}} \cdot^{A_{Pat_\Xi}} p = p$
- $(\neg f) \cdot^{A_{Pat_\Xi}} p = \neg(f \cdot^{A_{Pat_\Xi}} p)$
- $(f \wedge g) \cdot^{A_{Pat_\Xi}} p = (f \cdot^{A_{Pat_\Xi}} p) \wedge (g \cdot^{A_{Pat_\Xi}} p)$
- $(\langle stm, obs \rangle f) \cdot^{A_{Pat_\Xi}} p = \langle stm, obs \rangle (f \cdot^{A_{Pat_\Xi}} p)$

for all $f, g \in A'_{Hml_\Xi}$, $p \in A_{Pat_\Xi}$, $stm \in A'_{Stim_\Xi}$, $obs \in A'_{Obsrv_\Xi}$

8.1.3 Algebra for Arithmetic terms

When we defined the arithmetic terms for SATEL in section 7.3.2, we have also implicitly defined a signature $\Sigma_{Num} = \langle \{Num\}, \leq_{Num}, F_{Num} \rangle$, which we have included in definition 8.1.1 of the signature induced by a test intention module. Let us now define a model for Σ_{Num}. This model is in fact the typical integer algebra, which consists of an A_{Num} carrier set (which is in fact the well-known set $\mathbb{Z}$), the typical arithmetic operations $\{+^{Num}, -^{Num}, *^{Num}, /^{Num}\}$ and a set of functions we define in the context of SATEL.

Definition 8.1.6 *Algebra for arithmetic terms*

Let Σ *be a* complete global signature *and* $\Xi = \bigcup_{1 \leq i \leq n} \Xi_i$ *be a set of context module interfaces such that* $\Xi_i = \langle M_i, G_i \rangle$. *Let also* A *be a* Σ*-algebra.*

Let us define $Sig_{Num}(\Sigma, \Xi_i) = \langle S', \leq', F' \rangle$ *as the signature obtained by extending the signatures* $Sig_{Pat}(\Sigma, \Xi_i) = \langle S_{\Xi_i}, \leq_{\Xi_i}, F_{\Xi_i} \rangle$ $(1 \leq i \leq n)$ *(see definition 8.1.3) as follows:*

- $S' = \bigcup_{\Xi_i} S_{\Xi_i} \cup \{Num\}$
- $\leq' = \bigcup_{\Xi_i} \leq_{\Xi_i} \cup \{(Num, Num)\}$
- $F' = \big\{0 :\rightarrow Num,\ succ : Num \rightarrow Num,$
  $+ : Num, Num \rightarrow Num,\ - : Num, Num \rightarrow Num,$
  $* : Num, Num \rightarrow Num,\ / : Num, Num \rightarrow Num,$
  $nbEvents : Hml_{\Xi_j} \rightarrow Num,\ depth : Hml_{\Xi_j} \rightarrow Num,$
  $nbOccur : \big(M_j, Hml_{\Xi_j}\big) \rightarrow Num,$
  $nbSynchro : Stim_{\Xi_j} \rightarrow Num \,|\, 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$

Let $(A'_{\Xi_i}, F^{A'}{}_{\Xi_i}) = Sem_A(\Sigma_{Pat_{\Xi_i}})$ *be the model of the* $\Sigma_{Pat_{\Xi_i}}$ *signature. The algebra for* Σ_{Num} *is the* S'*-sorted set* $\bigcup_{\Xi_i} A'_{\Xi_i} \cup A_{Num}$ *and the family of functions* $F^{A'} \cup \big\{0^{A_{Num}}_{\epsilon, Num},\ succ^{A_{Num}}_{Num, Num},\ nbEvents^{A_{Num}}_{Hml_{\Xi_j}, Num},\ depth^{A_{Num}}_{Hml_{\Xi_j}, Num},\ nbOccur^{A_{Num}}_{M_j\, Hml_{\Xi_j}, Num},$ $nbSynchro^{A_{Num}}_{Stim_{\Xi_j}, Num} \,|\, 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$[1] *where the functions having* Num *as their co-domain's sort are defined as follows:*

Definition 8.1.7 $nbEvents^{A_{Num}} : A'_{Hml_{\Xi_i}} \rightarrow A_{Num}$

- $nbEvents^{A_{Num}}(T^{A_{Hml_{\Xi_i}}}) = 0^{A_{Num}}$
- $nbEvents^{A_{Num}}(\neg f) = nbEvents^{A_{Num}}(f)$
- $nbEvents^{A_{Num}}(f \wedge g) = nbEvents^{A_{Num}}(f) +^{A_{Num}} nbEvents(g)$
- $nbEvents^{A_{Num}}(\langle stm, obs \rangle f) = nbSynchro^{A_{Num}}(stm) +^{A_{Num}} nbEvents^{A_{Num}}(f)$

for all $stm \in A'_{Stim_{\Xi_i}}$, $obs \in A'_{Obsrv_{\Xi_i}}$, $f, g \in A'_{Hml_{\Xi_i}}$ $(1 \leq i \leq n)$

The $nbEvents^{Num}$ *function returns the total number of events present in an execution pattern.*

Definition 8.1.8 $depth^{Num} : A'_{Hml_{\Xi_i}} \rightarrow A_{Num}$

[1] Although we do not describe the functions associated to the $\{+, -, *, /\}$ functors, they are implicitly present in the set of functions of the algebra for arithmetic terms.

- $depth^{A_{Num}}(T^{A_{Hml_{\Xi_i}}}) = 0^{A_{Num}}$
- $depth^{A_{Num}}(\neg f) = depth^{A_{Num}}(f)$
- $depth^{A_{Num}}(f \wedge g) = max\big(depth^{A_{Num}}(f), depth^{A_{Num}}(g)\big)$
- $depth^{A_{Num}}(\langle stm, obs\rangle f) = 1^{A_{Num}} +^{A_{Num}} depth^{A_{Num}}(f)$

where:

- $stm \in A'_{Stim_{\Xi_i}}$ *and* $obs \in A'_{Obsrv_{\Xi_i}}$, $f, g \in A'_{Hml_{\Xi_i}}$ $(1 \leq i \leq n)$
- $max : A_{Num} \times A_{Num} \rightarrow A_{Num}$ *is a function returning the largest of two integer numbers.*

The $depth^{A_{Num}}$ *function returns the length of the longest branch of an execution pattern.*

Definition 8.1.9 $nbOccur^{A_{Num}} : \big(M_i \times A'_{Hml_{\Xi_i}}\big) \rightarrow A_{Num}$

- $nbOccur^{A_{Num}}(m_{s_1 \ldots s_n}, T^{A_{Hml_{\Xi_i}}}) = 0^{A_{Num}}$
- $nbOccur^{A_{Num}}(m_{s_1 \ldots s_n}, \neg f) = nbOccur^{A_{Num}}(f)$
- $nbOccur^{A_{Num}}(m_{s_1 \ldots s_n}, f \wedge g) = nbOccur^{A_{Num}}(f) + nbOccur^{A_{Num}}(g)$
- $nbOccur^{A_{Num}}(m_{s_1 \ldots s_n}, \langle stm, obs\rangle f) = 1^{A_{Num}} +^{A_{Num}} nbOccur^{A_{Num}}(f)$
  if stm is based on $m_{s_1 \ldots s_n}$
- $nbOccur^{A_{Num}}(m_{s_1 \ldots s_n}, \langle stm, obs\rangle f) = nbOccur^{A_{Num}}(f)$
  if stm is ***not*** *based on* $m_{s_1 \ldots s_n}$

where $stm \in A'_{Stim_{\Xi_i}}$, $obs \in A'_{Obsrv_{\Xi_i}}$ *and* $m_{s_1 \ldots s_n} \in M_i$, $f, g \in A'_{Hml_{\Xi_i}}$ $(1 \leq i \leq n)$

The $nbOccur^{A_{Num}}$ *function measures how many times a given method shows up in an execution pattern.*

Definition 8.1.10 $nbSynchro^{A_{Num}} : A'_{Stim_{\Xi_i}} \rightarrow A_{Num}$

- $nbSynchro(stm) = 1^{A_{Num}}$ *for all* $stm = m_{s_1 \ldots s_n}(\ldots)$
- $nbSynchro(stm \,//\, stm') = nbSynchro(stm) +^{A_{Num}} nbSynchro(stm')$

where $stm, stm' \in A'_{Stim_{\Xi_i}}$ $(1 \leq i \leq n)$

The $nbSynchro^{A_{Num}}$ *function measures the number of simultaneous synchronizations in a stimulation term.*

8.1.4 Algebra for Boolean terms

As happens for arithmetic terms, we have also implicitly defined in section 7.3.2 a signature $\Sigma_{Bool} = \langle\{Bool\}, \leq_{Bool}, F_{Bool}\rangle$ for boolean terms for a given test intention module with its focus on a context module Ξ. Also as for arithmetic terms, this signature is included in definition 8.1.1 of the signature induced by a test intention module. As Σ_{Bool}-algebra we will use the well known boolean algebra which we will call A_{Bool}. A_{Bool} includes the carrier set $\{true^{Bool}, false^{Bool}\}$, the typical boolean functions $\{and^{Bool}, or^{Bool}, not^{Bool}\}$ and a set of functions we define in the context of SATEL.

Definition 8.1.11 *Algebra for boolean terms*

Let Σ *be a* complete global signature *and* $\Xi = \bigcup_{1\leq i\leq n} \Xi_i$ *be a set of context module interfaces such that* $\Xi_i = \langle M_i, G_i\rangle$*. Let also* A *be a* Σ*-algebra.*

Let us define $Sig_{Bool}(\Sigma, \Xi_i) = \langle S', \leq', F'\rangle$ *as the signature obtained by extending the signatures* $Sig_{Pat}(\Sigma, \Xi_i) = \langle S_{\Xi_i}, \leq_{\Xi_i}, F_{\Xi_i}\rangle$ $(1 \leq i \leq n)$ *(see definition 8.1.3) as follows:*

- $S' = \bigcup_{\Xi_i} S_{\Xi_i} \cup \{Bool\}$
- $\leq' = \bigcup_{\Xi_i} \leq_{\Xi_i} \cup \{(Bool, Bool)\}$
- $F' = \bigcup_{\Xi_i} F_{\Xi_i} \cup \big\{\{true, false\} :\rightarrow Bool,\ not : Bool \rightarrow Bool,$
  $and : Bool, Bool \rightarrow Bool,\ or : Bool, Bool \rightarrow Bool$
  $sequence : Hml_{\Xi_j} \rightarrow Bool,\ positive : Hml_{\Xi_j} \rightarrow Bool,$
  $trace : Hml_{\Xi_j} \rightarrow Bool \mid 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$

Let $(A'_{\Xi_i}, F^{A'}{}_{\Xi_i}) = Sem_A(\Sigma_{Pat_{\Xi_i}})$ be the model of the $\Sigma_{Pat_{\Xi_i}}$ signature. The algebra for Σ_{Num} is the S'-sorted set $\bigcup_{\Xi_i} A'_{\Xi_i} \cup A_{Num}$ and the family of functions $F^{A'} \cup$ $\big\{true^{A_{Bool}}_{\epsilon,Bool},\ false^{A_{Bool}}_{\epsilon,Bool},\ sequence^{A_{Bool}}_{Hml_{\Xi_j},Bool},\ positive^{A_{Bool}}_{Hml_{\Xi_j},Bool},\ nbOccur^{A_{Num}}_{M_j\, Hml_{\Xi_j},Num},$ $trace^{A_{Num}}_{Hml_{\Xi_j},Num} \mid 1 \leq j \leq n \wedge \Xi_j \in \Xi\big\}$[2] where where the functions having $Bool$ as its co-domain's sort are defined as follows:

Definition 8.1.12 $sequence^{A_{Bool}} : A'_{Hml_{\Xi_i}} \rightarrow A_{Bool}$

- $sequence^{A_{Bool}}(T^{A_{Hml_{\Xi_i}}}) = true^{A_{Bool}}$

[2]As for the arithmetic terms, although we do not describe the functions associated to the $\{not, and, or\}$ functors they are implicitly present in the set of functions of the algebra for boolean terms.

- $sequence^{A_{Bool}}(\neg f) = sequence^{A_{Bool}}(f)$
- $sequence^{A_{Bool}}(f \wedge g) = false^{A_{Bool}}$
- $sequence^{A_{Bool}}(\langle stm, obs\rangle f) = sequence^{A_{Bool}}(f)$

where $stm \in A'_{Stim_{\Xi_i}}$, $obs \in A'_{Obsrv_{\Xi_i}}$, $f, g \in A'_{Hml_{\Xi_i}}$ $(1 \leq i \leq n)$

The $sequence^{Bool}$ function returns the $true^{Bool}$ value when an execution pattern does not include the *and* Hml operator.

Definition 8.1.13 $positive^{A_{Bool}} : A'_{Hml_{\Xi_i}} \rightarrow A_{Bool}$

- $positive^{A_{Bool}}(T^{A_{Hml_{\Xi_i}}}) = true^{A_{Bool}}$
- $positive^{A_{Bool}}(\neg f) = false^{A_{Bool}}$
- $positive^{A_{Bool}}(f \wedge g) = positive^{A_{Bool}}(f)\ and^{A_{Bool}}\ positive^{A_{Bool}}(g)$
- $positive^{A_{Bool}}(\langle stm, obs\rangle f) = positive^{A_{Bool}}(f)$

where $stm \in A'_{Stim_{\Xi_i}}$, $obs \in A'_{Obsrv_{\Xi_i}}$, $f, g \in A'_{Hml_{\Xi_i}}$ $(1 \leq i \leq n)$

The $positive^{A_{Bool}}$ function returns the $true^{A_{Bool}}$ value when an execution pattern does not include the *not* Hml operator.

Definition 8.1.14 $trace^{A_{Bool}} : A'_{Hml_{\Xi_i}} \rightarrow A_{Bool}$ *is defined as follows:*

$trace^{A_{Bool}}(f) = positive^{A_{Bool}}(f)\ and^{A_{Bool}}\ sequence^{A_{Bool}}(f)$ *where* $f \in A'_{Hml_{\Xi_i}} (1 \leq i \leq n)$

8.1.5 Algebra for the global SATEL signature of a CO-OPN Specification (SATEL Algebra)

Definition 8.1.15 *SATEL Algebra*

Let Σ *be a* complete global signature *and* $\Xi = \bigcup_{1 \leq i \leq n} \Xi_i$ *be a set of context module interfaces such that* $\Xi_i = \langle M_i, G_i\rangle$. *Let also* A *be a* Σ*-algebra. The model of the* $\Sigma^{Sat}_{\Sigma,\Omega,\Xi}$ *global SATEL signature is the algebra* $(A', F^{A'})$ *which we define as follows:*

$$(A', F^{A'}) = (A^{Num} \cup A^{Bool}, F^{A_{Num}} \cup F^{A_{Bool}})$$

where $(A^{Num}, F^{A_{Num}}) = Sem_A(Sig_{Num}(\Sigma, \Xi))$ *and* $(A^{Bool}, F^{A_{Bool}}) = Sem_A(Sig_{Bool}(\Sigma, \Xi))$.

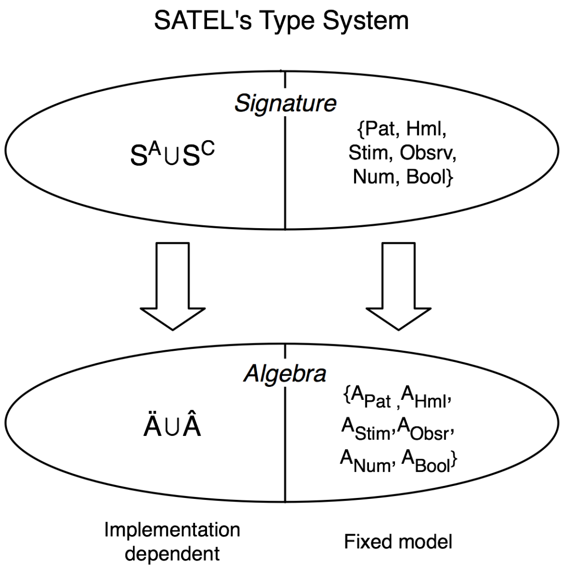


Figure 8.1: SATEL's Type System

Notice that in definition 8.1.15 the algebras composing the model are fixed for the types Hml_{Ξ_i} Hml_{Ξ_i}, $Stim_{\Xi_i}$, $Obsrv_{\Xi_i}$, Num and $Bool$ and not fixed for any type $s \in S$. This is so because the types $s \in S$ are type specifications of CO-OPN requiring a real implementation — which can be done in a different amount of ways — while the remaining types are particular to SATEL. Figure 8.1 depicts SATEL's type system which consists of the types induced by the ADT and Class modules (on the left of the figure) and the types induced by a *test intention* module (on the right of the figure). Notice that the $\{Pat_{\Xi_i}, Hml_{\Xi_i}, Stim_{\Xi_i}, Obsrv_{\Xi_i}\}$ types are particular to a given Ξ_i *context module* and their terms describe *execution patterns*, *stimulations* and *observations* of the SUT modeled by that *context module*.

8.2 Substitutions

Definition 8.2.1 *SATEL Substitution*

Let $\Sigma^{Sat} = \langle S, \leq F\rangle$ be a global CO-OPN and SATEL signature, $M = (M_w)_{w\in S^*}$ an S^*-sorted set of method names of $\mathbf{M}$*, $G = (G_w)_{w\in S^*}$ an S^*-sorted set of gate names of* $\mathbf{G}$*, $(X_s)_{s\in S}$ an S-sorted set of variables and $I \in \mathbf{I}$ a set of test intention names. A SATEL substitution $/^{Sat}$ replaces values in the conditions $Cond_{\Sigma^{Sat},M,G,X,I}$ and is defined as follows:*

- $(tn\ op_{Num}\ tn')[v/^{Sat}x] = (tn[v/x]\ op_{Num}\ tn'[v/x])$

- $(tb\ bop_{Bool}\ tb')[v/^{Sat}x] = (tb[v/x]\ bop_{Bool}\ tb'[v/x])$
- $(tb\ uop_{Bool}\ tb')[v/^{Sat}x] = (uop_{Bool}\ tb[v/x])$
- $(tp = tp')[v/^{Sat}x] = (tp[v/x] = tp'[v/x])$
- $(ts = ts')[v/^{Sat}x] = (ts[v/x] = ts'[v/x])$
- $(to = to')[v/^{Sat}x] = (to[v/x] = to'[v/x])$
- $(tn = tn')[v/^{Sat}x] = (tn[v/x] = tn'[v/x])$
- $(tb = tb')[v/^{Sat}x] = (tb[v/x] = tb'[v/x])$
- $(t = t')[v/^{Sat}x] = (t[v/x] = t'[v/x])$

We extend the SATEL substitution $/^{Sat}$ to sets of conditions $Cond_{\Sigma^{Sat},M,G,X,I}$ as follows:

- $\emptyset[v/^{Sat}x] = \emptyset$
- $(cond \cup \{c\})[v/^{Sat}x] = cond[v/^{Sat}x]\ \cup\ \{c[v/^{Sat}x]\}$

where:

- $op_{Num} \in \{==, <>, <, >, <=, >=\}$, $bop_{Bool} \in \{and, or\}$ *and* $uop_{Bool} \in \{not\}$
- $tn, tn' \in A_{Num}$, $tb, tb' \in A_{Bool}$, $tp, tp' \in A_{Pat_{\Xi_i}}$, $ts, ts' \in A_{Stim_{\Xi_i}}$, $to, to' \in A_{Obsrv_{\Xi_i}}$ *and* $t, t' \in A_s$ $(1 \leq i \leq n)$
- $cond \in \mathcal{P}(Cond_{\Sigma^{Sat},M,G,X,I})$ *and* $c \in Cond_{\Sigma^{Sat},M,G,X,I}$

Example 8.2.2 *Consider $\{sequence(f), nbEvents(f)\ <=\ 3\}$ which is the set of conditions for the test intention axiom defined in line 26 of figure 7.2. The SATEL substitution:*

$$\{sequence(f), nbEvents(f) <= 3\}\ \left[\langle login(d)\rangle T/^{Sat}f\right]$$

replaces all instances of f in the set of conditions by the Hml formula '$\langle login(d)\rangle T$', producing the new set of conditions:

$$\{sequence(\langle login(d)\rangle T), nbEvents(\langle login(d)\rangle T) <= 3\}$$

Definition 8.2.3 *Satisfaction of Variable Constraint Conditions*

Let $\Sigma^{Sat} = \langle S, \leq F\rangle$ be a global CO-OPN and SATEL signature, $M = (M_w)_{w\in S^*}$ an S^*-sorted set of method names of M*, $G = (G_w)_{w\in S^*}$ an S^*-sorted set of gate names of* G*, $(X_s)_{s\in S}$ an S-sorted set of variables and $I \in$* I *a set of test intention names. Let also A be a finitely generated[3] Σ^{Sat}-SATEL algebra. Given an S-sorted substitution θ, the satisfaction relation $\models_{\mathsf{c}} \subseteq A \times Subs_S(X) \times Cond_{\Sigma^{Sat},M,G,X,I}$ is given by:*

- $A, \theta \models_{\mathsf{c}} tn\, op_{Num}\, tn' \Leftrightarrow \left(\llbracket\theta^{\#}(tn)\rrbracket_{Num}\; \mathbb{I}_{op_{Num}}\; \llbracket\theta^{\#}(tn')\rrbracket_{Num}\right) = true^{Bool}$
- $A, \theta \models_{\mathsf{c}} tb\, bop_{Bool}\, tb' \Leftrightarrow \left(\llbracket\theta^{\#}(tb)\rrbracket_{Bool}\; \mathbb{I}_{bop_{Bool}}\; \llbracket\theta^{\#}(tb')\rrbracket_{Bool}\right) = true^{Bool}$
- $A, \theta \models_{\mathsf{c}} uop_{Bool}\, tb \Leftrightarrow \left(\mathbb{I}_{uop_{Bool}}\; \llbracket\theta^{\#}(tb')\rrbracket_{Bool}\right) = true^{Bool}$
- $A, \theta \models_{\mathsf{c}} tp = tp' \Leftrightarrow \llbracket\theta^{\#}(tp)\rrbracket_{Pat_{\Xi_i}} = \llbracket\theta^{\#}(tp')\rrbracket_{Pat_{\Xi_i}}$
- $A, \theta \models_{\mathsf{c}} ts = ts' \Leftrightarrow \llbracket\theta^{\#}(ts)\rrbracket_{Stim_{\Xi_i}} = \llbracket\theta^{\#}(ts')\rrbracket_{Stim_{\Xi_i}}$
- $A, \theta \models_{\mathsf{c}} to = to' \Leftrightarrow \llbracket\theta^{\#}(to)\rrbracket_{Obsrv_{\Xi_i}} = \llbracket\theta^{\#}(to')\rrbracket_{Obsrv_{\Xi_i}}$
- $A, \theta \models_{\mathsf{c}} tn = tn' \Leftrightarrow \llbracket\theta^{\#}(tn)\rrbracket_{Num} = \llbracket\theta^{\#}(tn')\rrbracket_{Num}$
- $A, \theta \models_{\mathsf{c}} tb = tb' \Leftrightarrow \llbracket\theta^{\#}(tb)\rrbracket_{Bool} = \llbracket\theta^{\#}(tb')\rrbracket_{Bool}$
- $A, \theta \models_{\mathsf{c}} t = t' \Leftrightarrow \llbracket\theta^{\#}(t)\rrbracket_{s} = \llbracket\theta^{\#}(t')\rrbracket_{s}$
- $A, \theta \models_{\mathsf{c}} unif(t)$ *for all* $t \in (T_{\Sigma^{Sat},\emptyset})_{s\in S}$
- $A, \theta \models_{\mathsf{c}} subUnif(t)$ *for all* $t \in (T_{\Sigma^{Sat},\emptyset})_{s\in S}$

We extend the satisfaction relation $\models_{\mathsf{c}}$ to sets of Variable Constraint Conditions *($\models_{\mathsf{c}} \subseteq A \times Subs_S(X) \times \mathcal{P}(Cond_{\Sigma^{Sat},M,G,X,I})$) in the following way:*

$$A, \theta \models_{\mathsf{c}} cond \;\Leftrightarrow\; \forall c \in cond \;.\; A, \theta \models_{\mathsf{c}} c$$

where:

- $op_{Num} \in \{==, <>, <, >, <=, >=\}$*, $bop_{Bool} \in \{and, or\}$ and $uop_{Bool} \in \{not\}$*
- *$\mathbb{I}_{op_{Num}}$, $\mathbb{I}_{bop_{Bool}}$ and $\mathbb{I}_{uop_{Bool}}$ correspond to the interpretation of predicate names into their algebraic counterparts. For example $\mathbb{I}_{==}$ corresponds to the equality predicate in $A_{Num} \in A_{\Xi}$ (integer algebra), while $\mathbb{I}_{and}$ corresponds to the conjunction predicate in $A_{Bool} \in A_{\Xi}$ (boolean algebra)*

[3]We require the algebra to be finitely generated in order to guarantee that we can produce tests for all values in the implementation.

- $tn, tn' \in A_{Num},\ tb, tb' \in A_{Bool},\ tp, tp' \in A_{Hml_{\Xi_i}},\ ts, ts' \in A_{Stim_{\Xi_i}},\ to, to' \in A_{Obsrv_{\Xi_i}}$ *and* $t, t' \in A_s$ $(1 \leq i \leq n)$
- $cond \in \mathcal{P}(Cond_{\Sigma^{Sat},M,G,X,I})$ *and* $c \in Cond_{\Sigma^{Sat},M,G,X,I}$

8.3 Exhaustive Test Set for a Test Intention

Let us now provide the inference rules that allow calculating the unfolding of *test intentions*. These rules are inspired from the well known *cut rule* in the general resolution. Note that in the text that follows in the present chapter we will use the $Var(pat)$ notation to denote the set of variables present in an *execution pattern*.

Definition 8.3.1 *Full Expanded Execution Pattern Set, Expanded Execution Pattern Set*

Let Spec be a CO-OPN and SATEL test specification having a SATEL global signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$. *Let also* $Md^{\mathsf{T}} = \langle \Gamma, Focus, \Lambda, X\rangle \in Spec$ *be a* test intention module, $i \in \Gamma$ *be a test intention and* $Focus = \langle M, G\rangle$. *The full expanded axiom set* $AllPat_{Spec}(i) \subseteq Cond_{\Sigma^{Sat},M,G,X,\Gamma} \times (T_{\Sigma^{Sat},X})_{Pat_{Focus}}$ *is the least set satisfying the following rules:*

$$Local\ \frac{\langle cond, pat, i\rangle \in \Lambda}{\langle cond, pat\rangle \in AllPat_{Spec}(i)}$$

$$RemUn\ \frac{\begin{array}{l}\langle cond \cup \{t\,in\,i', unif(t)\}, pat\rangle \in AllPat_{Spec}(i),\\ \langle cond', pat'\rangle \in AllPat_{Spec}(i')\end{array}}{\langle cond[pat'/^{Sat}t] \cup cond' \cup \left(\bigcup_{x \in Var(pat')} unif(x)\right),\ pat[pat'/t]\rangle \in AllPat_{Spec}(i)}$$

$$RemSubUn\ \frac{\begin{array}{l}\langle cond \cup \{t\,in\,i', subunif(t)\}, pat\rangle \in AllPat_{Spec}(i),\\ \langle cond', pat'\rangle \in AllPat_{Spec}(i')\end{array}}{\langle cond[pat'/^{Sat}t] \cup cond' \cup \left(\bigcup_{x \in Var(pat')} subunif(x)\right), pat[pat'/t]\rangle \in AllPat_{Spec}(i)}$$

$$Include\ \frac{\begin{array}{l}\langle cond \cup \{t\,in\,i'\}, pat\rangle \in AllPat_{Spec}(i), unif(t) \notin cond,\\ subunif(t) \notin cond, \langle cond', pat'\rangle \in AllPat_{Spec}(i')\end{array}}{\langle cond[pat'/^{Sat}t] \cup cond', pat[pat'/t]\rangle \in AllPat_{Spec}(i)}$$

where $i, i' \in \Gamma$, $cond, cond' \in \mathcal{P}(Cond_{\Sigma^{Sat},M,G,X,\Gamma})$ *and* $pat, pat' \in (T_{\Sigma^{Sat},X})_{Pat_{Focus}}$

When a pattern from an axiom is included in an existing pattern we require that all variables in the included pattern are renamed to avoid naming conflicts.

The expanded execution pattern set $ExpPat_{Spec}(i) \subseteq AllPat_{Spec}(i)$ *corresponds to the expanded patterns which do not contain a '*t in k*' condition, i.e. those that cannot be further expanded. It is defined as follows:*

$$ExpPat_{Spec}(i) = \big\{\langle cond, pat\rangle \in AllPat_{Spec}(i) \mid \nexists\, t, k \,.\, t\, in\, k \in cond\big\}$$

where $i, k \in \Gamma$, $cond \in \mathcal{P}(Cond_{\Sigma^{Sat},M,G,X,\Gamma})$ *and* $pat, t \in (T_{\Sigma^{Sat},X})_{Pat_{Focus}}$

```
    T in repeatAction;

    uniformity(c) | f in repeatAction =>
         f . Hml({action1(c)} T) in repeatAction;

    f in repeatAction, nbEvents(f) < 3 =>
         f . Hml({action2(c)} T) in useRepeatAction;

Variables
    f : primitiveHml;
    c : someUserType
```

Figure 8.2: Fictitious Test Intentions

Example 8.3.2 *Consider the 'repeatAction' and the 'useRepeatAction' test intentions defined in figure 8.2. Let us start by exemplifying the construction of the* full expanded execution pattern set *for the 'repeatAction' test intention.*

by a double application of the Local *rule we initially obtain:*

$$AllPat_{Spec}(repeatAction) \supseteq$$
$$\Big\{\big\langle\emptyset, T\big\rangle,\ \big\langle\{unif(c), f\, in\, repeatAction\},\ f.\langle action1(c)\rangle T\big\rangle\Big\}$$

then, by applying two times the RemUn *rule we can obtain for example the following sets:*

$$AllPat_{Spec}(repeatAction) \supseteq$$
$$\Big\{\big\langle\emptyset, T\big\rangle,\ \big\langle\{unif(c), f\, in\, repeatAction\},\ f.\langle action1(c)\rangle T\big\rangle,$$
$$\big\langle\{unif(c)\},\ T.\langle action1(c)\rangle T\big\rangle\Big\}$$

$$AllPat_{Spec}(repeatAction) \supseteq$$
$$\Big\{\langle \emptyset, T\rangle,\ \big\langle\{unif(c), f\ in\ repeatAction\},\ f.\langle action1(c)\rangle T\big\rangle,$$
$$\big\langle\{unif(c)\},\ T.\langle action1(c)\rangle T\big\rangle,$$
$$\big\langle\{unif(f'), unif(c), unif(c'), f'\ in\ repeatAction\},\ f'.\langle action1(c')\rangle T.\langle action1(c)\rangle T\big\rangle\Big\}$$

The expanded execution pattern set *for the 'useRepeatAction' test intention is formed by using the* full expanded axiom set $AllPat_{Spec}(repeatAction)$*. Note that* $ExpPat_{Spec}(repeatAction)$ *contains only the execution patterns of* $AllPat_{Spec}(repeatAction)$ *which cannot be further expanded.*

Finally, the expanded execution pattern set *for the* 'useRepeatAction*' test intention is built using the* expanded execution pattern set *for the* 'repeatAction*' test intention. The incomplete* $ExpPat_{Spec}(useRepeatAction)$ *set is as follows:*

$$ExpPat_{Spec}(useRepeatAction) = \Big\{\big\langle\{depth(T) < 3\},\ T.\langle action2(c)\rangle T\big\rangle,$$
$$\big\langle\{unif(c'), depth(T.\langle action1(c)\rangle T) < 3\},\ T.\langle action1(c')\rangle T.\langle action2(c)\rangle T\big\rangle, \ldots\Big\}$$

In definition 8.3.1 we start from the set of $\langle condition, execution\ pattern\rangle$ pairs describing the test intention we wish to expand (*Local* inference rule). By applying the *Include* inference rule we then build $\langle condition, execution\ pattern\rangle$ pairs where all the *inclusion* atomic conditions are removed from the conditions in those pairs. In order to do this the *Include* inference rule "chooses" all possible pattern replacements for an Hml variable belonging to an *inclusion* condition and replaces that variable in both the *condition* and the *pattern* part of the treated pair. The conditions of the included pair are added to the list of conditions of the pair being treated — which means new *inclusion* conditions may be introduced and the rule may be reapplied. Rules *RemUn* and *RemSubUn* are similar to rule *inclusion*, except that they remove the *uniform* and *subuniform* predicates since the variables they quantify are replaced by the imported patterns. All variables in an imported pattern are marked with either the '*unif*' or '*subUnif*' predicate if the variable they replace is marked with the same predicate.

Notice also that in definition 8.3.1 we may create an infinite amount of possibly arbitrarily lengthy test patterns, depending on the way the axioms in a test intention module are defined. We allow this to happen in our semantics as the evaluation of the term complexity measurement functions in the expanded axiom set may cut it down to a finite size.

Definition 8.3.3 *Execution Pattern Validation*

Let Spec be a CO-OPN and SATEL specification having a global annotated SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be a finitely generated annotated Σ^{Sat}-SATEL algebra. Let also $Md^{\mathsf{X}} \in Spec$ be a context module *with signature Ξ and $SP = \langle Q, Ev, Tr, q_0\rangle$ be the* transition system semantics *of Md^{X}. The execution pattern validation is the $\models_{Pat_\Xi} \subseteq SP \times Q \times A_{Pat_\Xi}$ satisfaction relation defined as follows:*

- $SP, q \models_{Pat_\Xi} T$
- $SP, q \models_{Pat_\Xi} (\neg f) \Leftrightarrow G, q \not\models_{Pat_\Xi} f$
- $SP, q \models_{Pat_\Xi} (f \wedge g) \Leftrightarrow G, q \models_{Pat_\Xi} f \ \wedge \ G, q \models_{Pat_\Xi} g$
- $SP, q \models_{Pat_\Xi} (\langle stm \,\boldsymbol{with}\, obs\rangle f) \Leftrightarrow \exists\, q' \in Q \;.\; q \xrightarrow{stm\,\boldsymbol{with}\,obs} q' \in Tr \ \wedge$ $SP, q \models_{Pat_\Xi} f$

$\not\models_{Pat_\Xi} \subseteq SP \times Q \times A_{Pat_\Xi}$

- $SP, q \not\models_{Pat_\Xi} (\neg f) \Leftrightarrow SP, q \models_{Pat_\Xi} f$
- $SP, q \not\models_{Pat_\Xi} (f \wedge g) \Leftrightarrow SP, q \not\models_{Pat_\Xi} f \ \vee \ SP, q \not\models_{Pat_\Xi} g$
- $SP, q \not\models_{Pat_\Xi} (\langle stm \,\boldsymbol{with}\, obs\rangle f) \Leftrightarrow \exists\, q' \in Q \;.\; q \xrightarrow{stm\,\boldsymbol{with}\,obs} q' \in Tr \ \wedge$ $SP, q \not\models_{Pat_\Xi} f$
- $SP, q \not\models_{Pat_\Xi} (\langle stm \,\boldsymbol{with}\, obs\rangle f) \Leftrightarrow \nexists\, q' \in Q \;.\; q \xrightarrow{stm\,\boldsymbol{with}\,obs} q' \in Tr$

where $stm \in (T_{\Sigma^{Sat},\emptyset})_{Stim_\Xi}$ and $obs \in (T_{\Sigma^{Sat},\emptyset})_{Obsrv_\Xi}$.

Notice that in order to simplify the definition we have omitted events without observation, although they are naturally included in the satisfaction relation.

Definition 8.3.4 *Exhaustive Positive and Negative Substitution Sets for a Test Intention, Exhaustive Test Set for a Test Intention*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be any finitely generated Σ^{Sat}-SATEL algebra. Let also $Md^{\mathsf{T}} = \langle \Gamma, Focus, \Lambda, X\rangle \in Spec$ be a test intention module *and $SP = \langle Q, Ev, Tr, q_0\rangle = Sem_{Spec,A}(Md^{\mathsf{X}})$ (see definition 5.4.10) be the semantics of context module $Md^{\mathsf{X}}_{Focus} \in Spec$. The* exhaustive positive substitution set for a test intention $i \in \Gamma$ *is defined as follows:*

$$\begin{aligned}ExhaustSubs^{+}_{Spec}(i) = \big\{\langle\theta, pat\rangle \in GroundSubs_S(Var(pat)) \times (T_{\Sigma^{Sat},X})_{Pat_{Focus}} \mid\\ \langle cond, pat\rangle \in ExpPat_{Spec}(i) \wedge\\ SP, q_0 \models_{Pat_{Focus}} [\![\theta^{\#}(pat)]\!]_{Pat_{Focus}}\big\}\end{aligned}$$

The exhaustive negative substitution set for a test intention $i \in \Gamma$ *is defined as follows:*

$$\begin{aligned}ExhaustSubs^{-}_{Spec}(i) = \big\{\langle\theta, pat\rangle \in GroundSubs_S(Var(pat)) \times (T_{\Sigma^{Sat},X})_{Pat_{Focus}} \mid\\ \langle cond, pat\rangle \in ExpPat_{Spec}(i) \wedge\\ SP, q_0 \not\models_{Pat_{Focus}} [\![\theta^{\#}(pat)]\!]_{Pat_{Focus}}\big\}\end{aligned}$$

The exhaustive test set for a test intention $i \in \Gamma$ *is defined as follows:*

$$\begin{aligned}&Exhaust_{Spec}(i) =\\ &\big\{\langle\theta^{\#}(pat), true\rangle \in (T_{\Sigma^{Sat},X})_{Pat_{Focus}} \times \{true, false\} \mid \langle\theta, pat\rangle \in ExhaustSubs^{+}_{Spec}(i)\big\}\\ &\cup\big\{\langle\theta^{\#}(pat), false\rangle \in (T_{\Sigma^{Sat},X})_{Pat_{Focus}} \times \{true, false\} \mid \langle\theta, pat\rangle \in ExhaustSubs^{-}_{Spec}(i)\big\}\end{aligned}$$

We can now state an important result regarding the expressiveness of SATEL to build *exhaustive test sets.*

Theorem 8.3.5 *Completeness of the Test Intention Language SATEL*

Given a CO-OPN$_{/2c++}$ specification Spec, it is possible to write in SATEL a test intention *that establishes the correctness of an SUT regarding a model given as a* context module $CtxMod \in Spec$.

Proof. *By construction:*

```
TestIntentionSet Example Focus CtxMod;
   Interface
      Intentions
          ExhaustiveTestSet;

   Body
      Axioms
          f in ExhaustiveTestSet;

Variables
     f : primitiveHML;

End Example;
```

*By definition 8.4.6, the '*ExhaustiveTestSet*'* test intention *produces the set* test cases *corresponding to all HML formulas built using events formed from the signature of the context module CtxMod. The definition of* HML equivalence *tells us that two transition systems are HML equivalent if, for all HML formulas built using the set of events of those transition system, both transition systems are models of those formulas. On the other hand, there is a* full agreement *between between HML equivalence and bisimulation equivalence — as stated in [35]. This means that HML equivalence implies bisimulation equivalence. In definition 6.2.2 the correctness of an SUT regarding a specification has been defined as the existence of a bisimulation relation between the transition systems denoting the semantics of the SUT and the specification. Given that by definition 5.4.10 context module CtxMod has a transition system semantics, the '*ExhaustiveTestSet*'* test intention *produces a test set capable of establishing the correctness of an SUT specified by CtxMod.*

Corollary 8.3.6 *Production of a* pertinent *test set using SATEL*

Given a CO-OPN$_{/2c++}$ specification Spec, it is possible to produce using SATEL a pertinent — valid *and* unbiased — *test set for a* context module $CtxMod \in$ *Spec.*

8.4 Reduced Test Set for a Test Intention

Definition 8.4.1 *Annotation*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be a finitely generated Σ^{Sat}-SATEL algebra. Let also $Md_i^{\mathsf{C}} = \langle \Omega_i^{\mathsf{C}}, P_i, I_i, X_i, \Psi_j\rangle \in$ Spec $(1 \leq i \leq n)$ and $Md_j^{\mathsf{X}} = \langle \Xi_j^X, O_j, C_j, X_j, \chi_j\rangle \in$ Spec $(1 \leq j \leq m)$ be a set of class *and* context *modules. The set of annotations for specification Spec is noted $Ann_{Spec,A}$ and is defined as follows:*

$$Ann_{Spec,A} = \mathcal{P}\big((Ax^{\mathsf{C}} \times \widehat{A}) \cup Ax^{\mathsf{X}}\big)$$

where

- $Ax^{\mathsf{C}} = \bigcup_{1\leq i\leq n} \Psi_j$ *and* $Ax^{\mathsf{X}} = \bigcup_{1\leq i\leq n} \chi_j$ *are respectively sets containing all the class behavioral formulas and the context coordination formulas present in specification Spec;*
- $\widehat{A} \subseteq A$ *corresponds to the* object identifier *algebra for specification Spec.*

The notion of annotation provides us with the tools to "instrument" the transition system corresponding to the semantics of *context module* we have introduced in definition 5.4.10. We are interested in marking each event of the transition system with a set of structural conditions which leads to the fact that the event is part of a context module's semantics. An *annotation* as defined in 8.4.1 is a set including possibly a set of *(behavioral formula, object identifier)* pairs and a set of *coordination formulas* — enough to identify each event with all the components of the specification involved in its existence.

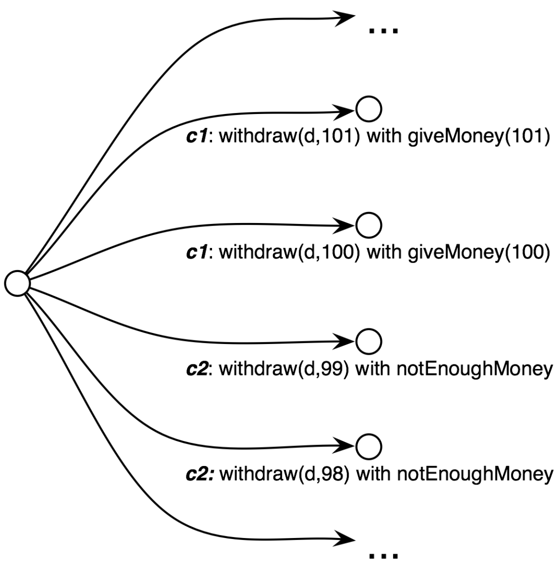

...
c1: withdraw(d,101) with giveMoney(101)
c1: withdraw(d,100) with giveMoney(100)
c2: withdraw(d,99) with notEnoughMoney
c2: withdraw(d,98) with notEnoughMoney
...

Figure 8.3: Sample Annotation of the Banking Server's Semantics

Definition 8.4.2 *Annotated Transition System Semantics of a Context Module*

Let Spec be a CO-OPN and SATEL specification and A be a finitely generated Σ^{Sat}-SATEL algebra. Let also $Md^{\mathsf{X}} \in Spec$ be a context module. *The* annotated transition system semantics *of Md^{X} corresponds to the* context module *semantics $Sem_{Spec,A}(Md^{\mathsf{X}})$ (see definition 5.4.10) where each event of the transition system is augmented by an annotation reflecting all the* coordination formulas *and* behavioral formulas *that lead to the event's existence. Also, each behavioral formula in an annotation is extended by the* class instance *it belongs to, i.e. the object identifier.*

Let us introduce the following notation: given $Sem_{Spec,A}(Md^{\mathsf{X}}) = \langle Q, Ev, Tr, i\rangle$ the transition system representing the semantics of Md^{X}, the annotated transition system semantics of Md^{X} is a transition system $\langle Q, Ev, Tr', i\rangle$ where Tr' is formed as follows:

$$q \xrightarrow{e} q' \in Tr \Rightarrow q \xrightarrow{c\,:\,e} q' \in Tr'$$

such that $c \in Ann_{Spec,A}$ *is the least set of annotations reflecting the firing of* e.

The Annotated Transition System Semantics of a Context Module *is obtained by augmenting the inference rules we propose in definitions 5.4.1 through 5.4.9 with the appropriate constructs. We do not present the rules in this thesis as they would amount to a repetition of the inference rules in definitions 5.4.1 through 5.4.9 with minor changes.*

Example 8.4.3 *Consider the Banking Server example we have introduced in section 4.3. In figure 8.3 we present the annotation of a sample of the semantics of the* Banking Server *context module in figure 4.7. Figure 8.3 describes a sample of a transition system where the nodes correspond to states of the* Banking Server *context module and the edges to labeled transitions denoting state change. In the leftmost state a user '*d*' is logged in having 100CHF in his/her account. Four events in the transition system then correspond to the withdrawal by user '*d*' of different amounts of money.*

*Two different kinds of '*withdraw*' events can be observed in figure 8.3. The events where the withdrawal is possible are annotated with '***c1***', while the ones where the withdrawal is not possible are annotated with '***c2***'. Annotation '***c1***' corresponds to the set which is the union of the following* coordination formulas *and* (behavioral formula, object identifier) *pairs:*

$$\left.\begin{array}{ll}
1. & \text{`}withdraw\,usr\,am\ \mathbf{with}\ lmObj.isLogged\,usr\,//\,accVar.withdraw\,am;\text{'}\\
2. & \big(\text{`}isLogged\,usr\ ::\ logged\,usr\ \rightarrow\ logged\,usr;\text{'}\,,\ lmObj\big)\\
3. & \big(\text{`}(b >= am) = true\ =>\\
 & \qquad withdraw\,am\ \mathbf{with}\ this.giveMoney\,am\ ::\\
 & \qquad\qquad balance\,b \rightarrow balance\,b - am;\text{'}\,,\ accObj1\big)\\
4. & accVar.giveMoney\,am\ \mathbf{with}\ giveMoney\,am;
\end{array}\right\} = \mathbf{c}1$$

Going back to the graphical representation of the *Banking Server* context module in figure 4.7 (full specification in appendix B), axioms 1 and 4 in the '***c1***' annotation correspond to the connections of the objects involved in the event with the borders of the context. Axiom 2 corresponds to checking if user '*d*' is logged — in the '*lmObj*' which is also part of the annotation — and axiom 3 corresponds to the actual withdrawal of the money in the '*accObj1*' object (assuming the account the user '*d*' is managed by the '*accObj1*' object).

Finally, the '***c2***' annotation is similar to '***c1***' except for the condition '$(b >= am) = true$' in axiom 3 allowing the withdrawal of money, which is changed to '$(b >= am) = false$'.

Definition 8.4.4 *Annotated SATEL Terms, Annotated SATEL Algebra*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F \rangle$ where $\Xi = \bigcup_{1 \leq i \leq n} \Xi_i$. Let also A be a finitely generated Σ^{Sat}-SATEL algebra. The annotated SATEL terms *of sort $s \in S$ are noted $(T^{Ann}_{\Sigma^{Sat},\emptyset})_s$ and are obtained by augmenting the terms of $(T_{\Sigma^{Sat},\emptyset})_s$ as follows:*

- $c\!:\!t \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_s$ *for all* $c \in Ann_{Spec,A}$, $t \in (T_{\Sigma^{Sat},\emptyset})_s$ *and* $s \in S \setminus (Stim_{\Xi_i} \cup Obsrv_{\Xi_i} \cup Hml_{\Xi_i})$
- $c\!:\!t \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Stim_{\Xi_i}}$ *for all* $c \in Ann_{Spec,A}$, $t \in (T_{\Sigma^{Sat},\emptyset})_{Stim_{\Xi_i}}, Stim_{\Xi_i} \in S$
- $c\!:\!t \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Obsrv_{\Xi_i}}$ *for all* $c \in Ann_{Spec,A}$, $t \in (T_{\Sigma^{Sat},\emptyset})_{Obsrv_{\Xi_i}}, Obsrv_{\Xi_i} \in S$
- $C\!:\!t \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Hml_{\Xi_i}}$ *for all* $C \in \mathcal{P}(Ann_{Spec,A})$, $t \in (T_{\Sigma^{Sat},\emptyset})_{Hml_{\Xi_i}}, Hml_{\Xi_i} \in S$

In definition 8.4.4 we have slightly abused the notion of algebraic specification as we have mixed the *object identifier algebra* used in the definition of *annotation* (see definition 8.4.1) with the SATEL signature. This slight abuse allows us to avoid redefining annotations using signature object identifier operations which would be trivial and would not add to the understanding of the subsequent text.

In the text that follows we will us the notation $A^{Ann}_{Pat_\Xi}$ to denote the set of Hml formulas built — for a given context module with signature Ξ — from both *annotated* and *non annotated* subterms. As an example, assume a specification having as events $\{a\ with\ b, d(2)\}$ and a couple of annotations $\{c1, c2\}$. The following would be terms of $A^{Ann}_{Pat_\Xi}$:

$$\langle a\ with\ b\rangle\ \langle d(2)\rangle\ T$$

$$\langle \mathbf{c}1\!:\!a\ with\ b\rangle\ \langle d(\mathbf{c}2\!:\!2)\rangle\ T$$

$$\{\mathbf{c}1, \mathbf{c}2\}\big(\langle \mathbf{c}1\!:\!a\ with\ b\rangle\ \langle d(\mathbf{c}2\!:\!2)\rangle\big)\ T$$

Given a SATEL signature $\Sigma^{Sat} = \langle S, \leq, F \rangle$ and an S-Sorted set X of variables we will also extend the set of substitutions $GroundSubs_S(X)$ (see definition 3.2.2) by the $GroundAnnSubs_S(X)$ set of annotated substitutions which is a family of functions with signature $\theta_s : X_s \to (T^{Ann}_{\Sigma^{Sat},\emptyset})_s$ where $s \in S$.

In the subsequent text we will also use the function *stripped* which, given an annotated term returns its non-annotated counterpart. For example:

$$stripped\big(\{\mathbf{c}1, \mathbf{c}2\}(\langle \mathbf{c}1\!:\!a\ with\ b\rangle\ \langle d(\mathbf{c}2\!:\!2)\rangle)\ T\big) = \langle a\ with\ b\rangle\ \langle d(2)\rangle\ T$$

The *stripped* function is also extended to substitutions $GroundAnnSubs_S(X)$, by converting a substitution $\theta \in GroundAnnSubs_S(X)$ into a substitution $\theta' \in$

$GroundSubs_S(X)$. The conversion is achieved by stripping all the annotated terms replacing variables of X by their non annotated counterparts. For example:

$$stripped(\theta) = \theta' \text{ where } \theta = \{[d(\mathbf{c}2:2)/x] \text{ and } \theta' = \{[d(2)/x]\}$$

Definition 8.4.5 *Annotated Execution Pattern Validation*

Let Spec be a CO-OPN and SATEL specification having a global annotated SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be a finitely generated annotated Σ^{Sat}-SATEL algebra. Let also $Md^{\mathsf{X}} \in Spec$ be a context module *with signature Ξ and $SP_{Ann} = \langle Q, Ev, Tr, q_0\rangle$ be the annotated* transition system semantics *of Md^{X}. The annotated execution pattern validation is the $\models^{Ann}_{Pat_\Xi} \subseteq SP_{Ann} \times Q \times A^{Ann}_{Pat_\Xi}$ satisfaction relation defined as follows:*

- $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} T$
- $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} (\neg f) \Leftrightarrow SP_{Ann}, q \not\models^{Ann}_{Pat_\Xi} f$
- $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} (\neg\, C : f) \Leftrightarrow SP_{Ann}, q \not\models^{Ann}_{Pat_\Xi} C : f$
- $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} (f \wedge g) \Leftrightarrow SP_{Ann}, q \models^{Ann}_{Pat_\Xi} f \;\wedge\; SP_{Ann}, q \models^{Ann}_{Pat_\Xi} g$
- $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} C : (f \wedge g) \Leftrightarrow SP_{Ann}, q \models^{Ann}_{Pat_\Xi} C' : f \;\wedge$ $SP_{Ann}, q \models^{Ann}_{Pat_\Xi} C'' : g \;\wedge\; C' \cup C'' = C \;\wedge\; C' \cap C'' = \emptyset$
- $SP_{Ann}, q \models_{Pat_\Xi} \langle stm\ \boldsymbol{with}\ obs\rangle\, f \Leftrightarrow$ $\exists q' \in Q \,.\, q \xrightarrow{c\,:\,stm\ \boldsymbol{with}\, obs} q' \in Tr \;\wedge\; SP_{Ann}, q \models_{Pat_\Xi} f$
- $SP_{Ann}, q \models_{Pat_\Xi} \{c\} : \big(\langle stm'\ \boldsymbol{with}\ obs'\rangle\, f\big) \Leftrightarrow$ $\exists q' \in Q \,.\, q \xrightarrow{c\,:\,stm\ \boldsymbol{with}\, obs} q' \in Tr \;\wedge\; SP_{Ann}, q \models_{Pat_\Xi} f \;\wedge\; stripped(stm') = stm \;\wedge\; stripped(obs') = obs$ *and all the annotated subterms of stm' and obs' are annotated with c*
- $SP_{Ann}, q \models_{Pat_\Xi} \{c\} \cup C : \big(\langle stm'\ \boldsymbol{with}\ obs'\rangle\, f\big) \Leftrightarrow$ $\exists q' \in Q \,.\, q \xrightarrow{c\,:\,stm\ \boldsymbol{with}\, obs} q' \in Tr \;\wedge\; SP_{Ann}, q \models_{Pat_\Xi} C : f \;\wedge\; stripped(stm') = stm \;\wedge\; stripped(obs') = obs$ *and all the annotated subterms of stm' and obs' are annotated with c*

$\not\models^{Ann}_{Pat_\Xi} \subseteq SP_{Ann} \times Q \times A^{Ann}_{Pat_\Xi}$

- $SP_{Ann}, q \not\models^{Ann}_{Pat_\Xi} (\neg f) \Leftrightarrow SP_{Ann}, q \models^{Ann}_{Pat_\Xi} f$
- $SP_{Ann}, q \not\models^{Ann}_{Pat_\Xi} (f \wedge g) \Leftrightarrow SP, q \not\models^{Ann}_{Pat_\Xi} f \;\vee\; SP_{Ann}, q \not\models^{Ann}_{Pat_\Xi} g$

- $SP_{Ann}, q \not\models_{Pat_{\Xi}} \langle stm\ \textbf{\textit{with}}\ obs\rangle\ f \Leftrightarrow$
  $\exists q' \in Q\ .\ q \xrightarrow{c\,:\,stm\ \textbf{\textit{with}}\ obs} q' \in Tr\ \wedge\ SP_{Ann}, q \not\models^{Ann}_{Pat_{\Xi}} f$

- $SP_{Ann}, q \not\models_{Pat_{\Xi}} \{c\} : \big(\langle stm'\ \textbf{\textit{with}}\ obs'\rangle\ f\big) \Leftrightarrow$
  $\exists q' \in Q\ .\ q \xrightarrow{c\,:\,stm\ \textbf{\textit{with}}\ obs} q' \in Tr\ \wedge\ SP_{Ann}, q \not\models^{Ann}_{Pat_{\Xi}} f\ \wedge\ stripped(stm') = stm\ \wedge\ stripped(obs') = obs$ *and all the annotated subterms of* stm' *and* obs' *are annotated with* c

- $SP_{Ann}, q \not\models_{Pat_{\Xi}} \{c\} \cup C : \big(\langle stm'\ \textbf{\textit{with}}\ obs'\rangle\ f\big) \Leftrightarrow$
  $\exists q' \in Q\ .\ q \xrightarrow{c\,:\,stm\ \textbf{\textit{with}}\ obs} q' \in Tr\ \wedge\ SP_{Ann}, q \not\models^{Ann}_{Pat_{\Xi}} C : f\ \wedge\ stripped(stm') = stm\ \wedge\ stripped(obs') = obs$ *and all the annotated subterms of* stm' *and* obs' *are annotated with* c

- $SP_{Ann}, q \not\models^{Ann}_{Pat_{\Xi}} (\langle stm'\ \textbf{\textit{with}}\ obs'\rangle\ f) \Leftrightarrow \nexists q' \in Q\ .\ q \xrightarrow{c\,:\,stm\ with\ obs} q' \in Tr\ \wedge\ stripped(stm') = stm\ \wedge\ stripped(obs') = obs$

where:

- $f \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Pat_{\Xi}}$, $stm' \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Stim_{\Xi}}$, $obs' \in (T^{Ann}_{\Sigma^{Sat},\emptyset})_{Obsrv_{\Xi}}$

- $stm \in (T_{\Sigma^{Sat},\emptyset})_{Stim_{\Xi}}$, $obs \in (T_{\Sigma^{Sat},\emptyset})_{Obsrv_{\Xi}}$

- $c \in Ann_{Spec,A}$ *and* $C, C', C'' \in \mathcal{P}(Ann_{Spec,A})$

  Notice that in order to simplify the definition we have omitted events without observation, although they are naturally included in the satisfaction relation.

Definition 8.4.6 *Exhaustive Annotated Positive and Negative Substitution Sets*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ *and A be any finitely generated* Σ^{Sat}*-SATEL algebra. Let also* $Md^{\mathsf{T}} = \langle \Gamma, Focus, \Lambda, X\rangle \in Spec$ *be a* test intention module *and* $SP = \langle Q, Ev, Tr, q_0\rangle = Sem_{Spec,A}(Md^{\mathsf{X}})$ *(see definition 5.4.10) be the semantics of context module* $Md^{\mathsf{X}}_{Focus} \in Spec$*. The* exhaustive positive substitution set for a test intention $i \in \Gamma$ *is defined as follows:*

$$\begin{aligned} ExhaustAnnSubs^{+}_{Spec}(i) = \big\{\langle \theta, pat\rangle \in GroundAnnSubs_S(Var(pat)) \times (T_{\Sigma^{Sat},X})_{Pat_{Focus}} |\\ \langle cond, pat\rangle \in ExpPat_{Spec}(i)\ \wedge\\ SP, q_0 \models^{Ann}_{Pat_{Focus}} [\![\theta^{\#}(pat)]\!]_{Pat_{Focus}}\big\}\end{aligned}$$

The exhaustive negative substitution set for a test intention $i \in \Gamma$ *is defined as follows:*

$$\begin{aligned}ExhaustAnnSubs^{-}_{Spec}(i) = \big\{&\langle\theta, pat\rangle \in GroundAnnSubs_S(Var(pat)) \times (T_{\Sigma^{Sat},X})_{Pat_{Focus}} | \\ &\langle cond, pat\rangle \in ExpPat_{Spec}(i) \wedge \\ &SP, q_0 \not\models^{Ann}_{Pat_{Focus}} [\![\theta^{\#}(pat)]\!]_{Pat_{Focus}}\big\}\end{aligned}$$

Definition 8.4.7 *Exhaustive Positive and Negative Annotated Substitution Set for an Expanded Pattern, Reduced Positive and Negative Annotated Substitution Set for an Expanded Pattern*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be any finitely generated Σ^{Sat}-SATEL algebra. Let also $Md^T = \langle \Gamma, Focus, \Lambda, X\rangle \in Spec$ be a test intention module *and $SP = \langle Q, Ev, Tr, q_0\rangle = Sem_{Spec,A}(Md^{\mathsf{X}}_{Focus})$ be the semantics of context module $Md^{\mathsf{X}}_{Focus} \in Spec$. The positive exhaustive substitution set for an expanded pattern $\langle cond, pat\rangle \in ExpPat_{Spec}(i)$ of a test intention $i \in \Gamma$ is defined as follows:*

$$\begin{aligned}&ExhaustSubsPat^{+}_{Spec}(\langle cond, pat\rangle) = \\ &\qquad \big\{\theta \in GroundAnnSubs_S(Var(pat)) \mid \langle\theta, pat\rangle \in ExhaustAnnSubs^{+}_{Spec}(i)\big\}\end{aligned}$$

Finally, the negative exhaustive substitution set for an expanded pattern $\langle cond, pat\rangle \in ExpPat_{Spec}(i)$ of a test intention $i \in \Gamma$ is defined as follows:

$$\begin{aligned}&ExhaustSubsPat^{-}_{Spec}(\langle cond, pat\rangle) = \\ &\qquad \big\{\theta \in GroundAnnSubs_S(Var(pat)) \mid \langle\theta, pat\rangle \in ExhaustAnnSubs^{-}_{Spec}(i)\big\}\end{aligned}$$

The reduced positive substitution set for an expanded pattern is defined as follows:

$$\begin{aligned}&RedSubsPat^{+}_{Spec}(\langle cond, pat\rangle) = \\ &\qquad \big\{\Theta \in \mathcal{P}(GroundAnnSubs_S(Var(pat))) \mid \Theta \subseteq ExhaustSubsPat^{+}_{Spec}(pat) \wedge \\ &\qquad \big(\forall\theta \in \Theta \;.\; A, stripped(\theta) \models_{\mathsf{c}} cond\big) \wedge \\ &\qquad \big(\forall x \in Var(pat)\,, \forall\theta, \theta' \in \Theta \;. \\ &\qquad\qquad unif(x) \in cond \Rightarrow (\forall\; [c : v/x] \in \theta,\; [c' : v'/x] \in \theta' \;.\; v = v') \vee \\ &\qquad\qquad subunif(x) \in cond \Rightarrow (\forall\; [c : v/x] \in \theta,\; [c : v'/x] \in \theta' \;.\; v = v')\big)\big\}\end{aligned}$$

The reduced negative substitution set for an expanded pattern is defined as follows:

$$\begin{aligned}
&RedSubsPat^{-}_{Spec}(\langle cond, pat\rangle) = \\
&\qquad \big\{\Theta \in \mathcal{P}(GroundAnnSubs_S(Var(pat))) \mid \Theta \subseteq ExhaustSubsPat^{-}_{Spec}(pat) \wedge \\
&\qquad \big(\forall \theta \in \Theta \;.\; A, stripped(\theta) \vDash_{\mathsf{c}} cond\big) \wedge \\
&\qquad \big(\forall x \in Var(pat)\,, \forall \theta, \theta' \in \Theta \;. \\
&\qquad\qquad unif(x) \in cond \Rightarrow (\forall\, [c : v/x] \in \theta,\ [c' : v'/x] \in \theta' \;.\; v = v') \vee \\
&\qquad\qquad subunif(x) \in cond \Rightarrow (\forall\, [c : v/x] \in \theta,\ [c : v'/x] \in \theta' \;.\; v = v')\big)\big\}
\end{aligned}$$

Definition 8.4.8 *Reduced Test Set for a Test Intention*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F\rangle$ and A be any finitely generated Σ^{Sat}-SATEL algebra. Let also $Md^T = \langle \Gamma, Focus, \Lambda, X\rangle \in Spec$ be a test intention module *and $SP = \langle Q, Ev, Tr, q_0\rangle = Sem_{Spec,A}(Md^{\mathsf{X}}_{Focus})$ be the semantics of context module $Md^{\mathsf{X}}_{Focus} \in Spec$. Finally let $i \in \Gamma$ be a test intention, $\boldsymbol{\Theta}^{+}_{k} = RedSubsPat^{+}_{Spec}\big(\langle cond_k, pat_k\rangle\big)$ and $\boldsymbol{\Theta}^{-}_{k} = RedSubsPat^{-}_{Spec}\big(\langle cond_k, pat_k\rangle\big)$ be the positive and negative reduced substitution sets for $\langle cond_k, pat_k\rangle \in ExpPat_{Spec}(i)$ $(1 \leq k \leq n)$. The reduced test set for a test intention i is defined as follows:*

$$Reduced_{Spec}(i) = \bigcup_{1\leq k\leq n}\Big(\bigcup_{1\leq j\leq n}\langle \theta^{\#}_{j}(pat_k), true\rangle\Big) \cup \bigcup_{1\leq k\leq n}\Big(\bigcup_{1\leq l\leq n}\langle \theta^{\#}_{l}(pat_k), false\rangle\Big)$$

where $\theta_j = stripped(\theta'_j)$, $\theta'_j \in \Theta^{+}_{k}$, $\theta_l = stripped(\theta'_l)$, $\theta_l \in \Theta^{-}_{k}$ and Θ^{+}_{k} and Θ^{-}_{k} are sets of substitutions chosen from the sets of substitutions available in $\boldsymbol{\Theta}^{+}_{k}$ and $\boldsymbol{\Theta}^{-}_{k}$ respectively.

```
uniformity(wam) | ( dam > 0) = true , (dam <= 3) = true =>
          Hml({deposit(d,dam) with null} {withdraw(d,wam) with obs} T)
                   in unWithdraw;

subUniformity(wam) | ( dam > 0) = true , (dam <= 3 ) = true =>
          Hml({deposit(d,dam) with null} {withdraw(d,wam) with obs} T)
                   in subUnWithdraw;

Variables
          dam,wam : natural;
          obs : primitiveObservation;
```

Figure 8.4: Partial Test Intention for the Banking Server

Example 8.4.9 *Consider the* 'unifTestAccount' *and* 'subUnifTestAccount' *test intentions defined in figure 8.4 and having as* focus *the* Banking Server *context module* *in figure 4.7 belonging to the Banking specification. We assume a user '*d*' already logged in and having 0CHF in his/her account. A sample of the annotated transition system for this context module is presented in figure 8.7. In the leftmost state we assume a user '*d*' already logged in and having 0CHF in his/her account. Annotations '*c1*', '*c2*' and '*c3*' correspond to different firing conditions, as previously introduced in definition 8.4.1.*

We will build the positive reduced test set *for the '*unWithdraw*' and '*subUnWithdraw*' test intentions. Let us start by building the* expanded execution pattern *(definition 8.3.1) set for both test intentions:*

$$AllPat_{Banking}(unWithdraw) = \Big\{\big\langle\{unif(wam), (dam > 0) = true, (dam <= 3) = true\}, \langle deposit(d, dam)\rangle\langle withdraw(d, wam), obs\rangle T\big\rangle\Big\}$$

$$AllPat_{Banking}(subUnWithdraw) = \Big\{\big\langle\{subUnif(wam), (dam > 0) = true, (dam <= 3) = true\}, \langle deposit(d, dam)\rangle\langle withdraw(d, wam), obs\rangle T\big\rangle\Big\}$$

*In order to produce the reduced test set for test intentions '*unWithdraw*' and '*subUnWithdraw*' it is necessary to first build the* exhaustive substitution set *(see definition 8.4.7) for the* expanded execution patterns*. The* positive exhaustive substitution set *for both previously obtained* expanded execution patterns *is as follows:*

$$
\begin{aligned}
&ExhaustSubsPat^{+}_{Banking}\big(\langle\{unif(wam),(dam>0)=true,(dam<=3)=true\},\\
&\qquad\langle deposit(d,dam)\rangle\langle withdraw(d,wam),obs\rangle T\rangle\big)=\\
&ExhaustSubsPat^{+}_{Banking}\big(\langle\{subUnif(wam),(dam>0)=true,(dam<=3)=true\},\\
&\qquad\langle deposit(d,dam)\rangle\langle withdraw(d,wam),obs\rangle T\rangle\big)=\\
&\Big\{\big\{[c1:1/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\big\},\\
&\big\{[c1:1/dam],[c3:2/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:1/dam],[c3:3/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:1/dam],[c3:4/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:1/dam],[c3:5/wam],[c3:notEnoughMoney/obs]\big\},\\
&\ldots\\
&\big\{[c1:2/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\big\},\\
&\big\{[c1:2/dam],[c2:2/wam],[c2:giveMoney(2)/obs]\big\},\\
&\big\{[c1:2/dam],[c3:3/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:2/dam],[c3:4/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:2/dam],[c3:5/wam],[c3:notEnoughMoney/obs]\big\},\\
&\ldots\\
&\big\{[c1:3/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\big\},\\
&\big\{[c1:3/dam],[c2:2/wam],[c2:giveMoney(2)/obs]\big\},\\
&\big\{[c1:3/dam],[c2:3/wam],[c2:giveMoney(3)/obs]\big\},\\
&\big\{[c1:3/dam],[c3:4/wam],[c3:notEnoughMoney/obs]\big\},\\
&\big\{[c1:3/dam],[c3:5/wam],[c3:notEnoughMoney/obs]\big\},\\
&\ldots\\
&\big\{[c1:4/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\big\},\\
&\big\{[c1:4/dam],[c2:2/wam],[c2:giveMoney(2)/obs]\big\},\ldots\Big\}
\end{aligned}
$$

Notice that the positive exhaustive substitution set *consists of all the substitutions of variables 'dam', 'wam' and 'obs' which yield ground Hml formulas which are satisfied (see definition 8.4.5) by the transition system in figure 8.7 denoting the semantics of the* BankingServer *context module. The presented* positive exhaustive substitution set *is partial given that the full set would include all combinations of values for* deposit *and* withdraw *operations for user* 'd'.

The reduced substitution set *(see definition 8.4.7) for the single* expanded execution pattern *for the* 'unWithdraw' *test intention is as follows:*

$$\begin{aligned}
RedSubsPat^{+}_{Banking}\big(\langle\{&unif(wam),(dam>0)=true,(dam<=3)=true\},\\
&\langle deposit(d,dam)\rangle\langle withdraw(d,wam),obs\rangle T\rangle\big)=\\
\Big\{\{&\{[c1:1/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\},\\
&\{[c1:2/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\},\\
&\{[c1:3/dam],[c2:1/wam],[c2:giveMoney(1)/obs]\}\},\\[1em]
\{&\{[c1:1/dam],[c3:2/wam],[c3:notEnoughMoney/obs]\},\\
&\{[c1:2/dam],[c2:2/wam],[c2:giveMoney(2)/obs]\},\\
&\{[c1:3/dam],[c2:2/wam],[c2:giveMoney(2)/obs]\}\},\\[1em]
\{&\{[c1:1/dam],[c3:3/wam],[c3:notEnoughMoney/obs]\},\\
&\{[c1:2/dam],[c3:3/wam],[c3:notEnoughMoney/obs]\},\\
&\{[c1:3/dam],[c2:3/wam],[c2:giveMoney(3)/obs]\}\}\Big\}
\end{aligned}$$

The reduced substitution set *includes the substitutions from the* positive exhaustive substitution set *obeying the constraints imposed by the algebraic conditions on the 'dam' variables and the* uniform *condition on the 'wam' variable — as stated in definition 8.4.7. The* reduced substitution set *is in fact a set of sets of substitutions, with each of the subsets encapsulating a possibility for the* uniformity *hypothesis. The possibilities for the* uniform *condition on the 'wam' variable are either having the value of '1', '2' or '3', with the values of 'dam' oscillating between the {1,2,3} values allowed by the conditions of the* 'unWithdraw' *test intention.*

```
HML( {deposit(d,1) with null} {withdraw(d,3), notEnoughMoney} T), True
HML( {deposit(d,2) with null} {withdraw(d,3), notEnoughMoney} T), True
HML( {deposit(d,2) with null} {withdraw(d,3), giveMoney(3)} T), True
```

Figure 8.5: $Reduced_{Banking}(unWithdraw)$

We present in figure 8.5 the reduced test set (see definition 8.4.8) for the 'unWithdraw' *test intention. The* test set *is obtained by randomly choosing one substitution set from* reduced substitution sets *we have previously calculated.*

Finally, the reduced substitution set *for the single* expanded execution pattern *for the* 'subUnWithdraw' *test intention is as follows:*

$$
\begin{aligned}
&RedSubsPat^{+}_{Banking}\big(\langle\{subUnif(wam), (dam > 0) = true, (dam <= 3) = true\},\\
&\qquad \langle deposit(d, dam)\rangle\langle withdraw(d, wam), obs\rangle T\rangle\big) =\\
&\Big\{\{\{[c1 : 1/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 2/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 3/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 2/wam], [c3 : notEnoughMoney/obs]\}\},\\
&\\
&\{\{[c1 : 1/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 2/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 3/dam], [c2 : 1/wam], [c2 : giveMoney(1)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\},\\
&\quad \{[c1 : 2/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\}\},\\
&\\
&\{\{[c1 : 2/dam], [c2 : 2/wam], [c2 : giveMoney(2)/obs]\},\\
&\quad \{[c1 : 3/dam], [c2 : 2/wam], [c2 : giveMoney(2)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 2/wam], [c3 : notEnoughMoney/obs]\}\},\\
&\\
&\{\{[c1 : 2/dam], [c2 : 2/wam], [c2 : giveMoney(2)/obs]\},\\
&\quad \{[c1 : 3/dam], [c2 : 2/wam], [c2 : giveMoney(2)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\},\\
&\quad \{[c1 : 2/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\}\},\\
&\\
&\{\{[c1 : 3/dam], [c2 : 3/wam], [c2 : giveMoney(3)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 2/wam], [c3 : notEnoughMoney/obs]\}\},\\
&\\
&\{\{[c1 : 3/dam], [c2 : 3/wam], [c2 : giveMoney(3)/obs]\},\\
&\quad \{[c1 : 1/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\},\\
&\quad \{[c1 : 2/dam], [c3 : 3/wam], [c3 : notEnoughMoney/obs]\}\}\Big\}
\end{aligned}
$$

The *subUniform* condition implies the condition that, per chosen substitution set, the values for the variables affected by the predicate must have the same value if they are similarly annotated. In practice this means that we choose one value per behavior of the '*withdraw*' operation. As for the '*subUnWithdraw*' test intention, we present in figure 8.6 a possible reduced test set for the '*subUnWithdraw*' test intention, obtained by randomly choosing one substitution set from the previously calculated *reduced substitution set*.

```
HML( {deposit(d,2) with null} {withdraw(d,2), giveMoney(2)} T), True
HML( {deposit(d,3) with null} {withdraw(d,2), giveMoney(2)} T), True
HML( {deposit(d,1) with null} {withdraw(d,3), notEnoughMoney} T), True
HML( {deposit(d,2) with null} {withdraw(d,3), notEnoughMoney} T), True
```

Figure 8.6: $Reduced_{Banking}(subUnWithdraw)$

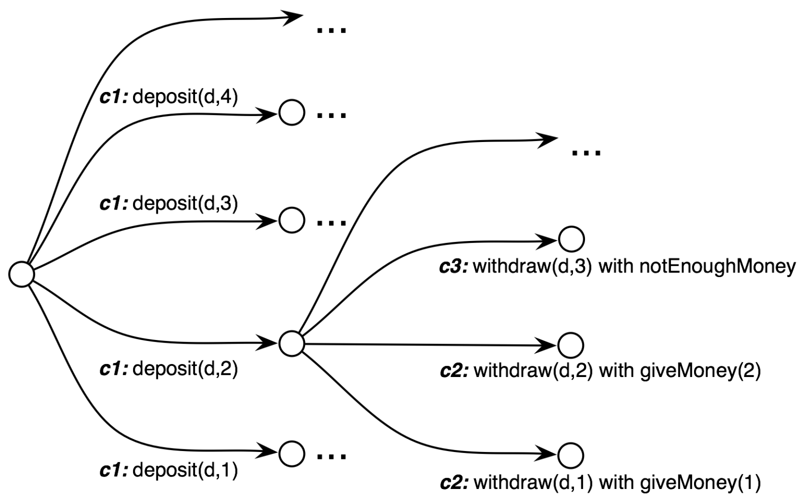

...
c1: deposit(d,4)
...
c1: deposit(d,3)
...
...
c3: withdraw(d,3) with notEnoughMoney
c1: deposit(d,2)
c2: withdraw(d,2) with giveMoney(2)
c1: deposit(d,1)
...
c2: withdraw(d,1) with giveMoney(1)

Figure 8.7: Sample Annotation of the Banking Server's Semantics

8.5 Test Set for a CO-OPN and SATEL Specification

We can now define the test set for a CO-OPN and SATEL specification. This corresponds to the union of all test sets generated for all *test intentions* in that specification.

Definition 8.5.1 *Test Set for a CO-OPN and SATEL Specification*

Let Spec be a CO-OPN and SATEL specification having a global SATEL signature $\Sigma^{Sat} = \langle S, \leq, F \rangle$ *and A be any finitely generated* Σ^{Sat}*-SATEL algebra. Let* *also* $Md_k^T = \langle \Gamma_k, Focus_k, \Lambda_k, X_k \rangle \in Spec$ $(1 \leq k \leq n)$ *be a set of* test intention modules. *The* test set *for Spec is calculated as follows:*

$$TestSet(Spec) = \bigcup_{i \in I} Reduced_{Spec}(i)$$

where $I = \bigcup_k \Gamma_k$ *is the set of all test intentions of Spec.*

8.6 Summary

In this chapter we have described the full semantics of the SATEL language. We have started by introducing the *global SATEL signature* which extends the previously defined *global signature* (see definition 4.4.4). The *global SATEL signature* includes not only the sorts defined by the *ADT modules* and induced by *Class modules*, but also the *Pat, Hml, Stim, Obsrv, Bool* and *Num* sorts which are the names of respectively the *execution pattern*, *Hml formulas*, *Stimulation*, *Observation*, *Boolean* and *Numeric* types introduced by the SATEL language. In particular since each *context module* specifies an SUT, the *Pat, Hml, Stim* and *Obsrv* types are *context module* specific.

We then define the notion of *SATEL Algebra* for a *global SATEL signature*. A *SATEL Algebra* is formed from the union of all the individual algebras for the sorts described in the *global SATEL signature*. The algebras for the *Pat, Hml, Stim, Obsrv, Bool* and *Num* sorts are models explicitly defined by us. In what concerns the algebras for sorts defined by the *ADT modules* and induced by *Class modules*, many models may exist, as long as they respect the equations in the algebraic specification of those types. The *SATEL Algebra* has then a fixed part corresponding to the SATEL types and a dynamic part corresponding to the remaining types.

In order to define the semantics of SATEL we start by building the *exhaustive test set* for a test intention. This construction is done in two steps: firstly we build the *expanded execution pattern set* for a given test intention. This set includes all *execution patterns* which may be generated from either the inclusion of a *test intention* within another *test intention* or by a *test intention* being recursively defined. In a second step we produce the *exhaustive test set* by uniting the *positive* and *negative exhaustive test sets*. Both *positive* and *negative exhaustive test sets* result from replacing the variables in the *expanded execution pattern set* by ground terms. The *positive exhaustive test set* is the set of execution patterns resulting from those substitutions whose interpretations in a given SATEL algebra are satisfied by the semantics of the focus *context module* of the considered test intention. Equally, the *negative exhaustive test set* is the set of execution patterns resulting from those substitutions whose interpretations in the same SATEL algebra are ***not*** satisfied by the semantics of the considered *context module*. Note that the *exhaustive test set* for a test intention does not take into consideration the conditions for the substitutions reflecting the hypothesis about the SUT — which means we produce a test set covering all the behavior expressed by the *test intention* without any reduction hypothesis. Finally we present an important *completeness* result which states that SATEL is sufficiently expressive to build the *exhaustive test set* for proving or disproving *bisimilarity* between the transition system semantics of a *context module* specification and an *SUT*.

We then proceed to define the *reduced test set* for a test intention. The *reduced test set* is a subset of the *exhaustive test set* where the substitutions for the variables in the *expanded execution pattern set* are constrained by the hypothesis expressed by the conditions associated to each *expanded execution pattern*. The reduction is performed at the level of the substitutions for each *expanded execution pattern*. In particular only the terms *satisfying* the conditions for the considered *expanded execution pattern* are kept.

The *uniform* and *subUniform* predicates are a particular kind of condition which require specialized treatment. The *uniform* predicate chooses one single substitution for a variable, while the *subUniform* predicate chooses one value per behavior of the *operation(s)* associated to that substitution. Both the *uniform* and the *subUniform* predicate can be applied to *Hml, Stimulation, Observation* or *operation parameter* variables. Enforcing a *subUniformity* constraint on an *Hml* variable means that the sequences of events resulting from the instantiation of that variable will cover all combinations of behaviors of the *operation(s)* involved in those sequences.

To produce substitution for variables on which a *uniformity* or *subUniformity* hypothesis is stated, we assume the transition system semantics of the focus *context module* for the considered test intention is annotated in such a way that each event (transition) is marked with all the *coordination* and *behavioral* formulas leading to its firing. Using a modified notion of satisfaction between *Hml formulas* and annotated transition systems we are able to retrieve a set of annotated substitutions which we use to perform the reductions imposed by the *uniform* and *subUniform* predicates. The reductions are achieved by collecting all possible substitutions for a given variable in an *expanded execution pattern* and then limiting the terms that can replace it to either one single value — in the case of a *uniformity* hypothesis — or several terms, one per possible behavior implied by that selection — in the case of a *subUniformity* hypothesis. The annotations in the terms used for the substitutions allow us to identify the substitutions that imply the same behavior. Finally, both the *uniformity* and the *subUniformity* predicates imply several possible reductions, one per choice for the value which represents the reduced domain. We solve this problem by *randomly* choosing one substitution inside the considered domain — given any of those substitution has the same *error finding* power regarding the hypothesis.

We finalize the chapter by defining the *test set for a CO-OPN and SATEL specification*. This test set corresponds to the union of all *reduced test sets* for all test intentions declared in *test intention modules* of that specification.

Chapter 9

Case Study

In this chapter we will describe the usage of SATEL in the context of testing a real-world system. The case study was performed in the context of an interregional (France–Switzerland) INTERREG III project having as participants the *University of Geneva*, the *Laboratoire d'Informatique de L'Université de Franche-Compté*, *LEIRIOS Technologies* and the *Centre des Technologies de l'Information* (CTI) of the state of Geneva. The project — named VALID — was developed throughout the year of 2006 and had as goal to address to quality of software by proposing innovative techniques for functional testing. A full report with the obtained results can be found in [62].

The presentation of the case study is divided in several steps. Firstly we will describe the SUT and its existing model in the UML. We will then describe the *test specification* we have produced in CO-OPN$_{/2c++}$ starting from the UML specification. We then proceed to defining *test intentions* for the *test specification*, while basing ourselves in the original UML model. We finish by a discussion of the obtained results.

9.1 The RTaxPM SUT and its Specification

Some years ago the state of Geneva has ordered the reconstruction of all the software dealing with the cantonal taxation. The example we will use as our case study consists of a subset of that software, namely the project dealing with the reimplementation of the software for *taxing moral persons* — called *RTaxPM*.

In order to tackle the problem of producing *test cases* for the already existing RTaxPM software we started by analyzing the existing specification, written in the UML. The model was well described, including *data types*, UML *modeling conven-*

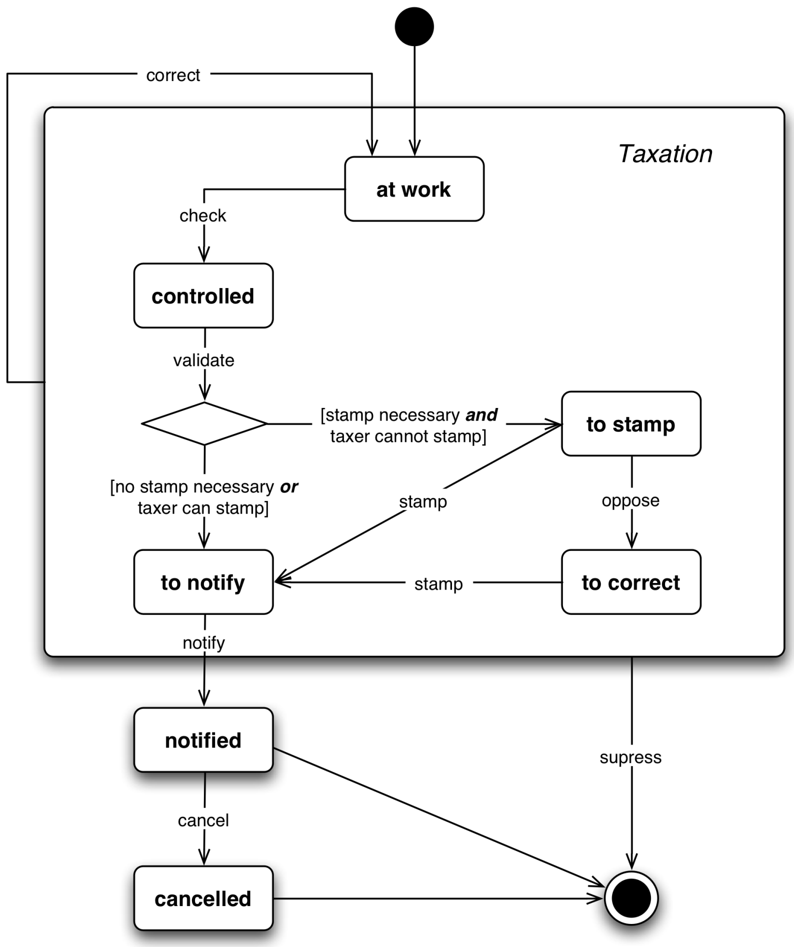


Figure 9.1: The life cycle of a Taxation

tions and conventions specific to the RTaxPM project itself. Although the software implements several functions which require separate testing, we concentrated our efforts on testing the *life cycle of a taxation*. This subset is sufficiently significant to enable our case study as it involves a central critical aspect of taxation process.

Figure 9.1 presents the *activity diagram* model of the *life cycle of a taxation*. The model represents the several states a *dossier* holding the information about a moral person goes through until the *taxation* activity finishes. Note that all state transitions are manually performed by a human taxer using a graphical interface. The taxation function works as follows:

- firstly the ***at work*** state indicates that a *dossier* has entered processing;

- a basic *check*[1] is then performed on the *dossier* which, if successful makes it enter a ***controlled*** state;
- *validation* is then required to enter the ***to notify*** state. This *validation* can be done either by *simple taxer* user in case no *stamp* is necessary, or by a *taxer who can stamp* in case *stamping* is necessary. In case stamping is necessary but the taxer handling the *dossier* does not have sufficient privilege to perform the operation, the *dossier* enters an intermediate state ***to stamp***. A taxer with sufficient privileges can then either perform the stamping, which allows the *dossier* to enter the ***to notify*** state or oppose the previous *validation* which makes the *dossier* enter a ***to correct*** state. From the ***to correct*** state it is still possible to rejoin the normal flow if the required stamping is done. Otherwise the only possibility is to *correct* the *dossier* which means returning to the ***at work*** state;
- it is then necessary to *notify* the moral person being taxed, which makes the *dossier* enter the ***notified*** state;
- after notification it is still possible to *cancel* the taxation.

Note that from the *Taxation* superstate it is possible to either *correct* or *supress* the taxation at any moment.

9.2 The CO-OPN Test Specification

In order to produce test cases for the *life cycle of a taxation* we started by producing a CO-OPN$_{/2c++}$ specification reflecting the UML specification. While performing this activity we encountered the problem that the system as described in section 9.1 is under-specified. In particular:

- the notions of *dossier* and *stamp* are not described in the statechart of figure 9.1. In particular a *stamp* is required to perform a taxation if the *dossier* obeys certain criteria. These criteria are business rules informally or semi-formally described elsewhere in the RTaxPM UML specification;
- the UML specification states that some taxers can *stamp* a *dossier* and others cannot. However, no particular information about *user* types is given;
- the conditions describing the guards of the transitions are described in natural language.

[1]The models abstracts from the actual conditions checked.

We were then obliged to find the correct abstractions in order to be able to build the CO-OPN model. This was achieved by discussing with the development team and by observing the already existing SUT. We have decided to describe the problem of a *dossier* needing a *stamp* by coding it directly as a *boolean* in the data type describing a *dossier*. We took a similar approach to define the *taxer* types: a *taxer* can either *stamp* or not, also described by a *boolean*. The definition of the data types *taxer* and *dossier* are shown in figures 9.2 and 9.3 respectively. The full CO-OPN specification for the RTaxPM SUT can be found in appendix C.

```
ADT TaxerPM;

Interface

    Use
        Char;
        Booleans;

    Sort
        taxer;

    Generator
        newTaxer _ allowedToStamp _ : char boolean -> taxer;

End TaxerPM;
```

Figure 9.2: The TaxerPM ADT

```
ADT Dossier;

Interface

    Use
        Char;
        Booleans;

    Sort
        dossier;

    Generator
        newDossier _ stampNeeded _ : char boolean -> dossier;

End Dossier;
```

Figure 9.3: The *Dossier* ADT

In order to model the activity described by the statechart itself we resorted to the usage of a Petri Net where *places* represent *statechart* states. *Statechart* transitions are naturally represented by Petri Net transitions. In particular we have chosen to specify the statechart for the *lifecycle of a taxation* as a CO-OPN class where each place of the classe's petri net can hold one or more *dossiers* under treatment. The fact that several *dossiers* can be held in the statechart is not strictly necessary, but we keep it to illustrate the capability of CO-OPN to serve as model in a more complex test generation scenario where a *taxer* can be simultaneously handling several *dossiers*.

The specification for the *TaxationEngine* class is partially shown in figure 9.4. Note that to address the problem of the guards being defined in natural language we have coded them in boolean algebra. The conditions be observed in lines 37 and 41 of figure 9.4 and are simple given that we have included in the specification the abstractions that both the need for a *dossier* to be stamped and the capacity for a *taxer* to stamp can be expressed as booleans. Note also that two axioms are used to specify the *createDossier* axiom which starts the process by transitioning from the *startState* to the *atWork* state. The only difference between the axioms is that one includes a '*dossierClearance = true*' condition while the other includes a '*dossier-Clearance = false*' condition. Although the behavior is more verbose than necessary, we will see during the present chapter that this over-specification is relevant for test generation using SATEL.

Due to the fact that an implementation of the *lifecycle of a taxation* was already existing when we started the collaboration with the CTI, we were obliged to take into consideration some operational issues while generating test cases. In particular, the *test driver* could only act as client for the database server managing the *dossier* information. The client consists of an API (Application Programming Interface) which is also used by a graphical interface in the *taxer's* local machine. This API has the restriction that only one *taxer* can be logged in at a time from a given machine. Given that we were only granted access to one machine at a time to launch our *test cases* we were forced to model the *logging* mechanism in our specification.

Figure 9.5 partially presents the CO-OPN textual specification of the *Login-Manager* class. The class includes two places '*loggedUsers*' and '*unLoggedUsers*' which some taxers, some of which having the privilege to stamp and others not having it. A couple of auxiliary places '*taxerLogged*' and '*noTaxerLogged*' allow preventing having more than one user logged in at a time. Finally, notice that, as for the '*TaxationEngine*' Class, the '*loginTaxer*' method is over-specified. It also includes a behavior dealing with the login when a *taxer* is already logged in. In this case the '*alreadyLogged*' gate is activated thus issuing an error message to the outside of the object.

In order to integrate both the *TaxationEngine* and the *LoginManager* classes we used a *context module* which is represented in its graphical syntax in figure 9.6. Notice that the context holds two objects, one belonging to each class. The objects communicate by synchronizing the '*isLogged*' gate of the *taxationEngine* object with

```
    Gate
        isLogged _ : taxer;

    Methods
        createDossier _ : dossier;
        check;
        validate;

Body

    Places
        startState _ : blackToken;
        atWork _ : dossier;
        controlled _ : dossier;
        toStamp _ : dossier;
        toCorrect _ : dossier;
        toNotify _ : dossier;
        notified _ : dossier;
        canceled _ : dossier;
        endState _ : blackToken;

    Initial
        startState @;

    Axioms

        dossierClearance = true =>
                createDossier newDossier aDossierN stampNeeded dossierClearance
                    With isLogged aTaxer::
                startState @ -> atWork aDossier;

        dossierClearance = false =>
                createDossier newDossier aDossierN stampNeeded dossierClearance
                    With isLogged aTaxer::
                startState @ -> atWork aDossier;

        check With isLogged aTaxer::
                atWork aDossier -> controlled aDossier;

        ((userClearance = false) and (dossierClearance = true)) = true =>
                validate With isLogged newTaxer aTaxerN allowedToStamp
                    userClearance::
                controlled newDossier aDossierN stampNeeded dossierClearance ->
                    toStamp newDossier aDossierN stampNeeded dossierClearance;

        ((userClearance = true) or (dossierClearance = false)) = true =>
                validate With isLogged newTaxer aTaxerN allowedToStamp
                    userClearance::
                controlled newDossier aDossierN stampNeeded dossierClearance ->
                    toNotify newDossier aDossierN stampNeeded dossierClearance;

    Where
        aTaxerN, aDossierN : char;
        aTaxer : taxer;
        aDossier : dossier;
        userClearance, dossierClearance : boolean;
```

Figure 9.4: The *TaxationEngine* Class (partial view)

```
    Gate
        alreadyLogged;

    Methods
        loginTaxer _ : char;
        logoutTaxer;
        isLogged _ : taxer;

Body

    Places
        taxerLogged _ : blackToken;
        noTaxerLogged _ : blackToken;
        loggedUsers _ : taxer;
        unLoggedUsers _ : taxer;

    Initial
        unLoggedUsers newTaxer a allowedToStamp false, unLoggedUsers newTaxer b
            allowedToStamp false, unLoggedUsers newTaxer l allowedToStamp true,
            unLoggedUsers newTaxer m allowedToStamp true, noTaxerLogged @;

    Axioms
        userClearance = true =>
             loginTaxer aTaxerN::
             unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                 noTaxerLogged @ -> loggedUsers newTaxer aTaxerN allowedToStamp
                 userClearance, taxerLogged @;

        userClearance = false =>
            loginTaxer aTaxerN::
            unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                noTaxerLogged @ -> loggedUsers newTaxer aTaxerN allowedToStamp
                userClearance, taxerLogged @;

        loginTaxer aTaxerN With alreadyLogged::
             unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                 taxerLogged @ -> unLoggedUsers newTaxer aTaxerN allowedToStamp
                 userClearance, taxerLogged @;

        logoutTaxer::
             loggedUsers newTaxer aTaxerN allowedToStamp userClearance, taxerLogged
                  @ -> unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                  noTaxerLogged @;

        isLogged aTaxer::
             loggedUsers aTaxer -> loggedUsers aTaxer;

    Where
        aTaxerN : char;
        aTaxer : taxer;
        userClearance : boolean;
```

Figure 9.5: The *LoginManager* Class (partial view)

the '*isLogged*' method of the *loginMgr* object — thus allowing the *taxationEngine* object to "ask" the *loginMgr* object if a taxer is logged in. Note also that '*alreadyLogged*' gate of the *LoginManager* class is connected to the outside of the

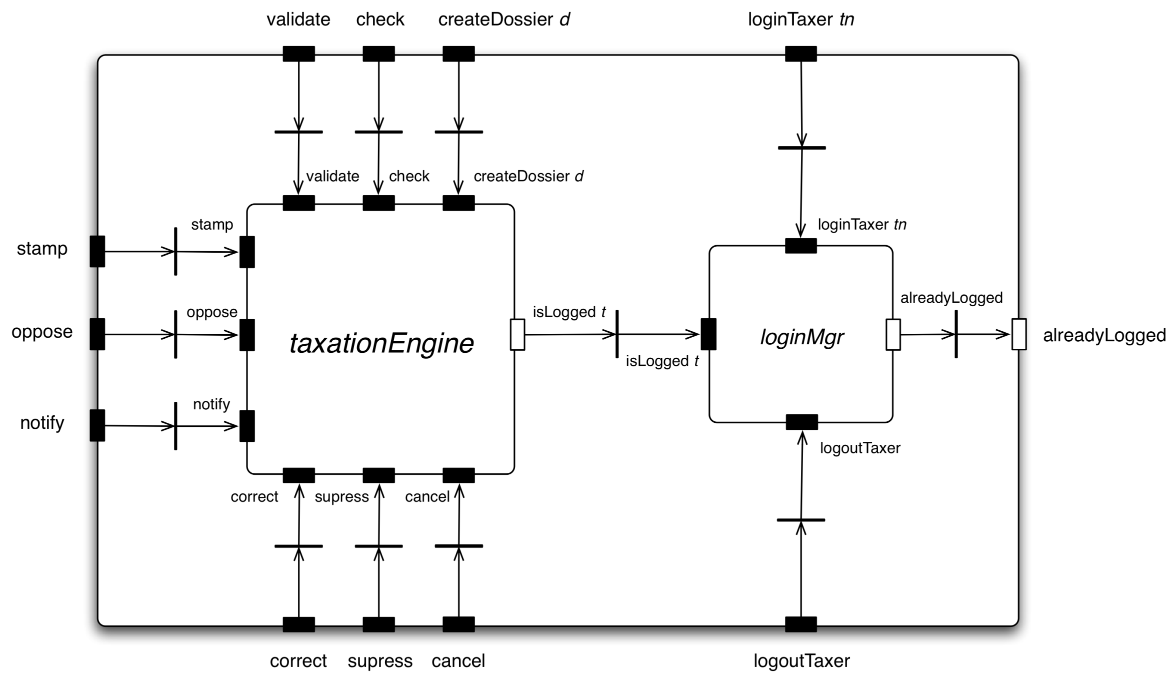


Figure 9.6: The TaxationInterface Context

specification allowing the propagation of the error message to the environment.

9.3 Test Intentions and Generated Test Cases

In this section we will present SATEL from a *test engineer* point of view, by stating test intentions in order to generate test cases for the RTaxPM SUT. We will split the test generation activity in two phases: in the first phase we will generate test cases for *unit testing* of the *LoginManager* class; in the second phase we will generate *test cases* for *integration testing* of the whole system. We will also introduce some test sets generated from the test intentions. Note that, for clarity reasons, the test sets we present in the examples of this chapter will only include *positive* test cases — i.e. those that correspond to valid behaviors of the context module which is the *focus* of the test intention.

9.3.1 Unit Testing

The SATEL formalism, as defined in chapters 7 and 8, allows writing *test intention* modules having as target (or *focus*) a particular *context module* — it is however not possible to write *test intentions* having as focus a CO-OPN class. In order to overcome this issue we have written a *LoginMgrInterface context module* (see appendix C) which encapsulates an instance of the *LoginManager* class. The *methods* and *gates* of the instance of the *LoginManager* class are synchronized with *methods* and *gates* of the same name of the *LoginMgrInterface context module*.

```
TestIntentionSet UnitTesting Focus LoginMgrInterface;

Interface

    Intentions
        loginLogout;
        onlyOneUserLogged;

Body

    Intentions
        repeatLoginLogout;

    Uses
        TaxerPM;
        Dossier;

    Axioms
        HML ( T )  in repeatLoginLogout;

        uniformity (t) | f in repeatLoginLogout =>
           HML ( { loginTaxer (t) with null }
           { logoutTaxer with null } T ) . f  in repeatLoginLogout;

        | f in repeatLoginLogout, ( nbEvent(f) / 2 ) <= 3, nbEvent(f) > 0 => f in
            loginLogout;

        uniformity (t), uniformity (u) | => HML ( { loginTaxer (t) with null }
           { loginTaxer (u) with alreadyLogged } T ) in onlyOneUserLogged;

    Variables
        t,u : taxer;
        d : dossier;
        f : primitiveHML;

End UnitTesting;
```

Figure 9.7: The *LoginManager* Class

In figure 9.7 we present the *test intention* module '*UnitTesting*' that generates *test cases* for the unit testing of the *LoginManager* class. Note that the *focus* of the test intention module is the *LoginMgrInterface context* module that encapsulates an instance of the *LoginManager* class.

The test intention module in figure 9.7 defines three test intentions: the '*repeatLoginLogout*' test intention is an auxiliary[2] test intention defining a sequence of

[2]Auxiliary test intentions are defined in the *body* of the *test intention module* and do not directly produce *test cases* — they are used as building blocks for other *test intentions*.

login/logout operations. It is not practicable to directly use '*repeatLoginLogout*' as this test intention is recursively defined (in lines 18-22 of figure 9.7) and produces an infinite amount of *test cases* with any number of login/logout pairs.

The '*loginLogout*' test intention defined in line 24 of figure 9.7 uses the '*repeat-LoginLogout*' test intention to generate three sequences of login/logout pairs. Note that the '$(nbEvent(f)/2) <= 3$, $nbEvent(f) > 0$' conditions forces the number of events of the generated *test cases* to be pair, inferior to 6 and larger than 0 — in this fashion we stating we want at most 3 login/logout pairs, but not the empty[3] test case. Another interesting fact about the '*loginLogout*' test intention is that each instantiation of the '*t*' variable in the '*loginTaxer (t)*' method involves a uniformity hypothesis on the possible instantiations of '*t*'. That means each renaming of variable '*t*' for a '*loginTaxer (t)*' method inside the same *expanded execution pattern set* (see definition 8.3.1) for test intention '*loginLogout*' can be instantiated only once while generating test cases. An example of a possible *test set* generated by the '*loginLogout*' test intention is shown in figure 9.8.

```
HML( { loginTaxer (a) with null } { logoutTaxer with null } T ), true
HML( { loginTaxer (b) with null } { logoutTaxer with null } { loginTaxer (l) with
    null } { logoutTaxer with null } T ), true
HML( { loginTaxer (m) with null } { logoutTaxer with null } { loginTaxer (a) with
    null } { logoutTaxer with null } { loginTaxer (a) with null } { logoutTaxer
    with null } T ), true
```

Figure 9.8: Possible test set generated by the '*loginLogout*' test intention

Finally, the '*onlyOneUserLogged*' test intention in line 26 of figure 9.7 produces a test set for making sure the *alreadyLogged* gate call is performed when a *taxer* is already logged in and another *taxer* tries logging. Again, the uniformity hypothesis on variables '*t*' and '*u*' allows restricting the instantiations of these variables to a single taxer name. An possible test set for the '*onlyOneUserLogged*' test intention — holding one single test case — is presented in figure 9.9.

```
HML( { loginTaxer (a) with null } { loginTaxer (b) with alreadyLogged } T ), true
```

Figure 9.9: Possible test set generated by the '*loginLogout*' test intention

9.3.2 Integration Testing

Let us now present a set of test intentions for generating test cases that allow *integration testing* of the RTaxPM SUT. We will use as *focus* the *TaxationInterface*

[3]HML (T)

context module we have presented in figure 9.6 (the textual syntax can be found in appendix C). The *TaxationInterface* context module allows producing test cases for *integration testing* as in fact it coordinates the activities of an object of type *TaxationEngine* and an object of type *TaxationEngine*. By chance it happens that the *TaxationInterface* context module is also the *topmost* context in this CO-OPN specification, which means we are also generating test cases for *system testing*.

We will proceed in several steps to illustrate the multiple features of SATEL. Let us start by writing an *Auxiliary* test intention module containing a library of test intention building blocks — which will be later used to define complete test intentions.

```
TestIntentionSet Auxiliary Focus TaxationInterface;

Interface
    Intentions
        prefix;
        postfix;
        repeatCorrection;

Body

    Uses
        TaxerPM;
        Dossier;

    Axioms
        uniformity (t), uniformity (d) | => HML ( { loginTaxer (t) with null } {
            addDossier (d) with null } T ) in prefix;

        HML ( { notify with null } { cancel with null } T ) in postfix;

        HML ( T ) in repeatCorrection;

        | f in repeatCorrection => f . HML ( { check with null } { validate with
            null } { correct with null } T ) in repeatCorrection;

    Variables
        t : taxer;
        d : dossier;
        f : primitiveHML;

End Auxiliary;
```

Figure 9.10: The *Auxiliary* Test Intention module

The *Auxiliary* test intention module is presented in figure 9.10 and includes the definition of three test intention: the *prefix* test intention defines a prelude for test cases including a login of a *taxer* and the adding of a *dossier* to the *taxation* process; the *postfix* test intention defines a couple of operations that, if reached, indicate the end of the *taxation* process; the *repeatCorrection* test intention which defines a sequence of any size of *check*, *validate* and *correct* operations.

Model Coverage

Let us now introduce the complete test intentions that generate test cases for the *TaxationInterface* context module. The test intention module *IntegrationTesting* is introduced in figure 9.11 and defines three test intentions: *coverValidation*, *reachEnd* and *multipleCorrections*.

```
 0 TestIntentionSet IntegrationTesting Focus TaxationInterface;

 2 Interface

 4      Intentions
            coverValidation;
 6          finishFlow;
            multipleCorrections;
 8
   Body
10
        Uses
12          TaxerPM;
            Dossier;
14          Auxiliary;

16      Axioms
            subUniformity(t), subUniformity(d) | => HML ( { loginTaxer (t) with null }
                { addDossier (d) with null } { check with null } { validate with null }
                 T ) in coverValidation;
18
            | f in prefix, h in postfix, nbEvent(g) < 10 => f . g . h in finishFlow;
20
            | f in prefix , g in repeatCorrection , (nbEvent(g) / 3) <= 2 => f . g in
                multipleCorrections;
22
        Variables
24          t : taxer;
            d : dossier;
26          f, g, h : primitiveHML;

28 End IntegrationTesting;
```

Figure 9.11: The *Auxiliary* Test Intention module

The *coverValidation* test intention is the most relevant of the set as it allows testing the *validation* method which is the most complex behavior of the SUT — as can be observed in its specification in figure 9.1. In order to test the *validation* method we have decided to apply the *subUniformity* predicate on the parameters of the '*loginTaxer (t)*' and the '*addDossier (u)*' methods as can be seen in line 17 of figure 9.11. The *subUniformity* predicate will choose one single instantiation of variables '*t*' and '*u*' per behavior of methods *loginTaxer* and *addDossier* respectively.

As can be seen in lines 20 to 29 of figure 9.5 the *loginTaxer* method has three behaviors: the taxer is allowed to stamp; the taxer is not allowed to stamp; a taxer is already logged. As we have previously explained the two first behaviors are somewhat artificial and were devised for the purpose of test generation. The *subUniformity* predicate applied to variable '*t*' in the axiom defining the *coverValidation* test intention will then select two values for '*t*': one allowing the login of a *taxer* that ***can*** stamp; another allowing the login of a *taxer* that ***cannot*** stamp. Notice that in this case the third behavior — a taxer is already logged — is not possible as in the initial state of an object of type *LoginManager* no *taxer* is logged (see line 17 of figure 9.7). The same reasoning as described in this paragraph can be applied to the *addDossier* method of class *TaxationEngine* defined in lines 26 to 32 of figure 9.4.

A possible *test set* generated by the *coverValidation* test intention is presented in figure 9.12. Notice that all combinations of a *taxer allowed* or not *allowed* to stamp and *dossier requiring* or not *requiring* a stamp are tried out. Also, due to the fashion in which the formal semantics of SATEL are defined (see section 8.4.8) the variables covered by the *subUniformity* — as well as by the *uniformity* — predicate are instantiated to the same values in all test cases produced by an expanded execution pattern (see definition 8.3.1). The two chosen values for variables '*t*' and '*d*' are thus the same in all the test cases of figure 9.12.

```
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded true) }
    { check with null } { validate with null } T ), true
HML( { loginTaxer (l) with null } { addDossier (newDossier e stampNeeded false) }
      { check with null } { validate with null } T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier e stampNeeded false) }
      { check with null } { validate with null } T ), true
HML( { loginTaxer (l) with null } { addDossier (newDossier d stampNeeded true) }
    { check with null } { validate with null } T ), true
```

Figure 9.12: Possible test set generated by the '*coverValidation*' test intention

The *finishFlow* test intention defined in line 19 of figure 9.11 makes use of the *prefix* and the *postfix* test intentions defined in the *Auxiliary* test intention module to generate test cases that always terminate by a '*notify*' method call followed by '*cancel*' method call. The '*g*' variable can be expanded to any HML formula involving method and gate calls of the *TaxationInterface* context, having a number of events inferior to 10. This test intention would produce for example the (non-exhaustive) test set displayed in figure 9.13.

It is interesting to note that, again due to the semantics of SATEL, the instantiation of the variables '*t*' and '*d*' from the imported test intention *prefix* (see figure 9.10) is the same for all generated test cases. In fact, depending on the random choice of the values for those variables by the *uniformity* predicate the generated test sets may be very diverse.

Finally, the *multipleCorrections* test intention defined in line 21 of figure 9.11 uses the auxiliary *repeatCorrection* test intention defined in the *Auxiliary* test in-

```
0 HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false) }
      { check with null } { validate with null } { notify with null } { cancel with
      null } T ), true
  HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false) }
      { check with null } { loginTaxer (b) with alreadyLogged } { validate with null
      } { notify with null } { cancel with null } T ), true
2 HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false) }
      { check with null } { correct with null } { check with null } { validate with
      null } { notify with null } { cancel with null } T ), true
```

Figure 9.13: Possible (non-exhaustive) test set generated by the '*finishFlow*' test intention

tention module to perform multiple correction cycles. We have limited the number of cycles to 2 in the *multipleCorrections* test intention which means a test set such as the one displayed in figure 9.14 is produced.

```
0 HML( { loginTaxer (m) with null } { addDossier (newDossier d stampNeeded false)
      with null } T ), true
  HML( { loginTaxer (m) with null } { addDossier (newDossier d stampNeeded false)
      with null } { check with null } { validate with null } { correct with null }
      T ), true
2 HML( { loginTaxer (m) with null } { addDossier (newDossier d stampNeeded false)
      with null } { check with null } { validate with null } { correct with null }
      { check with null } { validate with null } { notify with null } { correct with
      null } T ), true
```

Figure 9.14: Possible test set generated by the '*multipleCorrections*' test intention

Extending the Model for Observation

Although figures 9.8, 9.9, 9.12, 9.13 describe adequate test sets, the lack of observations in those test sets — note that almost all the stimulations have a '*with null*' counterpart — pose technical difficulties at the level of the test driver. While our colleagues from LEIRIOS Technologies were building the test driver it became apparent that in order to efficiently calculate verdicts for the test cases a more reactive SUT was needed. The SUT was then instrumented to output the name of the statechart *state name* at each transition. The instrumentation was performed only at the level of connecting the SUT's API to the test driver as the necessary information was already being produced by the SUT.

This instrumentation of the SUT required introducing some modifications at the level of our CO-OPN model. In particular we have introduced new *gate* ports in the *TaxationEngine* class and the *TaxationInterface* context which propagate a state reached by a given statechart transition. The axioms of the *TaxationEngine* also needed to be extended in order to activate the newly introduced *gate* ports. The extension consists of synchronizing each method of the *TaxationEngine* inducing a state transition with the respective observation gate. An example of this extension for the '*check*' method of the *TaxationEngine* class is presented in figure 9.15. In order for the CO-OPN test specification of the RTaxPM SUT to output the observations to the environment it is also necessary to: add observation *gate* ports to the *TaxationInterface* context module; coordinate them with the *taxationEngine*'s observation gates. The full extended specification is presented in appendix C.

```
    Gate
        reachedControlled

Body

    Places
        atWork _ : dossier;
        controlled _ : dossier;

    Axioms

        check With isLogged aTaxer // reachedControlled::
                atWork aDossier -> controlled aDossier;
```

Figure 9.15: Extending the *TaxationEngine* for observation

With the extended specification it is now possible to rewrite the test intentions for integration testing (figures 9.10 and 9.11) to include the observations we have added to the CO-OPN test model. We replicate in figure 9.16 the *postfix* test intention we have introduced in figure 9.10 and its extended version with observations in figure 9.17.

```
HML ( { notify with null } { cancel with null } T ) in postfix;
```

Figure 9.16: The *Postfix* Test Intention

```
HML ( { notify with reachNotify } { cancel with reachCancelled } T ) in postfix;
```

Figure 9.17: The *Postfix* Test Intention extended with observations

Robustness Testing

SATEL allows the generation of test cases for behaviors that should not be allowed by the SUT. Given that it is possible to expressing negative behaviors in the HML formalism, we shall use this feature to generate test cases that ensure a wrong state of the statechart is not reached. In order to do this we shall take advantage of the extended CO-OPN specification of the RTaxPM SUT we have introduced in section 9.3.2.

```
    Axioms
        uniformity(f), uniformity(stm ) |
        f in reachState, nbEvent(f) < 10, nbSyncro(stm) == 1, nbSyncro(obs) == 1 =>
            f . HML ( {stm with reachedToNotify} not ( {notify with obs} T ) T ) in
                robustNotify;

    Variables
        f : primitiveHML;
        stm : primitiveStimulation;
        obs : primitiveObservation;
```

Figure 9.18: Test Intentions for Robustness Testing

In figure 9.18 we present a *test intention* for robustness testing of the transitions leaving the *toNotify* state. The *robustCheck* test intention is composed of three parts: the '*f*' variable of type *primitiveHML* is used to find a path until a state that precedes the '*toNotify*' state. The '{*stm with reachedToNotify*}' event is then used to reach the '*toNotify*' state. Note that '*stm*' is a *primitiveStimulation* variable which will be instantiated to one method call which will have as observation the '*reachedToNotify*' gate. Finally, the '*not ({notify with obs} T)*' interior negative HML formula allows making sure that the '*obs*' variable is instantiated to an observation that is not possible. Uniformity hypothesis are made both on variables '*f*' and '*stm*' due to the fact that we are interested in testing the transition exiting the '*toNotify*' state, rather than the path leading to it. Also, the '$nbSyncro(stm) == 1, nbSyncro(obs) == 1$' conditions enforce that no complex synchronizations are instantiated for variables '*stm*' and '*obs*'. A possible (non-exhaustive) set of test cases generated by this test intention can be found in figure 9.19.

It is interesting to notice that the pattern we have defined with the *robustNotify* test intention in figure 9.18 can be reused with minor changes in order to generate test cases for robustness testing of the transitions exiting any state of the statechart presented in figure 9.1.

9.4 Discussion

A test driver was built for RTaxPM project and a test set built by our colleagues from LEIRIOS and the Université de Franche-Comté was applied to the SUT. As is presented in the final report for the project [62], the testing resulted in finding

```
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedAtWork} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedControlled} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedToStamp} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedToCorrect} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedToNotify} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedCancelled} T ) T ), true
HML( { loginTaxer (b) with null } { addDossier (newDossier d stampNeeded false)
    with reachedAtWork} { check with reachedControlled} { validate with
    reachedToNotify} not({notify with reachedEnd} T ) T ), true
```

Figure 9.19: Possible (non-exhaustive) test set generated by the '*robustNotify*' test intention

an unimplemented functionality at the level of the taxer allowed to stamp as well as a problem at the level of the LEIRIOS test specification. No errors were found at the level of the implementation, which can be explained by relatively simple functionality of the system.

We validated the CO-OPN test model built for the project with the CTI members and presented some tests generated by our test intentions — although we did not run them using the LEIRIOS test driver. One problem we would have nonetheless found would have been the issue of having negative oracles in our test cases, which usage was not previewed by the test driver.

The RTaxPM project provided us with a testbed in which we have explored some methodological issues in the usage of SATEL. We describe these issues in the following sections.

9.4.1 Test Intentions vs Equivalence Class Testing

Although we have not discussed this subject in section 9.3, it is important to mention that we can produce interesting test cases for any CO-OPN specification using a very simple test intention. In fact the application of the *subUniformity* predicate to an HML formula will calculate the equivalence classes for each event of each instantiation of that HML formula. This means that a test intention such as the one presented in figure 9.20 would be enough to cover all the paths of any given

CO-OPN specification, while choosing only one event per equivalence class (see definition 8.4.8) at each moment of the formula.

```
Axioms
    subUniformity(f) => f in equivalenceClassIntention
Variables
    f : primitiveHML;
```

Figure 9.20: Generic Test Intention for Equivalence Class Testing

The kind of coverage induced by the '*equivalenceClassIntention*' test intention we present in figure 9.20 would then correspond to a structural condition coverage of the specification by generating sequences of events representing valid behaviors and that, at each step, exercise all the possible combinations of axiom conditions allowing the firing of a given event.

It then becomes legitimate to ask why building test intentions, if a simple test intention such as the one presented in figure 9.20 is enough to cover all behaviors of the specification. The answer is manifold:

- *test intentions* allow being precise in the specification of the behavior to test, which is important in order to produce test cases rapidly understandable by the test engineer and where the discovered errors are traceable back to the specification;

- *test intentions* allow explicitly limiting repetitive behaviors which could produce arbitrarily large test cases — as was demonstrated by the '*multipleCorrections*' test intention in figure 9.11;

- due to the possibility of defining recursive *test intentions*, it is possible to specify *test intentions* including pattern repetition, which is similar to the Kleene closure. This can be observed in the '*repeatCorrection*' test intention in figure 9.10;

- *test intentions* allow splitting the calculation of a test case in several parts, using for example *uniformity* hypothesis where the functionality to test is less important and *equivalence class* or *regularity* hypothesis (or simple *exhaustivity*) where the functionality to test is more important. An example of this kind of division can be observed in the '*finishFlow*' test intention in figure 9.11. Another interesting aspect of splitting a test intention in several parts is the possibility of building a postfix for test cases which would reinitialize the SUT, in order for the test driver to apply test cases of a test set in a sequential manner;

- *test intentions* can serve as heuristics in order to operationally accelerate the calculation of test cases. Although we do not propose in this thesis an operational method for calculating test cases, it is clear that the exhaustive instantiation of the variables present in a test intention is very costly — or *impossible* in the case of infinite domains — and that directing the test generation algorithm by the usage of test intentions is a possibility in order to reduce search complexity.

9.4.2 Positive and Negative Test Cases

As we have explained in chapter 8, test intentions generate both *positive negative* test cases — depending on whether the transition system semantics of a CO-OPN specification *satisfies* or *does not satisfy* (see definition 8.3.3) the HML formulas resulting from the instantiation of the variables present on those test intentions. In the examples presented in section 9.3 we have only shown the positive test cases, i.e. the HML formulas representing valid behaviors of the SUT. In fact, most of the *test intentions* discussed in section 9.3 produce negative test cases resulting from the exhaustive instantiations of the variables present in a test intention. These negative test cases, although theoretically important to build the exhaustive test set for CO-OPN specifications (regarding the *bisimulation* relation in definition 6.2.1), are most of the time not interesting for the testing activity. They represent behaviors which are not only difficult to understand by a test engineer, but also difficult to trace back to the specification in order to find where an error occurred if one was found by the *test driver*.

We have thus chosen to treat the negative cases in a positive fashion. This can be observed in the '*robustNotify*' test intention in figure 9.18, where we use the *not* predicate of HML temporal logic in order to generate *positive* test cases including a negative part — in this case the last event which should not be possible since it leads to an impossible state. Note that this technique allows keeping control of the positive part of the testing intention, while limiting the negative part by the *not* HML predicate.

9.4.3 Test Intention Modules

In section 9.3 we have presented both *Unit* and *Integration* — or, in the present case, also *System* — testing by using the same methodology: firstly we have isolated the part of the SUT we wanted to test by surrounding it with a *context* module; secondly we have connected the interesting *methods* and *gates* of the subsystem specification to *methods* and *gates* in the surrounding *context modules*; thirdly we wrote *test* *intentions* for the surrounding *context* module. In order to perform *System* testing one can always use the topmost context as the *focus* of the test intention module. This methodology seems practical and one could even devise an automatic fashion of producing the surrounding *context* modules.

It is however to be noticed that while performing *unit testing* by surrounding an instance of a CO-OPN class with a context module we do not guarantee that we are really generating unit tests for the SUT. In order for this to happen it would be necessary that a one-to-one mapping always exists between CO-OPN classes and SUT (implementation) classes, which may or may not be the case.

Another issue which arose while writing the *test intentions* for the RTaxPM SUT lies in the fact that it would be interesting to augment SATEL with the possibility of having parametric *test intentions*. For example the '*robustNotify*' test intention in figure 9.18 could be made more generic if we could build a '*reachState(s)*' test intention that reaches any given state '*s*' provided as parameter. We could then use that test intention as a prefix in order to reach the state we wish to perform the robustness testing upon.

9.5 Summary

In this chapter we have presented a realistic case study having as goal the generation of test cases for a taxing application (RTaxPM), developed for the state of Geneva. We have started by building a CO-OPN *test specification* which is sufficiently precise to generate oracles for test intentions expressed using SATEL. In order to build this *test specification* we were obliged to find the correct level of abstraction at which we could express the concepts of the taxing application in CO-OPN. This exercise allowed us to confirm that CO-OPN's modeling mechanisms, notably *ADT*, *Class* and *Context* modules are sufficiently expressive to generate precise and elegant abstractions for concepts of the real world.

We have then used SATEL to write test intentions for the RTaxPM CO-OPN test specification. This was done in several steps, starting with *unit testing* for a particular class of the test specification and then proceeding to *integration testing* of the whole system. Writing these test intentions allowed us to explore the many features of SATEL, namely the: *modularity* and the *reuse* of test intentions; the easy expression of *uniformity* and *regularity* hypothesis by using the available predicates to constraint SATEL variables; the usage of the *subUniformity* predicate for the calculation of tests which automatically subdivide the behaviors of operations expressed in CO-OPN and choose one event per equivalence class of those behaviors In particular this experience provided us with the opportunity of experimenting with the SATEL editor also developed in the context of this thesis [37, 38].

During this chapter we also approached some methodological issues while building test intentions using SATEL. We have discussed the usage of *context modules* to encapsulate parts of the specification in order to produce test sets for *unit*, *integration* or system testing. Robustness testing was also explored and we have informally introduced a technique for generating *positive* test cases while expressing *negative* behaviors at certain parts of a test intention.

Finally we have discussed the interest of using test intentions as opposed to simply performing an exploration of the specification by automatically producing one event per equivalence class of each operation present in the CO-OPN specification — i.e. by applying a *subUniformity* predicate to a free variable of type '*primitiveHML*'. Although we have shown that test intentions allow clear and precise test generation, it would seem that operational experimentation with larger systems is needed to further our results.

Chapter 10

Conclusion

In this chapter we will conclude and discuss the results of our work. We will start by discussing in detail our contributions and establishing how we have improved the state of the art in what concerns model based testing from CO-OPN specifications. We will then discuss these results, establish their importance and extrapolate their usefulness in the context of model based testing in general. Finally we will identify issues which were not addressed with our work and propose a set of directions for the continuation of the research.

10.1 Contributions

The contributions of this thesis can be stated at three levels: bug correction, clarification and improvement of the syntax and semantics of the CO-OPN language; proposal of the SATEL language including the new concept of *test intention*; complete formalization of a sub-domain decomposition method using a structural criterion based on CO-OPN axioms describing the behavior necessary for event firing.

10.1.1 CO-OPN

In figure 10.1 we present the differences between the previous CO-OPN$_{/2c}$ version and the new CO-OPN$_{/2c++}$ version introduced by our work. At the *syntactic* level the main difference consists of the revision of the notion of *context module* which was previously seen as a specification on its own. On the one hand we have integrated the abstract syntax of *Context modules* with that of *ADT* and *Class* modules; on the other hand we have devised a new notion of CO-OPN specification which is a collection of *ADT*, *Class* and *Context* modules obeying certain *well-formedness*

	CO-OPN$_{/2c}$	**CO-OPN**$_{/2c++}$
Syntax	• A *Context module* contains a CO-OPN$_{/2}$ specification • A CO-OPN$_{/2c}$ specification is a *Context module*	• A *Context module* is a kind of CO-OPN$_{/2c++}$ module • A CO-OPN$_{/2c++}$ specification is a set of *ADT*, *Class* and *Context* modules
Semantics	• By transformation • Does not directly treat *context* modules and *gates*	• Direct Semantics • Uses Hurzeler's composition technique [6] • Inductive definition based on the specification's hierarchical structure

Figure 10.1: Differences between CO-OPN$_{/2c}$ and CO-OPN$_{/2c++}$

conditions.

In what concerns the semantics we have not changed the *transition system* produced at the end, but have proposed an entirely different framework for its calculation. The previous proposal for CO-OPN$_{/2c}$ was transformational and involved translating a CO-OPN$_{/2c}$ specification into a CO-OPN$_{/2}$ one whose semantics were known. The transformational step was difficult to understand and did not explicitly take into consideration CO-OPN$_{/2c}$'s new features. In our CO-OPN$_{/2c++}$ proposal we have taken into consideration the new concepts of *context modules* and *gates* in a primitive fashion which produced a much clearer constructive statement of the language's semantics.

10.1.2 SATEL

In figure 10.2 we present the differences between the previous *CoopnTest* language developed by Péraire and Barbey for CO-OPN$_{/2}$ and the new *SATEL* language we have introduced for CO-OPN$_{/2c++}$. The main difference is the *test selection methodology*, which in *CoopnTest* is directly inherited from the BGM theory and consists of progressively narrowing the *exhaustive test set* by applying successive constraints on the variables of an HML formula. In *SATEL* we natively propose a *constructive approach*, which means expressing the behavior the *test engineer* wishes to test by means of a *test intention*. The *test intention* incorporates both the *behavior to test* — expressed as an HML formula with variables — and its reduction — expressed as constraints over those variables.

In terms of types of SUT's each language is adapted to produce *test cases* for,

	CoopnTest	***SATEL***
Specifications	CO-OPN$_{/2}$	CO-OPN$_{/2c++}$
Test Selection Methodology	Reduction of the *exhaustive test set*	Reduction of multiple behaviors using *test intentions*
Adapted To	Active systems	Reactive systems
Fault Model	Test engineer hypothesis + *unfolding* technique	Refined test engineer hypothesis + axiom-based behavior decomposition
Unit / Integration / System Testing	By choosing classes or objects under test	By enveloping the part of the specification under test with a *context module*
Test Compositionality	Not Supported	Supported
Test Case Format	Couple ⟨HML formula, result⟩	Couple ⟨HML formula, result⟩
Oracle	External observation of the program's behavior	External observation of the program's behavior
Operational Techniques	Prolog SLD resolution with control mechanisms	Not Treated
Editor	Stand Alone Java application	Integrated with *CoopnBuilder*

Figure 10.2: Feature comparison of *CoopnTest* and *SATEL*

CoopnTest was designed to test *active systems* while *SATEL* can be used to test both *active* and *reactive* systems. This difference comes from the existence of *gate* ports in CO-OPN$_{/2c++}$ specifications, allowing the native expression of observations during test selection. Although *CoopnTest* can be used to test *reactive* systems, observation is delegated to the *test driver* which is required to decide operation success — observed as the SUT's non-blockage.

The *fault model* is similar for *CoopnTest* and *SATEL*. Both languages rely on the *hypothesis* stated by the constraint language as well as on automatic sub-domain decomposition based on the semantics expressed in the CO-OPN specification. *SATEL* however brings a more precise fashion of constraining the behaviors under test. This is due to the concept of *test intentions*, the usage of variables for *stimulations* and *observations* and the addition of constraint predicates to the ones proposed in *CoopnTest*. The new *constraint predicates* were defined at the level of *stimulation*, *observation* and method or gate *parameter* variables.

At the level of *unit / integration / system* testing *SATEL* uses the notion of *context module* in order to delimit the part of the specification modeling the part of the SUT under test. Although we do not propose a formal methodology for *unit* or *integration* testing, we exemplify SATEL's possibilities in the case study of chapter 9.

Test compositionality and *reuse* is primitively supported by *SATEL*. This feature, which is not included in *CoopnTest*, reflects *SATEL*'s *constructive approach* to test case generation and allows composing *test intentions* thus generating *test cases* for the composition of the behaviors covered by those *test intentions*. Compositional testing methodologies can be devised using this technology, such as the ones suggested in the case study of chapter 9. We have introduced no changes in what regards *test case format* and the *oracle* decision. In order to have completeness regarding the *exhaustive test set* necessary to establish *bisimulation* between the transition systems of the *specification* and of the *SUT*, we need both *positive* and *negative* behaviors — expressed by the boolean *result* annotating each HML formula. The decision on whether an *error* is uncovered or not by a *test case* is achieved by comparing the *oracles* present in the *test case* — the shape of the HML formula and the truth value annotating it — with the observation of the SUT's behavior. Four cases are possible:

- the sequence of events predicted in the *test case* is observed in the SUT and the truth value of the test case is *true* — no error is discovered;
- the sequence of events predicted in the *test case* is ***not*** observed in the SUT and the truth value of the test case is *true* — at least one error is discovered;
- the sequence of events predicted in the *test case* is observed in the SUT and

	CoopnTest	***SATEL***
Strategy	*Unfolding* technique	*Axiom-based* behavior decomposition
Granularity	Axiomatization of the *algebraic operations*	Combination of the axioms required to fire a given event
Formalization	Operational, in *Prolog*	Annotated *transition systems* + mathematical logics
Adapted To	Algebraic specifications	Component-based, hierarchical specifications

Figure 10.3: Comparison of sub-domain decomposition methods

the truth value of the test case is *false* — at least one error is discovered;

- the sequence of events predicted in the *test case* is ***not*** observed in the SUT and the truth value of the test case is *false* — no error is discovered.

Finally, although a *test intention* editor exists for the SATEL language, we have not implemented an operational semantics for the language. The editor is included in the newest IDE for CO-OPN specifications and respects the abstract syntax we have defined in chapter 7.3. The implemented concrete syntax is the one described in appendix E.

10.1.3 Semantics of SATEL

The fashion in which *SATEL*'s semantics are expressed is considerably different from the one used to express *CoopnTest*'s semantics. Firstly, we employ an algebraic approach to the express the semantics of the exhaustiveness of the variables present in HML formulas. In particular we define a new *global signature* for CO-OPN and SATEL specifications including the new types induced by the testing language as well as the signature of all the constraining predicates. The semantics of the constraint predicates is then given in terms of functions which are part of the algebra for the *global signature* for CO-OPN and SATEL specifications. This is significantly different from the semantics of *CoopnTest* where no integrated treatment for the types of the testing language was introduced.

In terms of the sub-domain decomposition technique used in *CoopnTest* and *SATEL*, they differ both in the produced *test cases* and the fashion in which they are expressed. As can be observed in figure 10.3, the employed strategy is different: while in *CoopnTest* the decomposition is based on the unfolding technique which is operational and depends on the axiomatization of operations defined in *ADT* modules, our technique is directly based on the axioms used in the definition of each behavior of the specification. We introduce with our work a formal logical study on sub-domain decomposition of CO-OPN which is independent of any implementation technique. This differs from the previous work of Péraire and Barbey where the sub-domain decomposition was blurred with both the axiomatization of *algebraic operations* (in ADT modules) and the operational semantics of CO-OPN — given that the unfolding mechanism was used to decompose the behavior of all conditions in the symbolic execution of CO-OPN specifications.

Our technique using *annotated transition systems* is thus adapted to *state-* and *component*-based hierarchical specifications, as opposed to the previous sub-domain decomposition mechanism for *CoopnTest* which was based on the *unfolding* technique initially devised for stateless *algebraic specifications*.

10.2 Discussion

The work presented in this document has been developed along two main axes: CO-OPN$_{/2c++}$ and SATEL. These axes may be seen as relatively independent — the work on CO-OPN$_{/2c++}$ has as objective devising a semantics for component-based concurrent and hierarchical systems; the work on SATEL has as objective the generation of reduced *test sets* from a CO-OPN$_{/2c++}$ specification and a set of *test intentions*. The two lines of work come together when we need to produce the annotated transition system required to reduce the *exhaustive test set* for a given *test intention*. The direct semantics we have produced for CO-OPN$_{/2c++}$ provides a clear base not only to reduce *exhaustive test sets*, but also to calculate oracles for our *test cases*.

With our work we have made a significant step in adapting the BGM testing theory and the previous work on test generation from CO-OPN specifications to the pragmatic aspects of test engineering. In particular we propose a series of new predicates allowing precision and expressiveness while defining *test intentions* for *test set* construction. The notion of *test intention* has been clearly stated — both *syntactically* and *semantically* — and fully integrated from a formal point of view with the existing work on CO-OPN.

We have redefined the notion of *exhaustive test set* in the context of our *test intentions*. Although we prove the *completeness* of SATEL regarding the possibility of building the *exhaustive test set* for a specification, we do not state under which conditions a set of *test intentions* produces the *exhaustive test set*. In fact this condition would correspond to adding a *decomposition hypothesis* as a minimal hypothesis for generating a *test set*. The formalization of the *decomposition hypothesis* would correspond to the fact that the union of all *exhaustive test sets* for all *test intentions* cover the transition system denoting the semantics of the CO-OPN specification of the SUT. Adding this *decomposition hypothesis* allows SATEL to remain in the same *test set* reduction framework as BGM — i.e. pertinence is kept.

We have equally devised a fashion of calculating *equivalence* classes among *test cases* which is independent from the *unfolding mechanism*. This work was missing in the context of CO-OPN as it relates test selection to the domain semantics of the model — the *conditions* expressed by the modeler and the model's structure — rather than taking into consideration the internal semantics of the modeling language itself. It is our claim that our *equivalence class* calculation approach can be employed without many changes while formalizing and implementing *test case reduction* for specification languages other than CO-OPN which do not necessarily have data types defined as algebraic specifications.

10.3 Future Work

Several issues remain open following the work we have presented. In particular:

- As we have already identified, a *decomposition hypothesis* needs to be completely formalized in order to keep *pertinence* regarding the *test set* obtained from a set of *test intentions*;

- Experimentation with operational techniques is necessary in order to implement both the semantics of CO-OPN$_{/2c++}$ and SATEL. We have made some preliminary experiments with Prolog which have worked to a certain extent, but which required modifications of the resolution mechanism that resulted in high complexity. A new approach extending DDD and SDD [63, 64] to *algebraic specifications* is currently being developed in our laboratory and we will apply it to the implementation of SATEL. The main advantage of this approach is that represents *sets* in a compact fashion and allows defining operations on those sets — making it possible to directly code the semantics we have developed in this document;

- We have presented in this thesis a *case study* of realistic size where we have shown that SATEL is a very expressive *test intention* language accommodating mechanisms that allow many methodological possibilities. We are currently setting up a project with the Université de Franche-Comté and the Geneva-based company CLIO where our concept of *test intention* will be further explored;

- As we have proved, SATEL is expressive enough to build the *exhaustive test set* for any SUT. This can be done by declaring a free variable of type $PrimitiveHML$ as part of a *test intention*. It would however be interesting to understand how precise a test engineer can be while writing test intentions, i.e., is it possible to explicitely write *test intentions* for any arbitrary behavior?

- Generalizing our formalization of the calculation of *equivalence classes* for sub-domain decomposition by performing a similar study of other specification formalisms;

- From a methodological point of view it would be interesting to extend SATEL to include parametric *test intentions*. Also, developing a predefined catalog of *test intentions* as a toolkit for the *test engineer* presents an interesting possibility.

Appendix A

The Drink Vending Machine Specification

ADT Modules

```
ADT Drink;

Interface

    Sort
        drink;

    Generators
        IceTea : -> drink;
        Soda : -> drink;
        Beer : -> drink;
        Water : -> drink;
End Drink;
```

Class Modules

```
Class moneyBox;

Interface

    Use
        Naturals;
        Drink;
    Type
        moneybox;
    Gates
        askPrice _ _ : drink natural;
        notEnoughMoney;
    Methods
```

```
        insertCoin;
        consume _ : drink;

Body

    Place
        coinHolder _ : natural;
    Initial
        coinHolder 0;
    Axioms
        insertCoin::
                coinHolder am -> coinHolder am + 1;
        (am > = p) = true =>
                consume d With checkPrice d p::
                coinHolder am -> coinHolder am - p;
        (am < p) = true =>
                consume d With checkPrice d p / / notEnoughMoney::
                coinHolder am -> coinHolder am;

    Where
        am : natural;
        p : natural;
        d : drink;

End moneyBox;
```

```
Class drinkShelf;

Interface

    Use
        Drink;
        Naturals;
    Type
        drinkshelf;
    Gates
        distributeDrink _ : drink;
        notEnoughDrinks;
    Methods
        giveDrink _ : drink;
        returnPrice _ _ : drink natural;

Body

    Places
        availableUnits _ : natural;
        name _ : drink;
        price _ : natural;
    Axioms
        (units > 0) = true =>
                giveDrink d::
                availableUnits units, name d -> availableUnits units - 1, name d;
        (units = 0) = true =>
                giveDrink d With notEnoughDrinks::
                availableUnits units, name d -> availableUnits units, name d;
        returnPrice d p::
                name d, price p -> name d, price p;

    Where
        units : natural;
        p : natural;
        d : drink;
```

```
End drinkShelf;
```

Context Modules

```
Context DVM;

Interface

     Use
          Drink;
          drinkShelf;
          moneyBox;

     Gates
          notEnoughMoney;
          notEnoughDrinks;
          distributeDrink _ : drink;

     Methods
          insertCoin;
          buyDrink _ : drink;

Body

     Objects
          beerShelf : drinkshelf;
          waterShelf : drinkshelf;
          iceTeaShelf : drinkshelf;
          sodaShelf : drinkshelf;
          mBox : moneybox;

     Axioms
          insertCoin With mBox . insertCoin;
          mBox . checkPrice (d, p) With shelf . returnPrice (d, p);
          buyDrink d With mBox . consume d // shelf . giveDrink d;
          shelf . distributeDrink d With distributeDrink d;
          mBox . notEnoughMoney With notEnoughMoney;
          shelf . notEnoughDrinks With notEnoughDrinks;

     Where
          shelf : drinkshelf;
          d : drink;
          p : natural;

End DVM;
```

Appendix B

The Banking Server Specification

ADT Modules

```
ADT User;

Inherit MyChar;

    Rename
        char -> user;

End User;
```

```

ADT Money;

Inherit Naturals;

    Rename
            natural -> money;

End Money;
```

```
ADT Challenge;

Interface

    Use
        Digit;
        User;
        Booleans;
    Sort
        challenge;
    Generator
        newChal _ _ : user, digit -> challenge;
    Operation
        _ = _ : challenge, challenge -> boolean;

Body

```

```
    Axioms
        ((c1 = c2) = true) & ((d1 = d2) = true) =>
                        ((newChal c1 d1) = (newChal c2 d2)) = true;
        ! (((c1 = c2) = true) & ((d1 = d2) = true)) =>
                        ((newChal c1 d1) = (newChal c2 d2)) = false;
    Where
        c1, c2 : user;
        d1, d2 : digit;

End Challenge;
```

```
ADT Password;

Interface

    Use
        Digit;
        Booleans;
    Sort
        password;
    Generator
        newPassword _ _ _ _ : digit digit digit digit -> password;
    Operation
        _ = _ : password password -> boolean;

Body

    Axioms
      (n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true
              => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = true;
      ! ((n1 = m1) = true & (n2 = m2) = true & (n3 = m3) = true & (n4 = m4) = true)
              => (newPassword n1 n2 n3 n4 = newPassword m1 m2 m3 m4) = false;
    Where
        n1, n2, n3, n4, m1, m2, m3, m4 : digit;

End Password;
```

Class Modules

```
Class LoginManager;

Interface

    Use
        Password;
        Challenge;
        User;

    Type
        loginManager;

    Gates
        verifyPass _ _ : user password;
        askChallenge _ : challenge;

    Methods
        isLogged _ : user;
        login _ : user;
        insertPassword _ _ : user password;
        logout _ : user;

Body

    Places
        unLogged _ : user;
        waitingForPass _ : user;
        logged _ : user;

    Initial
        unLogged d, unLogged e, unLogged f;

    Axioms
        login usr With
                this . askChallenge (newChal a 1)::
                unLogged usr -> waitingForPass usr;
        insertPassword usr p With
                this . verifyPass usr p::
                waitingForPass usr -> logged usr;
        logout usr::
                logged usr -> unLogged usr;
        isLogged usr::
                logged usr -> logged usr;

    Where
        usr : user;
        p : password;
        this : loginManager;

End LoginManager;
```

```
Class Account;

Interface

    Use
        User;
        Password;
        Money;

    Type
        account;

    Gates
        giveMoney _ : money;


        notEnoughMoney;

    Methods
        deposit _ : money;
        withdraw _ : money;
        hasPass _ _ : user password;

Body

    Places
        userName _ : user;
        userPass _ : password;
        balance _ : money;
    Axioms
        deposit am::
                balance b -> balance b + am;
        (b > = am) = true =>
                withdraw am With this . giveMoney am::
                balance b -> balance b - am;
        (b > = am) = false =>
                withdraw am With this . notEnoughMoney::
                balance b -> balance b;
        hasPass usr p::
                userName usr, userPass p -> userName usr, userPass p;

    Where
        b : money;
        am : money;
        usr : user;
        p : password;
        this : account;

End Account;
```

Context Modules

```
Context BankingServer;

Interface

    Use
        User;
        Money;
        Password;
        Challenge;
        LoginManager;
        Account;

    Gates
        askChallenge _ : challenge;
        giveMoney _ : money;

    Methods
        login _ : user;
        insertPassword _ _ : user password;
        logout _ : user;
        deposit _ _ : user money;
        withdraw _ _ : user money;

Body

    Objects
        lmObj : loginManager;
        accObj1 : account;
        accObj2 : account;

    Axioms
        login usr With lmObj . login usr;
        insertPassword usr p With lmObj . insertPassword usr p;
        logout usr With lmObj . logout usr;
        deposit usr am With lmObj . isLogged usr // accVar . deposit am;
        withdraw usr am With lmObj . isLogged usr // accVar . withdraw am;
        accVar . giveMoney am With giveMoney am;
        lmObj . askChallenge c With askChallenge c;
        lmObj . verifyPass usr p With accVar . hasPass usr p;

    Where
        accVar : account;
        usr : user;
        p : password;
        c : challenge;
        am : money;

End BankingServer;
```

Appendix C

The RTaxPM CO-OPN Specification

ADT Modules

```
ADT TaxerPM;

Interface

    Use
        Char;
        Booleans;

    Sort
        taxer;

    Generator
        newTaxer _ allowedToStamp _ : char boolean -> taxer;

End TaxerPM;
```

```
ADT Dossier;

Interface

    Use
        Char;
        Booleans;

    Sort
        dossier;

    Generator
        newDossier _ stampNeeded _ : char boolean -> dossier;

End Dossier;
```

Class Modules

```
Class LoginManager;

Interface

    Use
        TaxerPM;
        BlackTokens;
        MyChar;
        Booleans;

    Type
        loginManager;

    Gate
        alreadyLogged;

    Methods
        loginTaxer _ : char;
        logoutTaxer;
        isLogged _ : taxer;

Body

    Places
        taxerLogged _ : blackToken;
        noTaxerLogged _ : blackToken;
        loggedUsers _ : taxer;
        unLoggedUsers _ : taxer;

    Initial
        unLoggedUsers newTaxer a allowedToStamp false, unLoggedUsers newTaxer b
            allowedToStamp false, unLoggedUsers newTaxer l allowedToStamp true,
            unLoggedUsers newTaxer m allowedToStamp true, noTaxerLogged @;

    Axioms
        userClearance = true =>
                loginTaxer aTaxerN::
                unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                    noTaxerLogged @ -> loggedUsers newTaxer aTaxerN allowedToStamp
                    userClearance, taxerLogged @;
        userClearance = false =>
                loginTaxer aTaxerN::
                unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                    noTaxerLogged @ -> loggedUsers newTaxer aTaxerN allowedToStamp
                    userClearance, taxerLogged @;
        loginTaxer aTaxerN With alreadyLogged::
                unLoggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                    taxerLogged @ -> unLoggedUsers newTaxer aTaxerN allowedToStamp
                    userClearance, taxerLogged @;
        logoutTaxer::
                loggedUsers newTaxer aTaxerN allowedToStamp userClearance,
                    taxerLogged @ -> unLoggedUsers newTaxer aTaxerN allowedToStamp
                    userClearance, noTaxerLogged @;
        isLogged aTaxer::
                loggedUsers aTaxer -> loggedUsers aTaxer;

    Where
        aTaxerN : char;
        aTaxer : taxer;
        userClearance : boolean;

End LoginManager;
```

```
Class TaxationEngine;

Interface

    Use
        TaxerPM;
        Dossier;
        BlackTokens;
        Booleans;

    Type
        taxationEngine;

    Gates
        isLogged _ : taxer;
        reachedAtWork;
        reachedControlled;
        reachedToNotify;
        reachedToStamp;
        reachedToCorrect;
        reachedNotified;
        reachedCancelled;
        reachedEnd;

    Methods
        createDossier _ : dossier;
        check;
        correct;
        supress;
        validate;
        notify;
        stamp;
        oppose;
        cancel;

Body

    Places
        startState _ : blackToken;
        atWork _ : dossier;
        controlled _ : dossier;
        toStamp _ : dossier;
        toCorrect _ : dossier;
        toNotify _ : dossier;
        notified _ : dossier;
        canceled _ : dossier;
        endState _ : blackToken;

    Initial
        startState @;

    Axioms
        dossierClearance = true =>
                createDossier newDossier aDossierN stampNeeded dossierClearance
                    With isLogged aTaxer / / reachedAtWork::
                startState @ -> atWork aDossier;
        dossierClearance = false =>
```

```
                  createDossier newDossier aDossierN stampNeeded dossierClearance
                      With isLogged aTaxer // reachedAtWork::
                  startState @ -> atWork aDossier;
         check With isLogged aTaxer // reachedControlled::
                  atWork aDossier -> controlled aDossier;
         ((userClearance = false) and (dossierClearance = true)) = true =>
                  validate With isLogged newTaxer aTaxerN allowedToStamp
                      userClearance // reachedToStamp::
                  controlled newDossier aDossierN stampNeeded dossierClearance ->
                      toStamp newDossier aDossierN stampNeeded dossierClearance;
         ((userClearance = true) or (dossierClearance = false)) = true =>
                  validate With isLogged newTaxer aTaxerN allowedToStamp
                      userClearance // reachedToNotify::
                  controlled newDossier aDossierN stampNeeded dossierClearance ->
                      toNotify newDossier aDossierN stampNeeded dossierClearance;
         userClearance = true =>
                  oppose With isLogged newTaxer aTaxerN allowedToStamp userClearance
                      // reachedToCorrect::
                  toStamp aDossier -> toCorrect aDossier;
         userClearance = true =>
                  stamp With isLogged newTaxer aTaxerN allowedToStamp userClearance /
                       / reachedToNotify::
                  toStamp aDossier -> toNotify aDossier;
         userClearance = true =>
                  stamp With isLogged newTaxer aTaxerN allowedToStamp userClearance /
                       / reachedToNotify::
                  toCorrect aDossier -> toNotify aDossier;
         notify With isLogged aTaxer // reachedNotified::
                  toNotify aDossier -> notified aDossier;
         correct With isLogged aTaxer // reachedAtWork::
                  controlled aDossier -> atWork aDossier;
         correct With isLogged aTaxer // reachedAtWork::
                  toStamp aDossier -> atWork aDossier;
         correct With isLogged aTaxer // reachedAtWork::
                  toCorrect aDossier -> atWork aDossier;
         correct With isLogged aTaxer // reachedAtWork::
                  toNotify aDossier -> atWork aDossier;
         supress With isLogged aTaxer // reachedEnd::
                  controlled aDossier -> endState @;
         supress With isLogged aTaxer // reachedEnd::
                  toCorrect aDossier -> endState @;
         supress With isLogged aTaxer // reachedEnd::
                  toNotify aDossier -> endState @;
         cancel With isLogged aTaxer // reachedEnd::
                  notified aDossier -> canceled aDossier;

    Where
         aTaxerN : char;
         aTaxer : taxer;
         aDossierN : char;
         aDossier : dossier;
         userClearance : boolean;
         dossierClearance : boolean;

End TaxationEngine;
```

Context Modules

```
Context LoginMgrInterface;

Interface

    Use
        LoginManager;
        MyChar;
        TaxerPM;

    Methods
        loginTaxer _ : char;
        logoutTaxer;
        isLogged _ : taxer;

Body

    Object
        loginMgr : loginManager;

    Axioms
        loginTaxer aTaxerN With loginMgr . loginTaxer aTaxerN;

        logoutTaxer With loginMgr . logoutTaxer;

        isLogged aTaxer With loginMgr . isLogged aTaxer;

    Where
        aTaxerN : char;
        aTaxer : taxer;

End LoginMgrInterface;
```

```
Context TaxationInterface;

Interface

    Use
        TaxerPM;
        TaxationEngine;
        LoginManager;
        Dossier;
        Booleans;

    Gates
        alreadyLogged;
        reachedAtWork;


        reachedControlled;
        reachedToNotify;
        reachedToStamp;
        reachedToCorrect;
        reachedNotified;
        reachedCancelled;
        reachedEnd;

    Methods
        loginTaxer _ : char;
        logoutTaxer;
        createDossier _ : dossier;
        check;
        validate;
        oppose;
        stamp;
        notify;
        correct;
        supress;
        cancel;

Body

    Objects
        taxEngine : taxationEngine;
        loginMgr : loginManager;

    Axioms
        taxEngine.islogged t With loginMgr t;
        loginMgr . alreadyLogged With alreadyLogged;
        createDossier d With taxEngine . createDossier d;
        loginTaxer tn With loginMgr . loginTaxer tn;
        logoutTaxer With loginMgr . logoutTaxer;
        check With taxEngine . check;
        validate With taxEngine . validate;
        stamp With taxEngine . stamp;
        oppose With taxEngine . oppose;
        notify With taxEngine . notify;
        correct With taxEngine . correct;
        supress With taxEngine . supress;
        cancel With taxEngine . cancel;
        taxEngine . reachedAtWork With reachedAtWork;
        taxEngine . reachedControlled With reachedControlled;
        taxEngine . reachedToStamp With reachedToStamp;
        taxEngine . reachedToCorrect With reachedToCorrect;
        taxEngine . reachedToNotify With reachedToNotify;
        taxEngine . reachedNotified With reachedNotified;
        taxEngine . reachedCancelled With reachedCancelled;
        taxEngine . reachedEnd With reachedEnd;

    Where
        d : dossier;
        aUserN : char;
        t : taxer;

End TaxationInterface;
```

Appendix D

Class Normal Form

In order to use the component composition technique proposed by Hurzeler in [6] we can only consider synchronizations linking *gate* to *method* ports of two different components or of the same component. In the syntax for class axioms which is proposed by CO-OPN$_{/2}$ [1] as well as CO-OPN$_{/2c}$ [2] it is possible to synchronize two objects by linking *method* to *method* ports. In order to keep *backwards compatibility* with previous CO-OPN specifications we thus need a mechanism to transform *method-method* synchronizations into *gate-method* synchronizations.

This problem can be avoided if we consider that all classes in a CO-OPN$_{/2c++}$ specification are syntactically transformed into a kind of "*normal form*", where *method-method* synchronizations are transformed into *gate-method* synchronizations using the method that we now describe. Assume that a CO-OPN class module includes the following *behavioral formula*:

$$o.m(x, y) \textit{ with } o'.m'(x) :: \textit{cond}(x, y) \Rightarrow \textit{pre} \rightarrow \textit{post}$$

In this axiom o and o' are object identifiers, m and m' are method ports having as parameters variables x and y. The condition for the firing of the axiom also depends on variables x and y.

The normalization of this *behavioral formula* corresponds to splitting it into into a syntactically modified *behavioral formula* and a *coordination formula* (see definition 4.4.11) that we will consider while building the semantics as part of the contextual coordination performed by an enveloping *context* module. The normalization of the behavioral formula defined above produces the following set of axioms:

$$\big\{o.m(x, y) \textit{ with } o.g(x, y) :: \textit{cond}(x, y) \Rightarrow \textit{pre} \rightarrow \textit{post},$$
$$\textit{cond}(x, y) :: o.g(x, y) \textit{ with } o'.m'(x)\big\}$$

There are several points to be understood in the *behavioral formula* normalization: firstly we introduce a new *gate* port at the level of the considered *class* module. Clearly this *gate* port has to be assigned a different name from the ones already existing for that *class* module. The parameters for the *gate* port will be the same as the ones for the *method* port m and the effective parameter values will also be the same. Secondly, the new *coordination formula* synchronizes the class's newly introduced *gate* port with method m'. For the new *gate* port we use the same effective parameters as the ones for m and we keep the same conditions as in the original *behavioral formula*.

The method also scales to a synchronization involving complex synchronization expressions on the right side of the event on a *behavioral formula*. For instance, if we extend the previous example:

$$o.m(x,y)\ \mathit{with}\ o'.m'(x)\ ..\ o''.m''(y) :: \mathit{cond}(x,y) \Rightarrow \mathit{pre} \rightarrow \mathit{post}$$

we would obtain the following normal form:

$$\big\{o.m(x,y)\ \mathit{with}\ o.g(x,y) :: \mathit{cond}(x,y) \Rightarrow \mathit{pre} \rightarrow \mathit{post}$$
$$\mathit{cond}(x,y) :: o.g(x,y)\ \mathit{with}\ o'.m'(x)\ ..\ o''.m''(y)\big\}$$

Appendix E

SATEL's Concrete Syntax

In this appendix we present a BNF-like definition of the concrete syntax of SATEL. Notice that the grammar is defined bottom-down starting by the production at the level of the test intention module.

$\langle$*TestIntentionModule*$\rangle\rightarrow$ **'TestIntentionSet'** $\langle$*Word*$\rangle$ Focus $\langle$*Word*$\rangle$
  [$\langle$*TestIntentionInterface*$\rangle$]
  [$\langle$*TestIntentionBody*$\rangle$]
  'End' $\langle$*Word*$\rangle^{+}$
  ';'

$\langle$*TestIntentionInterface*$\rangle\rightarrow$ **'Interface'** [$\langle$*TestIntentionDeclaration*$\rangle$]

$\langle$*TestIntentionBody*$\rangle\rightarrow$ **'Body'**
  [$\langle$*TestIntentionDeclaration*$\rangle$]
  [$\langle$*UseDeclaration*$\rangle$]
  [$\langle$*AxiomDeclaration*$\rangle$]
  [$\langle$*VariableDeclaration*$\rangle$]
  [$\langle$*ExternalDeclaration*$\rangle$]

$\langle$*TestIntentionDeclaration*$\rangle\rightarrow$ **'TestIntention'** ($\langle$*Word*$\rangle$ **';'**)$^{+}$

$\langle$*UseDeclaration*$\rangle\rightarrow$ **'Use'** ($\langle$*Word*$\rangle$ **';'**)$^{+}$

$\langle$*AxiomDeclaration*$\rangle\rightarrow$ **'Axioms'** ($\langle$*Axiom*$\rangle$**';'**)$^{+}$

⟨*VariableDeclaration*⟩→'**Variables**'
(⟨*Word*⟩ '**:**' ⟨*Word*⟩ '**;**'
| ⟨*Word*⟩ '**:**' '**primitiveHML**' '**;**'
| ⟨*Word*⟩ '**:**' '**primitiveStimulation**' '**;**'
| ⟨*Word*⟩ '**:**' '**primitiveObservation**' '**;**'
| ⟨*Word*⟩ '**:**' '**primitiveInteger**' '**;**'
| ⟨*Word*⟩ '**:**' '**primitiveBoolean**' '**;**')$^+$

⟨*ExternalDeclaration*⟩→ '**External**' (⟨*Word*⟩ '**in**' ⟨*Word*⟩ '**;**')$^+$

⟨*Axiom*⟩→ [⟨*Condition*⟩ '**=>**'] ⟨*Inclusion*⟩

⟨*Inclusion*⟩→ ⟨*HMLTerm*⟩ '**in**' ⟨*Word*⟩

⟨*Condition*⟩→ [DomainQuantifier] | ConditionBody

⟨*DomainQuantifier*⟩→ '**uniformity**' '**(**' ⟨*NameList*⟩ '**)**' ['**subuniformity**' '**(**' ⟨*NameList*⟩ '**)**']
| '**subuniformity**' '**(**' ⟨*NameList*⟩ '**)**' ['**uniformity**' '**(**' ⟨*NameList*⟩ '**)**']

⟨*ConditionBody*⟩→ ⟨*ConditionAtom*⟩ ('**,**' ⟨*ConditionAtom*⟩)*

⟨*ConditionAtom*⟩→ AlgebraicEquality
| ⟨*BooleanTerm*⟩
| ⟨*ArithmeticPredicate*⟩
| ⟨*Inclusion*⟩

⟨*AlgebraicEquality*⟩→ '**{**' ⟨*AlgebraicTerm*⟩ '**}**' '**=**' '**{**' ⟨*AlgebraicTerm*⟩ '**}**'
| ⟨*HMLTerm*⟩ '**=**' ⟨*HMLTerm*⟩
| '**{**' ⟨*SynchronizationTerm*⟩ '**}**' '**=**' '**{**' ⟨*SynchronizationTerm*⟩ '**}**'
| ⟨*BooleanTerm*⟩ '**=**' ⟨*BooleanTerm*⟩
| ⟨*ArithmeticTerm*⟩ '**=**' ⟨*ArithmeticTerm*⟩

⟨*HMLTerm*⟩→ HMLFormula ('**.**' ⟨*HMLFormula*⟩)*

⟨*HMLFormula*⟩→ '**HML**' '**(**' ⟨*HMLFormulaContent*⟩ '**)**'

⟨*HMLFormulaContent*⟩→ T

| ‘**{**’ ⟨*HMLEvent*⟩ ‘**}**’ ⟨*HMLFormulaContent*⟩
| ‘**not**’ ‘(’ ⟨*HMLFormulaContent*⟩ ‘)’
| ‘(’ ⟨*HMLFormulaContent*⟩ ‘**and**’ ⟨*HMLFormulaContent*⟩ ‘)’

⟨*HMLEvent*⟩→ ⟨*SynchronizationTerm*⟩ ‘**with**’ ⟨*SynchronizationTerm*⟩

⟨*BooleanTerm*⟩→ ‘ (’ ⟨*BooleanTerm*⟩ ‘**and**’ ⟨*BooleanTerm*⟩ ‘)’
| ‘(’ ⟨*BooleanTerm*⟩ ‘**or**’ ⟨*BooleanTerm*⟩ ‘)’
| ‘**not**’ ‘(’ ⟨*BooleanTerm*⟩ ‘)’
| ‘**sequence**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**positive**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**trace**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**onlyConstructor**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**onlyMutator**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**onlyObserver**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**onlySimultaneity**’ ‘(’ ⟨*SynchronizationTerm*⟩ ‘)’
| ‘**onlySequence**’ ‘(’ ⟨*SynchronizationTerm*⟩ ‘)’
| ‘**onlyAlternative**’ ‘(’ ⟨*SynchronizationTerm*⟩ ‘)’
| ⟨*BooleanVariable*⟩
| ⟨*Boolean*⟩

⟨*ArithmeticTerm*⟩→ ⟨*ArithmeticTerm*⟩ ⟨*BiAOp*⟩ ⟨*ArithmeticTerm*⟩
| ‘**-**’ ⟨*ArithmeticTerm*⟩
| ‘(’ ⟨*ArithmeticTerm*⟩ ‘)’
| ‘**nbEvents**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**depth**’ ‘(’ ⟨*HMLTerm*⟩ ‘)’
| ‘**nbOccurrences**’ ‘(’ ⟨*HMLTerm*⟩ ‘**,**’ ⟨*Word*⟩ ‘)’
| ‘**nbSynchro**’ ‘(’ SynchronizationTerm ‘)’
| ⟨*IntegerVariable*⟩
| ⟨*Integer*⟩

⟨*BiAOp*⟩→ ‘+’ | ‘**-**’ | ‘*’ | ‘/’

⟨*ArithmeticPredicate*⟩→ ⟨*ArithmeticTerm*⟩ ⟨*BiAPred*⟩ ⟨*ArithmeticTerm*⟩

⟨*BiAPred*⟩→ ‘==’ | ‘<>’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’

⟨*NameList*⟩→ ⟨*Word*⟩ (‘,’ ⟨*Word*⟩)*

$\langle$*Word*$\rangle \rightarrow$ $\langle$*Letter*$\rangle$ | $\langle$*Word*$\rangle$ $\langle$*Letter*$\rangle$ | $\langle$*Word*$\rangle$ $\langle$*Digit*$\rangle$ | $\langle$*Word*$\rangle$ $\langle$*Symbol*$\rangle$

$\langle$*Letter*$\rangle \rightarrow$ [A-Z] | [a-z]

$\langle$*Digit*$\rangle \rightarrow$ [0-9]

$\langle$*Symbol*$\rangle$ $\rightarrow$ **'_'** | **'-'**

Bibliography

[1] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva, 1997.

[2] Mathieu Buffo. *Contextual Coordination: a coordination model for distributed object systems.* PhD thesis, Université de Genève, 1997.

[3] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, 1991.

[4] Cecile Péraire. *Formal testing of object-oriented software: from the method to the tool.* PhD thesis, EPFL - Switzerland, 1998.

[5] Stéphane Barbey. *Test Selection for Specification-Based Unit Testing of Object-Oriented Software Based on Formal Specifications.* PhD thesis, EPFL - Switzerland, 1997.

[6] David Huerzeler. *Flexible subtyping relations for component-oriented formalisms and their verification.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2004.

[7] Smv Group. CoopnBuilder, 2007. `http://smv.unige.ch/tiki-index.php?page=DownloadCoopn`.

[8] Gordon E. Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, pages 56–59, 2000.

[9] Jr. Frederick P. Brooks. *The mythical man-month – Essays on Software-Engineering.* Addison Wesley, 1975.

[10] Frederick P. Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[11] E. W. Dijkstra. Notes on structured programming. In E. W. Dijkstra O.-J. Dahl and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82, 1972.

[12] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[13] Object Management Group members. Uml 2.0 superstructure specification. Technical report, OMG, August 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[14] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[15] Object Management Group. Mda guide version 1.0.1. Technical report, OMG, June 2003.

[16] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[17] C B Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1986.

[18] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[19] Edsger W. Dijkstra. On the cruelty of really teaching computing science. circulated privately, December 1988.

[20] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, 2000.

[21] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[22] Alexander Pretschner, Oscar Slotosch, Ernst Aiglstorfer, and Stefan Kriebel. Model-based testing for real. *STTT*, 5(2-3):140–157, 2004.

[23] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM.

[24] Gilles Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT, Vol.2*, pages 99–119, 1991.

[25] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[26] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[27] G.W. Brams. *Réseaux de Petri: théorie e pratique - Tome 1: Théorie et analyse.* Masson, 1983.

[28] G.W. Brams. *Réseaux de Petri: théorie e pratique - Tome 2: Modélisation et applications.* Masson, 1983.

[29] Jean-Marie Proth and Xiaolan Xie. Petri nets: a tool for design and management of manufacturing systems. *Quality and Reliability Engineering International*, 13(2):114–114, December 1998.

[30] Bruno Marre. *Une métode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques.* PhD thesis, Universite de Paris-Sud – Centre D'Orsay, 1991.

[31] Bruno Marre. Génération automatique de jeux de tests, une solution: Spécifications algébriques et programmation logique. In *SPLT*, pages 213–, 1989.

[32] Mathieu Buffo. *Contextual Coordination: a Coordination Model for Distributed Object Systems.* PhD thesis, University of Geneva, 1997.

[33] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[34] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, and M.-L. Potet. Test purposes: Adapting the notion of specification to testing. *ase*, 00:127, 2001.

[35] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[36] Stanislav Chachkov. *Generation of Object-Oriented programs from CO-OPN Specifications.* PhD thesis, Ecole Polytechnique Federal de Lausanne, 2004.

[37] Adrien Chantre. Implementation of a test language for co-opn: User guide and graphical user interface. Bachelor Thesis, Université de Genève, 2006.

[38] Sérgio Coelho. Implementation of a test language for co-opn: Syntactic and static semantics verifications. Bachelor Thesis, Université de G'enève, 2006.

[39] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.

[40] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. April 2006.

[41] Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, London, UK, 1995. Springer-Verlag.

[42] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[43] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999.

[44] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *ISSRE*, pages 3–14, 2002.

[45] Cécile Péraire, Stéphane Barbey, and Didier Buchs. Test selection for object-oriented software based on formal specifications. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 385–403, London, UK, UK, 1998. Chapman & Hall, Ltd.

[46] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In *FASE*, pages 281–294, 2004.

[47] Bruno Legeard and Fabien Peureux. Generation of functional test sequences from b formal specifications-presentation and industrial case study. In *ASE*, pages 377–381, 2001.

[48] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[49] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276. North-Holland, 1994.

[50] M. Clatin. *Manuel d'utilisation de TVEDA V3*. France Télécom, 1996. Manual LAA/EIA/EVP/109.

[51] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Sci. Comput. Program.*, 29(1-2):123–146, 1997.

[52] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In *IWTCS*, pages 179–196, 1999.

[53] Bruno Legeard, Fabien Peureux, and Mark Utting. Controlling test case explosion in test generation from b formal models. *Softw. Test., Verif. Reliab.*, 14(2):81–103, 2004.

[54] O. Biberstein, D. Buchs, C. Buffard, M. Buffo, J. Flumet, J. Hulaas, G. Di-Marzo, and P. Racloz. Sands1.5/co-opn1.5, an overview of the language and its supporting tools. Technical Report 95/133, Swiss Federal Institute of Technology (EPFL), June 1995.

[55] Stanislav Chachkov and Didier Buchs. From an abstract object-oriented model to a ready-to-use embedded system controller. In *Rapid System Prototyping, Monterey, CA*, pages 142 – 148. IEEE Computer Society Press, June 2001.

[56] Stanislav Chachkov and Didier Buchs. From formal specifications to ready-to-use software components: The concurrent object-oriented petri net approach. In *International Conference on Application of Concurrency to System Design, Newcastle*, pages 99 – 110. IEEE Computer Society Press, June 2001.

[57] Joseph A. Goguen and José Meseguer. Order-sorted algebra 1: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[58] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS, pages 70–127. Springer-Verlag, 2001.

[59] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, 1980.

[60] David Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science*, 104:167–183, 1981.

[61] Wikipedia. Application software definition. `http://en.wikipedia.org/wiki/Applicationsoftware`.

[62] Laboratoire d'Informatique de L'Université de Franche-Compté, University of Geneva, LEIRIOS Technologies, and CTI (Geneva). Projet valid: Rapport final de projet. Technical report, Projet INTERREG III France-Suisse, 2006.

[63] J.-M. Couvreur, E.Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. Wacrenier. Data decision diagram for petri nets analysis. In *23rd international conference on application and theory of Petri Nets (ATPN 2002), jun 2002, Australia.*, volume LNCS vol 2360, 2002.

[64] Y. Thierry-Mieg, J.M Ilié, and D. Poitrenaud. A symbolic symbolic state space representation. In D. de Frutos-Escrig and M. Núñez, editors, *Proceedings of Formal Techniques for Networked and Distributed Systems (FORTE'04)*, volume 3235 of *Lecture Notes in Computer Science*, pages 276–291. Springer, September 2004.