STEPWISE REFINEMENT OF FORMAL SPECIFICATIONS BASED ON LOGICAL FORMULAE

FROM CO-OPN/2 SPECIFICATIONS TO JAVA PROGRAMS

THÈSE N° 1931 (1999)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Giovanna A. DI MARZO SERUGENDO

Diplômée ès sciences informatiques, Université Genève

Diplômée ès sciences mathématiques, Université Genève

Originaire de Genève (GE)

acceptée sur proposition du jury:

Prof. A. Strohmeier, directeur de thèse

Dr D. Buchs, corapporteur

Prof. G. Coray, corapporteur

Dr N. Guelfi, corapporteur

Prof. J. Harms, corapporteur

Prof. D. Mandrioli, corapporteur

Lausanne, EPFL

1999

Acknowledgements

This thesis has been made possible, influenced, and realised thanks to many people. I take the opportunity to thank all of them:

- Especially *Professor Alfred Strohmeier* who accepted to supervise this thesis and enabled me to join his team. His perspicacity always pointed out relevant features;
- *Dr Didier Buchs* to have introduced me to CO-OPN/2 and to the world of formal methods. He always brought me his support and suggested to undertake my thesis at EPFL;
- *Professor Giovanni Coray* for his comments and the improvements he suggested for the thesis;
- Particularly *Dr Nicolas Guelfi* who guided me through the notions of refinement. His vision of integration enabled me to make of this thesis a coherent whole incorporating several works previously performed in the CO-OPN/2 framework;
- Especially *Professor Jürgen Harms* who offered me the opportunity to be initiated to scientific research and graduate teaching; and who always offered me his support, even though formal methods are only a restricted aspect of his research areas;
- *Professor Dino Mandrioli* for his invaluable comments on the theoretical part of this thesis;
- All my colleagues of the Software Engineering Laboratory of EPFL: *Anne, Cécile, Enzo, Gabriel, Jörg, Julie, Mathieu, Mohamed, Shane, Thomas*, and *Stéphane*; as well as the past members of the ConForM Team: *Jacques, Olivier*, and *Pascal*;
- The past and current members of the Teleinformatics and Operating Systems group of the Computer Science Department of the University of Geneva: *Alex, Christian, David and David, Eduardo, Jarle, Jean-Francois, Julien, Lassaad, Mira, Muhugusa, Noria, Stanislas, Stéphane*, and *Vito*. All of them contributed to make of this group a friendly and supporting work team;

- The early friends: *Catherine* and *Peter*, *Rosanna* and *Djamel*, *Laura* and *Nicolas*. We studied hard together to understand the subtleties of mathematics, physics, and computer science.

A special thank goes to my *parents* who gave me the opportunity to get a higher education. I wish as well to cordially thank *Patrice* for his constant encouragements, and for giving me appropriate advices when necessary. Finally, a big kiss goes to *Chloé* who enabled me to refresh my mind in the wonderful world of children.

Résumé

Une des démarches permettant d'augmenter la qualité et la fiabilité des logiciels s'exécutant sur des systèmes répartis consiste en l'utilisation de méthodes de génie logiciel dites formelles. La majorité des méthodes formelles actuellement existantes correspondent en fait plus à des langages de spécifications formelles qu'à des méthodes proprement dites. Ceci provient du fait que les deux aspects fondamentaux que sont: la logique d'utilisation du langage et la couverture du cycle de vie du logiciel ne sont, pour la plupart, pas définis. Le développement par raffinements successifs est l'un des moyens permettant de définir ces deux aspects.

Cette thèse vise à la définition des notions de raffinement et d'implantation de spécifications formelles orientées-modèles. Elle apporte par là-même une base méthodologique permettant d'utiliser un tel langage de spécifications lors d'un développement par raffinements successifs et lors de l'étape d'implantation.

Cette thèse définit, dans un premier temps, un cadre théorique pour le raffinement et l'implantation de spécification formelles orientées-modèles. L'idée principale consiste à associer un contrat à chaque spécification. Un contrat représente explicitement l'ensemble des propriétés de la spécification qu'il est nécessaire de préserver lors d'un raffinement de cette spécification. Pour montrer qu'une spécification concrète raffine une spécification plus abstraite, il s'agit alors de montrer que le contrat de la spécification concrète est suffisant pour assurer les propriétés correspondant au contrat de la spécification abstraite.

La seconde partie de cette thèse consiste à appliquer ce cadre théorique dans le contexte du langage CO-OPN/2. CO-OPN/2 est un langage de spécifications formelles orienté-objet, fondé sur les réseaux de Petri et les spécifications algébriques. Il est donc proposé pour ce langage, une définition des notions de contrats, de raffinement et d'implantation. Les contrats sont exprimés à l'aide de la logique temporelle de Hennessy-Milner (HML). Cette logique facilite la vérification des propriétés contractuelles, ainsi que la vérification des étapes de raffinement. Le raffinement et l'implantation sont contrôlés sémantiquement par la satisfaction des contrats; syntaxiquement, un renommage est autorisé. L'implantation utilisant le langage de programmation Java a été plus particulièrement étudiée. Il est montré comment spécifier des classes du langage de programmation Java à l'aide du langage CO-OPN/2, afin que la dernière étape du processus de raffinement conduise à une spécification entièrement construite à l'aide de composants CO-OPN/2 spécifiant des

classes Java. L'étape d'implantation dans le langage Java lui-même en est ainsi facilitée.

La troisième partie de cette thèse montre comment il est possible de vérifier pratiquement qu'une spécification CO-OPN/2 satisfait son propre contrat, qu'une étape de raffinement est correctement effectuée, et enfin que l'étape d'implantation est correctement réalisée. Ces vérifications s'effectuent à l'aide de la théorie du test fournie avec le langage CO-OPN/2.

Finalement, la dernière partie de cette thèse illustre le bien-fondé de cette approche en l'appliquant sur une étude de cas complète et détaillée. Une application répartie Java est développée selon la méthode introduite pour le langage CO-OPN/2. Le raffinement est guidé principalement par la satisfaction de charges fonctionnelles et par des contraintes de conception intégrant la notion d'architecture client/serveur. Enfin, les étapes choisies lors du processus de raffinement de ce développement permettent d'étudier certains aspects spécifiques aux applications réparties, et de proposer des schémas génériques pour la conception de telles applications.

Abstract

One of the steps making it possible to increase the quality and the reliability of the software executing on distributed systems consists of the use of methods of software engineering that are known as formal. The majority of the formal methods currently existing correspond in fact more to formal specifications languages than to methods themselves. This is due to the fact that the two fundamental aspects which are: the logic of use of the language and the coverage of the software life cycle are not, for the majority, defined. The development by stepwise refinement is one of the means making it possible to define these two aspects.

This thesis aims to the definition of the concepts of refinement and implementation of model-oriented formal specifications. It brings a methodological base making it possible to use such a specifications language during a development by stepwise refinements and during the implementation stage.

This thesis defines, initially, a theoretical framework for the refinement and the implementation of formal specifications. The main idea consists in associating a contract with each specification. A contract explicitly represents the whole of the properties of the specification which it is necessary to preserve at the time of a refinement of this specification. To show that a concrete specification refines some abstract specification, it is then a matter of showing that the contract of the concrete specification is sufficient to ensure the properties corresponding to the contract of the abstract specification.

The second part of this thesis consists in applying this theoretical framework in the context of the CO-OPN/2 language. CO-OPN/2 is an object-oriented formal specifications language founded on algebraic specifications and Petri nets. Thus, definitions of the concepts of contracts, refinement and implementation are proposed for this language. The contracts are expressed using the Hennessy-Milner temporal logic (HML). This logic is used in the theory of test provided with language CO-OPN/2. Thus, the verification of the contractual properties, as well as the verification of the stages of refinement are facilitated. Refinement and implementation are controlled semantically by the satisfaction of the contracts; syntactically, a renaming is authorised. We specifically study the implementation using the Java programming language. We show how to specify classes of the Java programming language using language CO-OPN/2, so that the last stage of the process of refinement leads to a specification entirely built using CO-OPN/2 components

specifying Java classes. The stage of implementation in the Java language itself is thus facilitated.

The third part of this thesis shows how it is possible to practically verify that a CO-OPN/2 specification satisfies its own contract, that a stage of refinement is correctly carried out, and finally that the stage of implementation is correctly performed. These verifications are carried out using the theory of the test provided with language CO-OPN/2.

Finally, the last part of this thesis illustrates the cogency of this approach by applying it to a complete and detailed case study. A distributed Java application is developped according to the method introduced for the CO-OPN/2 language. Refinement is guided mainly by the satisfaction of functional requirements and by constraints of design integrating the concept of client/server architecture. Lastly, the stages chosen in the refinement process of this development make it possible to study aspects specific to distributed applications, and to propose generic schemas for the design of such applications.

Contents

Chapter 1

Introduction

Within software engineering techniques, formal methods provide a mathematical framework to *analyse*, *design*, *implement*, and *verify* software systems.

A typical software development process begins with the *analysis* phase that enables to characterise the client's requirements. This phase produces the *requirement specification*, that describes properties of the system to be developped. Once the requirements have been established, the *design* phase produces first an *abstract system specification* that describes an operational model (behaviour) of the system. The abstract system specification should respect the requirement specification.

One of the ways for reaching an implementation from an abstract system specification is provided by the *stepwise refinement* of formal system specifications. This technique consists of gradually transforming the abstract system specification, in order to let it take into account more and more operational constraints related to the execution environment. After a series of refinement steps, a *concrete system specification* is reached that describes an operational model of the system, and takes into account the constraints of the execution environment (programming language, execution platform, etc.). The concrete system specification should of course respect the abstract system specification and as well the requirement specification.

At the end of the design phase, the *implementation* step leads to an executable program. In the case of a design phase performed with stepwise refinement, the concrete system specification is then translated into an executable program written using a *programming language*.

During design and implementation, the *verification* step is necessary in order to show: first, that the abstract system specification is correct wrt the requirement specification; second, that every system specification, obtained during the design phase, is correct wrt the system specification that precedes it in the refinement process, and is still correct wrt the requirement specification; and, finally, that the executable program, obtained during the implementation phase, is correct wrt the concrete system specification, and wrt the requirement specification. The first and last verifications of correctness listed above are part of what is traditionally called *validation*.

Formal specifications languages allow to express requirement specifications, as well as abstract and concrete system specifications. *Property-oriented* formal specifications languages, like logical languages, are well-suited for expressing the requirement specification, but it is more difficult to use them for system specifications. Conversely, *model-oriented* formal specifications languages, like Petri nets, are well-suited for expressing system specifications, but are not well-suited for expressing the requirements.

Formal methods traditionally use a single formal specifications language for expressing both the requirement specification, and the system specifications. When the chosen formal specifications language is a logical language, the specification task is more difficult, but the verification tasks is reduced to showing logical implications. When the chosen formal specifications language is model-oriented, specifications are more easily and powerfully expressed. However, the verification task usually follows an informal way (e.g., simulation), since it is difficult to determine if the (huge) set of all possible behaviours that are represented by the specification, are possible and desired behaviours of the system.

The problem of the choice between a model-oriented and a property-oriented formal specifications language is not an easy task, since requirement specifications and system specifications are both important in the development process as noted by Pnueli:

> *"(...), even if we decide to adopt system specification as the main specification mode for large systems, there is still an important role to requirement specification. It is the best and most rigorous way to validate the correctness of the system specification."* — **A. Pnueli** [54]

In order to bring a solution to the problem of the choice between a model-oriented and a property-oriented formal specifications language, some model-oriented specifications languages have acquired a property-oriented specifications language. This is known as the *two languages framework* described, among others, by Pnueli in [54]: a logical language is used for expressing requirements, and a model-oriented language is used for describing models or implementations. In addition, the logical language is also used for translating the system specification into logical properties, and the verification task is then realized in the logical framework.

The verification that a program is correct wrt a system specification is a problem similar to the one of verifying that a system specification is correct wrt the requirement specification. Thus, the use of a logical language in addition to a programming language should help for the verification task.

In the last decades, only few attempts have been undertaken to consider the idea of integrating assertions into programs. More recently, Meyer [50] has promoted this idea, and even goes a step further. Indeed, he advocates that, in order to face the problem of correctness, every program operation (instruction or routine body) should be systematically accompanied by a pre- and a post-condition. He characterises this method:

> *"(...) as a conceptual tool for analysis, design, implementation and documentation, helping us to build software in which reliability is built-in, rather than achieved or attempted after the fact through debugging; in Mill's terms, enabling us to build correct programs and know it."* — **B. Meyer** [50]

The work presented in this thesis is performed in the framework of a model-oriented formal specifications language, called CO-OPN/2 (Concurrent Object-Oriented Petri Nets). It is an object-oriented formal specifications language, which allows concurrent and distributed systems to be described in terms of: structured Petri net, describing behaviour, and algebraic specifications describing data structures. The verification that a program correctly implements a CO-OPN/2 specification is currently realised by the means of automatically generated test cases built with logical formulae derived from the CO-OPN/2 specification. Formulae are expressed using the Hennessy-Milner branching-time temporal logic (HML), which is a very simple logic well-suited for automatically generating formulae. A series of works around the CO-OPN/2 language have considerably enriched the CO-OPN/2 framework. However, there is still a lack of a rigorous development methodology.

This thesis brings some elements useful for establishing such a development methodology. A theory of stepwise refinement and implementation of model-oriented specifications is proposed, that lies within the scope of the two languages framework, as described by Pnueli, and that uses built-in features for addressing the correctness issue, as advocated by Meyer. Indeed, this thesis proposes:

- a general theory for the stepwise refinement and implementation of model-oriented formal specifications, which advocates the use of a model-oriented language and a logical language during the whole development process and the implementation phase;
- an application of this theory to the CO-OPN/2 language, using the Hennessy-Milner logic;
- a way of practically verifying the correctness of the refinement process and the implementation phase, by using test generation.

This chapter first presents the motivations and the principle of the stepwise refinement and implementation theory. Second, it discusses the positioning of this work in the framework of the CO-OPN/2 language, and finally it outlines the main contributions.

1.1 Motivation and Principle

Traditional definitions of stepwise refinement for model-oriented specifications languages, require that the *whole* behaviour, or at least the whole observable behaviour described by a specification (in case of object-oriented languages), has to be preserved by a subsequent refinement step. Such a requirement is too strong, since, from a practical point of view, it is not realistic to require the whole behaviour to be preserved. In the case of model-oriented specifications, the behaviour of the specification explicitly describes a particular solution, and implicitly describes properties of the system. This set of properties can be split in two parts: properties that are specific to the solution provided by the specification, and essential properties required by the client. What has to be preserved during a refinement step is not the whole behaviour (and hence all the particular properties), but only the *essential* properties that make the system convenient for the client.

Then it becomes necessary to be able to make the distinction between particular properties and essential properties. Since model-oriented specifications languages cannot be used to express explicitly properties, we advocate the use of an additional logical language for expressing the properties. Specifications are then made of two parts, a model-oriented part expressed expressing the system specification, and a property-oriented part expressing the properties to preserve. We call *contract* the property-oriented part, and *contractual specification* the pair made of a specification and a contract.

The definition of refinement is divided in two parts: a syntactical part that settles syntactical rules that a concrete specification has to respect wrt a more abstract specification; and a semantical part which ensures that the contract of an abstract specification is preserved by the contract of a more concrete specification. We call such a refinement, a *refinement based on contracts.*

As already mentioned above, the idea of combining a model-oriented specification with properties expressed with a logical language is not new. Object-oriented specifications languages like TROLL and VDM^{++}, as well as some classes of timed Petri nets employ a similar technique. The use of a logical language for expressing properties enables these specifications languages to formally prove that a refinement step is correct. The set of properties used to make the proof is generally the whole set of properties satisfied by the model of the specification.

The particularity of our approach is twofold: first it goes a step further and authorises some properties to be *lost* during a refinement step. The specifier is then free to refine, provided concrete specifications preserve the contract of more abstract specifications. Second, the use of contracts explicitly joined to specifications and to programs enables to address the problem of *correctness.* The specifier must explicitly give the properties that he wants to be preserved during a refinement step. Thus, from a methodological point of view, this facilitates the building of correct specifications, since the contract points out the properties to be verified.

The ultimate goal of a stepwise refinement is to reach an implementation. It seems then natural to extend the theory of refinement based on contracts, to the implementation, more especially as programming languages do not express explicitly the properties of a system. The *implementation based on contracts* requires that a contract be added to a program, in order to form a *contractual program*, and that this contract preserve the contract of the specification to implement.

According to these principles, a general theory of refinement and implementation based on contracts has been defined for model-oriented specifications languages and logical languages. Although it is presented in a general way, this theory is mostly thought for distributed and concurrent systems. Indeed, the work presented in this thesis is conduced in the framework of the CO-OPN/2 language, which defines a class of high-level Petri nets well-suited for specifying distributed and concurrent systems.

The general theory of refinement and implementation based on contracts has been applied to the CO-OPN/2 formal specifications language; the Hennessy-Milner logic (HML) is used for expressing the contracts on CO-OPN/2 specifications. Since CO-OPN/2 is an object-oriented specifications language, the implementation of CO-OPN/2 specifications has been investigated for object-oriented programming languages; HML is used for expressing formulae on programs. Some other works on CO-OPN/2 attempt to directly implement CO-OPN/2 specifications using the Java programming language. Therefore, attention has been given to refinement processes ending with an implementation phase using Java. In order to further built a development methodology using CO-OPN/2, the correctness issue has been considered under the semantic approach: automatically generated tests are used for verifying the contracts preservation.

1.2 Positioning

Active research is currently being conduced in the CO-OPN/2 framework. The following points summarise some past, present and future works on CO-OPN/2:

- *The CO-OPN/2 Formal Specifications Language*
  The CO-OPN/2 language, presented by Biberstein [14], is an object-oriented formal specifications languages based on Petri nets and algebraic specifications. This language allows the definition of active concurrent objects dynamically created, and includes facilities for sub-typing, and sub-classing;

- *Strong Refinement*
  The current definition of refinement of CO-OPN/2 specifications, due to Buchs and Guelfi [21], is based on the bisimulation equivalence. A more concrete CO-OPN/2 specification refines a more abstract CO-OPN/2 specification if the transition system of the former, restricted to the elements of the latter, is bisimulation equivalent to the transition system of the latter. Bisimulation equivalence requires that the transition systems have the same branching structure;

- *Incremental Prototyping Methodology*
  Hulaas [43] describes first a tool for compiling CO-OPN/2 specifications into an abstract distributed implementation, and second a manual optimisation of the abstract implementation, in order to reach a concrete implementation [43]. The possibility to directly implement CO-OPN/2 specifications in Java is currently being studied.

- *Automatic Test Generation*
  Barbey, Buchs and Péraire [12] define a theory of test generation for CO-OPN/2 specifications. This theory enables to derive, from a very large set of test cases, a reduced set of test cases, which is still fully representative of the specifications behaviour. Péraire [52] has completed this theory with a tool able to automatically generate the reduced set of test cases built with HML formulae;

- *Towards an Axiomatic Semantics for CO-OPN/2*
  Inference rules for computing all valid transitions are defined for CO-OPN/2 by Biberstein [14]. In addition, Vachon in [59] defines inference rules for computing all invalid transitions. Given these sets of rules, Buchs and Vachon [59] currently study how to obtain a complete axiomatic semantics for a subset of CO-OPN/2;

- *Contextual Coordination*
  Buffo [22] defines a contextual coordination model for distributed object systems and defines COIL, that is a language for the contextual coordination of CO-OPN/2 specifications. The model provides: coordination structures, by means of hierarchies of contexts and objects; and dynamic configurations, by means of object migrations, useful when the architecture of the distributed system dynamically changes;

- *Tools*
  Co-opnTools [24] is a project aiming at developing a set of tools dedicated to the visualisation, edition, and simulation of graphical and textual CO-OPN/2 specifications. Among others, we can mention **Co-opnCheck**, which is a tool able to verify that a CO-OPN/2 specification has a correct syntax and static semantics. **Co-opnTest** is a tool for automatically generating test cases [52]; it contains an editor for graphically viewing CO-OPN/2 textual specifications as well. A viewer and a simulator of CO-OPN/2 specifications are currently being studied. A former tool, called **TTool** automatically transforms CO-OPN/2 specifications into highly-parallelised CO-OPN/2 specifications [20].

The series of works mentioned above have contributed to first establish the CO-OPN/2 language, and second to enrich the language with theories and tools essentials to a practical and industrial use of the CO-OPN/2 language. However, the CO-OPN/2 framework still lacks of elements like: formal proofs for asserting that a formula is satisfied or not by the model of a CO-OPN/2 specification; a methodology of development and a tool for it; a graphical simulator.

This thesis is a first step towards the establishment of a development framework, both theoretical and practical, for CO-OPN/2. Figure 1.1 shows the theoretical basis of such a development framework. After the analysis phase, informal requirements are determined. On the basis of these requirements, an abstract contractual CO-OPN/2 specification $(\text{Spec}_0, \text{Contract}_0)$ is devised, whose contract formally expresses the requirements. During the design phase, several refinement steps are performed, that finally lead to a concrete contractual CO-OPN/2 specification $(\text{Spec}_n, \text{Contract}_n)$. The implementation phase then provides the contractual program (Program,Contract). The verification of correctness uses generated tests for: verifying that the model of a specification actually satisfies its contract, and in a similar way for the program (horizontal verification); verifying that a refinement step is correct (vertical verification); and finally, verifying that a program is a correct implementation (program verification). Besides this semantic approach to correctness, the refinement and implementation based on contracts can be used, in the future, to perform axiomatic verification on the basis of the axiomatic semantics being currently developped for CO-OPN/2. Moreover, future work could provide a compositional notion of refinement based on COIL components.

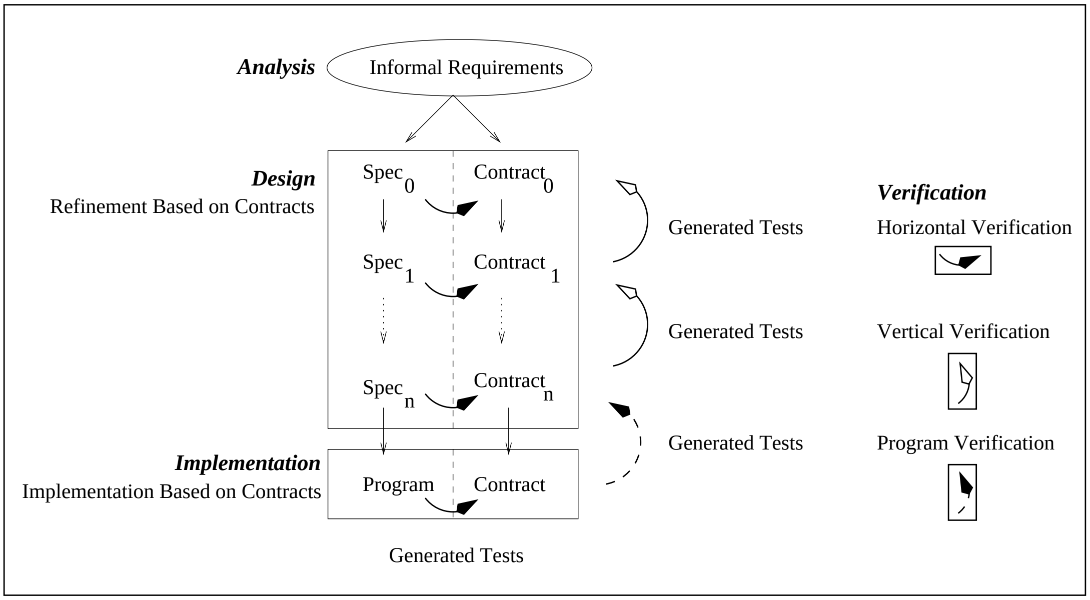


Figure 1.1: A Development Framework For CO-OPN/2

1.3 Contribution

The results presented in this thesis, and which contribute to the establishment of a development framework for CO-OPN/2 as explained above, can be split into three categories: first, a *general theory* of stepwise refinement and implementation based on the use of contracts; second, the *application* of these theories to the CO-OPN/2 language, in order to provide a theory of stepwise refinement and implementation of CO-OPN/2 specifications; and third, a *development methodology* for CO-OPN/2 which provides more particularly a development method of Java applications, and which uses test generation in order to perform verifications. The contributions of this thesis are as follows:

- *A General Theory of Stepwise Refinement Based on Contracts*
  The theory of stepwise refinement based on contracts is defined for model-oriented specifications. It advocates the joint use of a specification and a set of logical formulae, called a contract, satisfied by the model of the specification. A refinement

step is correct if the contract of a concrete specification preserves the contract of a more abstract one;

- *A General Theory of Implementation Based on Contracts*
  The theory of implementation based on contracts is defined in a way similar to that of refinement: a set of logical formulae, satisfied by the model of the program, is added to the program; the program correctly implements a specification if the program contract preserves the specification contract;

- *A Theory of Stepwise Refinement of CO-OPN/2 Specifications*
  The theory of refinement based on contracts is applied to the CO-OPN/2 specifications language. The Hennessy-Milner logic is used to express contracts on CO-OPN/2 specification. This logic is currently used in the framework of CO-OPN/2 for automatically generating test cases. The choice of this simple logic for expressing contracts is motivated by the will to further automate the proof that a refinement step is correct, using automatically generated test cases;

- *A Theory of Implementation of CO-OPN/2 Specifications*
  The theory of implementation based on contracts is applied to the CO-OPN/2 specifications language and to object-oriented programming languages. An abstract definition of object-oriented programs is provided, and HML formulae are defined on these programs;

- *Implementation of CO-OPN/2 Specifications in Java*
  The implementation of CO-OPN/2 specifications using the Java programming language is more particularly studied. The implementation step is trivially realized if the most concrete CO-OPN/2 specification reached at the end of a refinement process is very close to the Java program. By close, we mean that every instruction of the program is specified, and that the behaviour of the CO-OPN/2 specification and that of the Java program are the same. We show how to obtain a CO-OPN/2 specification which specifies a Java program and reflects the Java semantics. Advices are given on how to conduct a refinement process in order to easily perform the implementation step when the Java programming language is used;

- *Verification of Refinement and Implementation Using Test Generation*
  It is shown how test generation is used in order to practically verify that a set of formulae is actually a contract for a given CO-OPN/2 specification, that a refinement step is correctly performed, and that the implementation phase is correctly realized.

1.4 Document Organisation

Chapter 2 is made of two parts: a survey of some definitions of refinement for model-oriented specifications languages; and an analysis of these definitions, that enables to conclude that every definition of refinement can be reduced to the preservation of a set of properties.

Chapter 3 defines the general theory of stepwise refinement and implementation based on contracts, it gives some compositional results, and discusses the approach.

We intend to use this theory in order to define the formal refinement of CO-OPN/2 specifications. Therefore, Chapter 4 presents the syntax and semantics of CO-OPN/2 specifications.

Chapter 5 presents the Hennessy-Milner logic for expressing contracts on CO-OPN/2 specifications, and defines the theory of refinement based on contracts for the CO-OPN/2 specifications language. It defines as well a hierarchical operator on contractual CO-OPN/2 specifications, and a compositional refinement.

Chapter 6 applies the theory of implementation based on contracts to the CO-OPN/2 specifications language and object-oriented programming languages. In addition, it defines the compositional implementation of CO-OPN/2 specifications.

Since we are more particularly interested in implementations realized with the Java programming language, Chapter 7 explains how Java programs can be specified using the CO-OPN/2 specifications language, and gives some hints on how to conduct a refinement process in order to reach easily a Java program.

In the CO-OPN/2 framework, the Hennessy-Milner logic is used for expressing automatically generated tests. Chapter 8 shows how it is possible to use test generation in order to prove first that the transition system of a CO-OPN/2 specification satisfies a set of HML formulae, and second that refinement steps and implementation phase are correctly realized.

Through a concrete case study, Chapter 9 realizes the complete development of an application: starting from informal requirements, a refinement process ended by a Java implementation, is performed and informally proved.

Finally, Chapter 10 gives a summary of the principal results of this thesis and lists some future works.

Chapter 2

Related Works

The purposes of this thesis are first, to provide a formal definition of stepwise refinement of model-oriented specifications, that is based on the use of an additional logical language; and, second, to apply this definition to the CO-OPN/2 language, which is object-oriented and based on Petri nets and algebraic specifications. This chapter gives an informal description of some of the definitions of stepwise refinement that can be found in the areas of Petri nets, and object-oriented specifications. In order to complete this overview of definitions of refinement, we present also other definitions, which either are independent of a specific formalism, or make use of a logical language.

Once we have reported these definitions, we compare them from several points of view: syntactical obligations of the definition of refinement, e.g., preservation or not of the signature; semantical obligations of the definition of refinement, e.g., input/output behaviour preservation or trace behaviour preservation. As we are interested in systems having models based on events and states, emphasis will be given to refinements of such systems, rather than to functional systems. Then we devise the properties that a refinement must have and those that it may have. We discuss what should be the difference between an implementation and a refinement; and give some hints on development methodologies. Finally, we show how most of these definitions can be captured by a more "generic" definition, based on the preservation of observable properties of interest. This definition of refinement is informally stated at the end of this chapter. It is the core of this thesis; it is formalised for specifications in general in chapter 3, and applied to the CO-OPN/2 language in chapter 6.

In the rest of this chapter, we use as synonyms the terms: abstract specifications and high-level specifications; and the terms concrete specifications and low-level specifications. A concrete or low-level specification stands for the refinement of an abstract or higher-level specification. We also say that an element is abstract or concrete if it belongs to the abstract or to the concrete specification respectively. Moreover, we will report below diverse definitions of refinement, using the same words as the authors. For this reason, a given word may have a different meaning in two different definitions of refinement. This is particularly the case for the word "implementation"; either it is used as a synonym to refinement, or it has its own, different, meaning.

2.1 Refinement of Petri Nets/High-level Nets

This section presents some (of the numerous) definitions of refinements for different kinds of Petri nets. First, we introduce some refinements of unstructured Petri nets. These refinements usually rely on embedding techniques, such as the replacement of a transition by a subnet, or the replacement of a place by a subnet. These techniques ensure either that the initial net and the refined net have the same properties, or that two equivalent nets, refined in the same way, lead to two equivalent nets. A survey of equivalence notions for Petri nets, due to Pomello *et al.*, can be found in [55]. Second, we introduce an example of refinement of a kind of timed Petri nets based on the preservation of observable properties. Third, we give two different definitions of refinement in the framework of structured nets. Finally, a general definition of replacement of a subnet by another subnet is given. This definition can be applied to several kinds of Petri nets.

2.1.1 Refinement of Unstructured Petri Nets

The techniques for refining unstructured Petri nets are based on the replacement of a transition or a place by a subnet. These techniques differ in the way the subnet is embedded inside the initial net. Moreover, some of these techniques ensure that the initial net and its refinement have the same properties (they are equivalent in some sense). Some other techniques ensure that, given an equivalence relation, two equivalent nets are refined to two equivalent nets. According to the terminology used in the literature: if the equivalence relation and the refinement operation are such that two equivalent nets refine to two equivalent nets, then we say that the equivalence relation is a *congruence* wrt the refinement operation. The first technique (a net refines to an equivalent net) is used when both the original net and its refinement have the same behaviour. The second technique (two equivalent nets refine to two equivalent nets) is used when the refinement introduces new elements, such that the original nets and their respective refinements have different behaviours.

We now introduce four definitions of refinements: the first two ensure that the refined net preserves some properties of the original net, i.e., they are equivalent; and the last two ensure that two equivalent nets are refined to two equivalent nets.

Refinement of a Transition

The survey of Brauer *et al.* [19] lists several refinements for unstructured Petri nets. Among others, it describes the refinement of a transition t by a refinement net. A refinement net D, which refines a transition t, is a net that has some initial transitions, representing the beginning of t, and some final transitions, representing the end of t. The refined net is obtained by replacing the transition t by the refinement net, and by connecting each place in the preset of t with every initial transition of D, using an arc that has the same weight as the original arc between the place and t. Similarly, each place in the postset of t is connected with every final transition of D. This technique ensures that if the original net is safe (live or bound) and if D is also safe (live or bound), then the refined net is safe (live or bound).

Refinement of Places via Parallel Composition

Vogler [60] defines the refinement of a place by a refinement net. A refinement net D, which refines a place p, in a net N, via parallel composition, is a net that has some transitions labelled as the transitions adjacent to p. The parallel composition consists in splitting up the transitions of N, adjacent to p, such that each split transition is merged with every transition of net D with the same label. The refined net is obtained by parallel composition of the net N where place p has been replaced by D. This technique ensures under certain conditions that net N and its refinement have the same failure semantics. A dual approach exists for the refinement of transitions.

Action Refinement

Also taken from Brauer *et al.* [19], the action refinement consists in replacing *every* transition with some given label by a copy of the same refinement net. This technique ensures that the process equivalence, and the failure equivalence are congruences wrt this refinement. Two nets are process-equivalent if they have the same underlying process; they are failure-equivalent if they have the same set of failures. For instance, in the case of process equivalence, two nets, with the same underlying processes, refined by two process-equivalent refinement nets, lead to two nets with the same underlying processes.

Replacement of a Transition by a Net Modulo a Function

Best and Thielke [13] define a refinement for coloured Petri nets. This refinement is based on the idea that the replacement of a transition t, of a net N_1, by a subnet N_2 affects the environment of t: the type (set of colours) of the places before and after t will change in the refined net (after replacement) as well as the type (i.e., occurrence mode) of the transition corresponding to t and the labels of the arcs. In order to be able to insert the subnet N_2 into the net N_1, a *function* is needed. This function is a mapping from the places of N_2 to the set $\{e, i, x\}$. The places mapped to e, meaning entry, are to be combined with the places in the preset of t, the places mapped to x, meaning exit, are to be combined with the places in the postset of t, and the places mapped to i, meaning internal, are new places not related to a place of N_1.

The refinement is conducted in several steps. The places of N_1 that are in the preset and postset of t are merged with the places of N_2 mapped to e and x. The type of this new place is a combination (the set of all sums of multisets) of the types of the places of N_1 and those of N_2. The transition t is merged with all the transitions of N_2 adjacent to places mapped to e and x. The type of this new transition is the set of all sums of the types of t with every transition adjacent to places mapped to e and x. An arc links the new place to the new transition: its label stands for all the possible combinatorial ways of removing values when firing the merged transitions. Similarly, an arc links the new transition to the new place: its label stands for all the combinatorial ways of adding values when firing the merged transitions. Some more arcs link the new place to transitions of N_1 and the new transition to the internal places of N_2.

The transformation equivalence is a congruence wrt this refinement. Two nets, N_1 and N_1', are transformation-equivalent if they lead to the same net after having isolated the transition to be replaced, and merged its adjacent places. Two subnets, N_2 and N_2', are transformation-equivalent if they lead to the same net after having merged all the places mapped to e and x and merged their adjacent transitions. This technique ensures that if a net N_1 is refined by a subnet N_2 and if a net N_1', transformation-equivalent to N_1, is refined by subnet N_2', transformation-equivalent to N_2, then the two refined nets are still transformation-equivalent.

In addition, this technique is commutative modulo this equivalence, i.e., first replacing t_1 by net N_2 and then t_2 by net N_3 is equivalent to replacing first t_2 and then t_1. A dual definition can be given for the replacement of a place.

A similar definition of refinement for M-nets, a high-level class of Petri nets, has been given by Devillers *et al.* [29].

2.1.2 Refinement of Timed Petri Nets

We present now an interesting approach concerning the refinement of timed Petri nets based on the use of a temporal logic. TRIO is a linear, first-order typed temporal logic due to Ghezzi *et al.* [39]. A TRIO axiomatisation, due to Felder *et al.* [34], has been given to a kind of timed Petri nets where each transition is associated with a firing time interval describing its earliest and latest firing time after enabling. A transition consumes exactly one token from each place in its preset, and produces exactly one token into each place in its postset. At a given time a transition may fire several times.

The TRIO axiomatisation of these timed Petri nets is based on two predicates: $nFire(v, n)$ means that, at the current time, transition v fires n times, and $tokenF(s, i, p, v, j, d)$ means that, at the current time, the i^{th} firing of transition s produces a token that enters place p, this token is consumed after d time units by the j^{th} firing of transition v. Given a net N, a set of axioms $Ax(N)$ is built, that take into account the net and its initial marking. From $Ax(N)$ a theory is derived, noted $\mathcal{N}$. On the basis of the two above predicates and arithmetic operators, formulae can be expressed over the net. If a formula ϕ can be derived from the theory $\vdash_{\mathcal{N}} \phi$ then *every* execution of the net satisfies the property ϕ.

The implementation relation, of Felder *et al.* [33], of a net S by a net I, is based on the *preservation* of *observable* properties. A net I implements a net S if the observable properties of S are also observable properties of I after translating them into I. The only observable events in a net are transition firings. Therefore, an observable property ϕ, of a net S, is a formula constructed on the basis of the firing predicate $nFire(v, n)$ only, and must be derived from $\mathcal{S}$ (the theory of S): $\vdash_{\mathcal{S}} \phi$.

During a refinement step, it is possible to refine a transition by several transitions (not just one). An event function, $\lambda : T_I \rightarrow T_S$, maps transitions of I to transitions of S. The event function may be partial (a transition of I has no corresponding transition in S), has to be surjective (every transition in S must have at least one corresponding transition in I, so that every observable property of S can be translated into an observable property of I). The event function may be non-injective: a transition in S may be associated to several transitions in I.

Given an event function λ, a property function, $\Lambda : \mathcal{S} \rightarrow \mathcal{I}$ is univocally derived. It translates properties of the theory $\mathcal{S}$, of S, to properties of the theory $\mathcal{I}$, of I. The translation is based on the translation of the firing predicate:

$$\Lambda(nFire(v, n)) = \exists n_1 \ldots \exists n_s (n_1 + \ldots + n_s = n \wedge nFire(v_1, n_1) \wedge \ldots \wedge nFire(v_s, n_s))$$

where $\{v_1, \ldots, v_s\}$ is the set of all transitions of I mapped to v ($\lambda(v_i) = v, 1 \leq i \leq s$). The predicate that asserts that transition v fires n times is translated into a predicate that says that the sum of firings of the transitions of I mapped to v is also n.

A net I implements a net S through λ iff for each observable formula ϕ of S:

$$\vdash_{\mathcal{S}} \phi \Rightarrow \vdash_{\mathcal{I}} \Lambda(\phi).$$

Every observable formula of S is translated into an observable formula of I.

In addition, Felder *et al.* [33] give a method for proving implementation. It is based on the idea that for each observable property ϕ of a net S there exists in the axiomatisation of the implementation net I a proof of $\Lambda(\phi)$ that mirrors the proof of ϕ. They give also some refinement rules that ensure a correct refinement.

2.1.3 Refinement of Structured Petri Nets

In the field of structured Petri nets, a small number of definitions have been given. We mention two of them. The first is based on method calls, and the second is based on the preservation of the bisimulation equivalence.

Refinement as a Method Call

Kiehn [45] considers that if a transition t of a net N is refined by a subnet N', t is not statically replaced by N', but the *firing* of t is replaced by a *call* to N'. In the refined net, the firing of t leads to the initial marking of N', once N' reaches a final marking, control is given back to N, i.e., the tokens produced by the firing of t are inserted into the places of the postset of t. This definition of refinement is based on a structuring technique: a refinement is achieved when more structure is added to the original net. In addition, this technique aims at deriving the behaviour of the refined system from the behaviour of N and that of N'.

CO-OPN

CO-OPN is an object-based specifications language due to Buchs and Guelfi [21]. An object is an algebraic Petri net able to synchronise with another object. Objects have an external and an internal part. The external part is made of special transitions called methods that are used for the synchronisation. The internal part is made of transitions and places. It cannot be accessed by other objects. A method can fire only if the synchronisations it requires with the methods of other objects is possible, i.e., if these methods can fire simultaneously. The firing of a method is atomic (i.e., it occurs entirely or not at all). The semantics is a step semantics (several methods may fire simultaneously). It is given by a transition system taking into account an algebra (a model) for the algebraic specification part.

Two kinds of refinements, based on the preservation of the bisimulation equivalence, are defined: *object replacement* and *algebra replacement*. Given two CO-OPN specifications S_1 and S_2, and their corresponding transition systems TS_1 and TS_2, a bisimulation is a relation over states such that, if a state m_1 of TS_1 is in relation with a state m_2 of TS_2, then: (1) for every transition of TS_1, which transforms m_1 into a new state m_1', there is a transition of TS_2 with the *same event* that transforms m_2 into a state m_2', and m_1' is in relation with m_2'; (2) vice-versa for the transitions of TS_2 transforming m_2. In addition, the initial states (initial markings) must be in relation.

Given an algebra A of the algebraic specification, the object replacement consists in replacing a sub-specification by a bisimular sub-specification. The transition system of the whole initial specification must be bisimular to the transition system obtained after the replacement.

A transition system of a CO-OPN specification is given with an algebra A_1 for the algebraic specification. The algebra refinement consists in replacing the algebra A_1 by an algebra A_2, which is another model of the same algebraic specification, in the transition system of the CO-OPN specification. The new transition system obtained must be bisimular to the initial one.

2.1.4 Abstract Definition of Refinement for Petri nets

We now introduce an abstract definition of refinement for Petri nets, based on category theory, that encompasses technical definitions of refinement for several kinds of Petri nets. This refinement, due to Padberg [51], is called rule-based refinement. It considers the refinement as a production rule $p = (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R)$, where L, K, R are nets (objects in a category of nets), and l, r are morphisms. The meaning of the production rule is the following: the parts of the net L that are not in the image of K by l are deleted and they are replaced by the parts of the net R that are not in the image of K by r. K stands for a "common" part to keep. The particular case where K is empty leads to the replacement of the whole net L by the whole net R. K is actually a common part of both L and R when l, r are identities. The rule is applied to a net N where L is part of the net and produces a net M where $l(K)$ (a part of L) has been replaced by $r(K)$ (a part of R). The net N is said to be *transformed* to net M.

This theory has been applied to several kinds of Petri nets, among others: place/transition nets, algebraic high-level nets, predicate/transition nets, coloured nets. In the case of algebraic nets, the morphisms map places to places, transitions to transitions and there is a morphism from the algebraic specification of a net to that of the other. In addition the morphism between algebraic nets must be compatible with the pre- and post-conditions. By its abstractness, this technique generalises several notions of refinements for several kinds of Petri nets.

In addition, it ensures that: under certain conditions (independence), two transformations are commutative (they lead to the same object); parallel transformations (component-wise application of two transformations) can be viewed as a sequence of transformations and vice-versa. Moreover, horizontal structuring (fusion, union) is compatible with transformations. Fusion removes multiple copies of the same item, while union glues together two nets by a shared subpart. If we make first a transformation of net G and then we fusion the resulting net H, we obtain the same object as if we first make a fusion of G and then apply the transformation. If we make the union of two nets and then we apply a parallel transformation, we obtain the same object as if we first transform each net separately, and then make their union.

2.2 Refinement of Object-Oriented Specifications

Object-oriented specifications have visible parts and hidden parts. They define attributes, object identifiers, states and methods. The refinement of object-oriented specifications deals with problems like: the preservation or not of the visible parts, the management of object identifiers, the transformation of the attributes, the transformation of the state, and the preservation of the behaviour. This section reports the refinement of FOOPS, TROLL, and VDM^{++} specifications.

2.2.1 FOOPS

FOOPS, reported by Borba and Goguen in [17], is a concurrent object-oriented specifications language having an operational semantics. The FOOPS language clearly distinguishes between data elements and objects: a functional level is used to describe abstract data types (ADTs) and an object level is used to describe classes of objects. The functional level is a variant of OBJ defined by Goguen [40]. It enables to define sorts, sub-sort relations, operations, and properties the operations have to satisfy. The object level enables to define modules, i.e., sets of classes of objects with visible and hidden methods and attributes (state values), object identity, dynamic object creation and deletion, overloading, polymorphism, inheritance with overriding. Attributes are defined as operations from an object identifier to a value. Attributes are inquiry operations: they do not update the state of an object, they only return the value of the state. Methods are updating operations associated to an attribute. Their behaviour is specified with axioms indicating the new value for the attribute to be updated. The evaluation of a method is atomic unless the method behaviour is specified in terms of other operations using method combiners. A specification is a module.

The definition of refinement in FOOPS, due to Borba and Goguen [18, 16], is based on the notion of *experiment* and (P,Q)-*simulation* of a state by another state. An experiment is the invocation of a *visible* operation with arbitrary arguments (object identifiers and elements of ADTs). A visible operation is a visible attribute, a visible method, or an object creation and deletion routine. Informally, "a state P is simulated by a state Q if whatever can be observed by performing experiments with Q can also be observed by performing the same experiments with P." In other words, "we cannot detect whether Q or P is being used." This implies that all experiments feasible with P must be feasible with Q and must yield the same results. However, Q may allow more experiments than P.

The operational semantics of a FOOPS specification P is given by a transition relation $\rightarrow_{\mathtt{P}} \subseteq Conf(\mathtt{P}) \times Conf(\mathtt{P})$, where $Conf(\mathtt{P})$ is made of all pairs (e, P), e an expression, i.e., a composition of experiments, and P a state.

Given two FOOPS specifications P and Q, such that all experiments and object identifiers of P are also experiments and object identifiers of Q, and ADTs of P, restricted to primary sorts (sorts needed for experiments), are ADTs of Q:

- a (P,Q)-simulation is a relation $S \subseteq Conf(\mathtt{P}) \times Conf(\mathtt{Q})$ such that $(P, Q) \in S$ implies: (1) that any state *immediately* reached from Q is related to some state that might *eventually* be reached from P; (2) if the expression in Q cannot be further evaluated then the expression in P might eventually reach the same situation and the resulting state is related to Q by S. The results of the evaluation of expressions in Q might eventually be observed in a state reachable from P; (3) performing the same experiment in Q and P leads to states related by S, thus they yield the same result;

- a state Q refines a state P, noted $P \sqsubseteq_{(\mathtt{P},\mathtt{Q})} Q$, if there is S a (P,Q)-simulation such that $(P, Q) \in S$;

- an expression q refines an expression p, noted $p \sqsubseteq_{(\mathtt{P},\mathtt{Q})} q$, if there is S a (P,Q)-simulation such that $(\langle p, \varnothing_{\mathtt{P}}\rangle, \langle q, \varnothing_{\mathtt{Q}}\rangle) \in S$, where $\varnothing_{\mathtt{P}}$ stands for the initial state of P, and $\varnothing_{\mathtt{Q}}$ stands for the initial state of Q. The refinement of an expression is a congruence wrt FOOPS combiners: e.g., $p \sqsubseteq_{(\mathtt{P},\mathtt{Q})} q$ implies $p \,||\, o \sqsubseteq_{(\mathtt{P},\mathtt{Q})} q \,||\, o$, where $||$ is the parallel operator between expressions;

- finally a specification Q refines a specification P, noted P $\sqsubseteq$ Q, if every experiment of P is refined by the *same* experiment in Q.

To summarise: a specification Q refines a specification P if syntactically and semantically several conditions hold. Syntactically: (1) all visible methods and attributes of P are also visible methods and attributes of Q; (2) the ADTs of P restricted to the primary sorts are also ADTs of Q; (3) the object identifiers of P are also object identifiers of Q. This is necessary in order to be able to perform in Q the *same* experiments as in P. Semantically: all experiments of P must be experiments of Q, and the results (new reachable states, or end states) obtained when performing these experiments in Q are related to results that can be obtained when performing these experiments in P. This definition of refinement allows data refinement (states are abstracted by the means of observations, i.e., experiments) as well as action refinement (refinement of expressions). Refinement is achieved by the reduction of non determinism, and the introduction or the removal of stuttering steps (sequences of the same state are allowed in a trace).

2.2.2 Troll

Troll, reported by Denker and Hartel in [28], is an object-oriented specifications language with a denotational semantics based on event structures. A Troll *object* is a unit of structure described by its attributes (local state), actions and axioms (behaviour). The axioms describe the effects of actions on attributes, the enabling conditions for actions, and the communication structures between objects. A Troll *system* is a community of concurrently existing and communicating objects. In a system, several objects as well as their interactions: concurrent composition and synchronous communication (action calling) may be defined.

Every object has a behaviour represented by the set of all possible runs. A run is called a *sequential life cycle*; it is a sequence of local actions of the object. The model of an object is a labelled sequential event structure, i.e., a rooted tree where each branch of the tree is a sequential life cycle and each branching point is an alternative behaviour.

The behaviour of a Troll system is given by the set of all system runs. A system run is called a *distributed life cycle*. It consists of the sequential life cycles of each objects belonging to the system (one life cycle per object) glued together at communication points.

When the objects communicate, they share an event in their life cycles and perform a synchronous action. The semantics of a TROLL system is also given by an event structure.

The refinement of TROLL systems, due to Denker [27], is guided by the idea of integrating database aspects into a refinement theory for object-oriented specifications. The fundamental idea is the following: a TROLL action is refined (reified, in the TROLL terminology) to a *transaction* (a sequence of concrete actions). The correctness criterion, which forces the sequential execution of two abstract actions to be reified only by the sequential composition of the corresponding transactions, is considered to be too strict. For this reason, the sequential composition of transactions is liberalised such that independent concrete actions, i.e., actions which are not accessing the same resources, may be *interleaved* arbitrarily and do not have to wait for each other.

More precisely, to every distributed life cycle of a concrete TROLL system is associated a set of all *sequential schedules*. This set is obtained by interpreting concurrency between sequential life cycles as an arbitrary order. Over the set of all sequential schedules of all distributed life cycles is defined an equivalence relation partitioning this set into equivalence classes such that: two schedules are equivalent if they have been derived from the same distributed life cycle, i.e., they can be considered as two correct interleaved sequences of the same distributed life cycle. The number of equivalence classes is less or equal to the number of distributed life cycles. Finally, a concrete event structure refines an abstract event structure if there is a surjective map from the equivalence classes of the concrete event structure sequential schedules to the set of all distributed life cycles of the abstract event structure. This means that: (1) there is no behaviour in the refined model which does not correspond to some abstract behaviour; (2) the entire behaviour of the abstract system is represented in the concrete model. The concrete runs can be characterised as equivalence classes of sequential schedules. It is only necessary to have at least one equivalence class in the refined model for any abstract concurrent system run.

Besides this database driven aspect of reification, temporal logic issues related to the above semantic refinement have been investigated by Huhn, Wehrheim and Denker [26, 42]. In this approach, a system specification is a pair $SysSpec = (\Sigma, \Phi)$, where $\Sigma = (Id, Att, Ac)$ is a triple made of Id, a set of object identifiers, Att, an Id-indexed set of attributes, and Ac, an Id-indexed set of actions. The set Φ is an Id-indexed set of formulae. This set is derived from the specification by translating each TROLL concept to an appropriate temporal formula. This set of formulae establishes all the possible runs of the systems. The signature Σ is constructed on top of a data signature.

Given two system specifications: $SysSpec^{Abs} = (\Sigma^{Abs}, \Phi^{Abs})$ and $SysSpec^{Ref} = (\Sigma^{Ref}, \Phi^{Ref})$, $SysSpec^{Ref}$ refines $SysSpec^{Abs}$ if there is a total reification function $\rho : \Sigma^{Abs} \to \Sigma^{Ref}$, mapping identities to identities, attributes to attributes and actions to actions or transactions, such that:

$$\forall \phi \in \Phi^{Abs} \; : \; \Phi^{Ref} \Rightarrow \rho(\phi)$$

where $\rho(\phi)$ is the extension of the reification function to formulae over Σ^{Ref}.

This notion of refinement ensures that there exists a mapping from abstract signatures to reified signatures, such that the reified system models at least the behaviour of the abstract system (the reified system has more formulae than the abstract system).

2.2.3 VDM^{++}

VDM^{++}, due to Lano [47], is an object-oriented specifications language. A VDM^{++} class defines: (1) a data part with data types, constants and functions; (2) attributes of the class (including identifiers of instances); (3) invariants of the attributes; (4) initial states of the attributes; (5) update methods (changing the attributes); (6) inquiring methods (returning a result without changing the attributes); (7) a `sync` clause describing either an explicit history of an object, or a set of permissions restricting the conditions under which methods can be invoked; (8) a `thread` clause describing allowed execution paths. Methods are defined with pre- and post-conditions.

The definition of refinement is based on the following idea: "If D is a refinement of C, it must not be possible for a user of the common interface to be able to devise an experiment which would allow him to deduce whether he had an instance of C or of D." This implies the following: D must not remove functionality of behaviour from C, and D can add new methods only if the behaviour of the new methods can be described as a combination of the behaviour of methods of C.

More precisely, D refines C if there is a retrieve function R from the attributes of D to those of C, and a renaming ϕ of the visible methods of C to those of D. The retrieve function R and the renaming function must satisfy several conditions: (1) every attribute of C, satisfying the invariant, must be related to an attribute of D, satisfying the invariant (*adequacy* condition); (2) initial and invariant constraints must be compatible; (3) a method $\phi(m)$ of D can be used every time the corresponding method m of C is used (weaker pre-condition in D); (4) the method $\phi(m)$ of D must lead to the same conclusions when used in the same conditions than the corresponding method m of C (stronger post-condition); (5) the renaming ϕ must be total (every method of C is refined by a method in D), ϕ can be non-injective (two methods of C can be refined by the same method in D), and ϕ can be non-surjective (new methods can be introduced in D) provided that these new methods can be expressed (via R) with methods of C. Semantical conditions are required on method executions: every possible behaviour (trace) of C must be a (possibly renamed) behaviour of D; and every trace possible for D corresponds to a trace possible for C.

For each class C a logical RTL (Real Time Logic) language $\mathcal{L}_C$ is defined, and a theory Γ_C expressing the semantics of C in this language is given. Similarly for D, a theory Γ_D is given. The refinement is defined on the basis of these theories. D refines C via R and ϕ, noted $C \sqsubseteq^{ref}_{\phi,R} D$, if:

$$\forall \psi \in \mathcal{L}_C, \Gamma_C \vdash \psi \ \Rightarrow \ \Gamma_D \vdash \phi(\psi[R(v)/u]).$$

The translation in D of every formula that is true in the theory of C leads to a formula that is still true in the theory of D. The translation of a formula in C consists in replacing each attribute of C appearing in the formula by the corresponding expression of D (built with attributes of D) given by the retrieve function, and by renaming the methods using ϕ.

Composition of VDM^{++} refinement is obtained in the following way: if a class D is a client of a class C, and C_1 refines C, then substituting C_1 for C in D produces a class D_1 which refines D.

An *implementation* class is a class that is directly translatable into a procedural language, and which has no abstract type. Translation rules allow to implement VDM^{++} specifications into programs written in procedural languages. Testing is used to assert the correctness of the implementation.

2.3 Still Other Refinement Notions

This section describes some refinements that either discuss some aspects also considered in this thesis, or are not defined for a specific formalism, i.e., they can be applied to any system independently of the specification formalism used. First, we consider algebraic specifications. Second, we introduce the ASTRAL language, which specifies real-time systems. Third, we discuss the B method, which views a system as an abstract machine. Fourth, we report the refinement calculus, where programs are predicate transformers and refinements are given by order relations. Fifth, we describe the Temporal Logic of Actions, which defines a system with a next-state relation, and verification of refinement reduces to verification of implications. Finally, we report a definition of refinement that expresses a refinement as a property and vice-versa.

2.3.1 Refinement of Algebraic Specifications

An algebraic specification is a pair $SP = \langle \sigma, E \rangle$, where $\Sigma = \langle S, F \rangle$ is a signature (sorts and operations), and E is a set of equations on the operations of the signature. A Σ-algebra A consists of an S-sorted family of non-empty carrier sets $\{A_s\}_{s \in S}$ and of a total function $f^A : A_{s_1} \times \ldots \times A_{s_n}$ for each $f : s_1 \times \ldots \times s_n \in F$. $Alg(\Sigma)$ is the set of all Σ-algebras. A model of SP is a Σ-algebra A satisfying the formulae of E. $Mod(SP)$ is the set of all models of SP. There are several notions of refinement for algebraic specifications, they are based on the inclusion of the models. These definitions may be applied to algebraic specifications but also to specifications in general.

Wirsing [61] defines the refinement of a specification SP by a specification SP' by the inclusion of the models of the latter in the models of the former, i.e.,

$$Mod(SP') \subseteq Mod(SP).$$

It is noted $SP \rightsquigarrow SP'$. This implies that both specifications have the same signature. There is a diminution of the number of models when more design decisions are taken, i.e., when more formulae are satisfied. For parameterised specifications, if $P \leadsto P'$ and $SP \leadsto SP'$ then $P(SP) \leadsto P'(SP')$.

A version, due to Sannella and Tarlecki [57], allows to change the signature. It uses the notion of *constructor*. A constructor κ is determined by a function $f_\kappa : Alg(\Sigma') \to Alg(\Sigma)$ on algebras. The constructor κ transforms a specification SP', with signature Σ', to a specification SP, with signature Σ, such that $Mod(\kappa(SP')) = \{f_\kappa(A) \mid A \in Mod(SP')\}$. A specification SP is implemented by a specification SP' via a constructor κ if:

$$SP \leadsto \kappa(SP'), \text{ i.e., } Mod(\kappa(SP')) \subseteq Mod(SP).$$

The kind of refinement obtained depends on the choice of κ. For instance the *derive* constructor can be used to hide and/or rename some of the sorts and operations of SP'. In this case, an implementation SP' of SP may have more sorts and operations than SP, or the sorts and the operations may have a different name.

Sannella and Tarlecki [57] extend this definition of refinement with the notion of *abstractor*. This notion is motivated by the *abstract model specification* technique, in which the user defines desired results, any model giving the same results being acceptable. An abstractor α is determined by an equivalence relation $\equiv \subseteq Alg(\Sigma) \times Alg(\Sigma)$ on Σ-algebras. The abstractor transforms a specification SP, with signature Σ, into a specification $\alpha(SP)$, with the same signature. Models of $\alpha(SP)$ are all the models equivalent to at least one model of SP, i.e., $Mod(\alpha(SP)) = \{A \in Alg(\Sigma) \mid \exists A' \in Mod(SP) \text{ s.t. } A \equiv A'\}$. Abstractors and constructors are complementary techniques, which lead to the following definition of refinement. A specification SP is implemented by a specification SP' wrt an abstractor α via a constructor κ if:

$$\alpha(SP) \leadsto \kappa(SP'), \text{ i.e., } Mod(\kappa(SP')) \subseteq Mod(\alpha(SP)).$$

The kind of refinement obtained depends on the choice of the constructor and on the choice of the abstractor. For instance, the *behavioural* abstraction is based on the observational equivalence relation that does not distinguish between algebras that give the same results on terms of external sorts (i.e., sorts of interest for the observation). In this case, a refinement is an implementation of the (abstract) behaviour of SP rather than an implementation of SP itself.

2.3.2 ASTRAL

ASTRAL, due to Ghezzi and Kemmerer [37] is a formal specifications language for realtime systems, that uses types, variables, constants, transitions, and invariants. A real-time system is modelled by a collection of state machines specifications and a single global specification. There may be multiple instances of each state machine, one for each process. Operations of a state machine are specified with transitions defined by an entry assertion, an exit assertion and a duration time. In order to validate ASTRAL specifications, Ghezzi and Kemmerer [38] translate them into TRIO formulae, and apply the validation theory of TRIO.

Coen-Porisini *et al.* [25] define the refinement of ASTRAL specifications. An implementation mapping is used, that maps every type, constant, variable and transitions of a high-level ASTRAL specification to a corresponding term in a lower-level specification. Transitions may be refined either by *selection* or by *sequence*. Selection consists of mapping a high-level transition T to a choice between several lower-level transitions $T_1 \mid \ldots \mid T_n$, such that every time T fires, one and only one T_i $(1 \leq i \leq n)$ fires. Sequence consists of mapping a high-level transition T to a sequence $T_1 \, ; \ldots ; \, T_n$ of lower-level transitions.

Proof obligations use logical formulae for formally proving a refinement step: proofs are built on logical equivalences of entry and exit assertions. More precisely, proof obligations for selection mapping requires first, that at least one T_i fires when and only when T fires (entry assertions of T and entry assertions of T_j $(1 \leq j \leq n)$ logically imply each other); second, that the effect of T_i *logically implies* the effect of T (exit assertion of T_j implies that of T $(1 \leq j \leq n)$); and third, the duration of T_j $(1 \leq j \leq n)$ is equal to that of T.

In the case of sequence mapping, proof obligations are similar: first, sequence $T_1 \, ; \ldots ; \, T_n$ is enabled iff T is enabled (logical equivalence of their entry assertions); second, the effect of $T_1 \, ; \ldots ; \, T_n$ logically implies the effect of T (logical implication); and third, their duration is the same.

2.3.3 B

B, due to Abrial [5, 4], is a method for specifying, refining and coding software systems. The B method is based on the notion of *abstract machine*. An abstract machine can be viewed as a class, an abstract data type, a module or a package. It allows to organise large specifications as independent pieces having well-defined interfaces. An abstract machine models a software system in terms of a state and operations that either modify the state or return a result. The state is specified with: variables (attributes), an invariant, i.e., a logical statement constraining the variables, and an initial value for the variables. There are two kinds of operations: those changing the state without returning a result, and those returning a result (possibly changing the state). The operations modify the state within the limits of the invariant: the new state reached after the modification of the former state by the operation must still validate the invariant. Operations are given by a pre-condition and the way they modify the state. Large abstract machines can be constructed from smaller ones.

The refinement process is part of the method. The refinement M_1 of an abstract machine M is an abstract machine such that: (1) M_1 has the same name as M; (2) M_1 has the same operation names and parameters as M; (3) M_1 has usually a different state (low-level variables y) than M (high-level variables x), thus the invariant clause of M_1, defines an invariant on variables y of M_1, as well as a *change* clause linking the variables of M and those of M_1. In simple cases, the change clause may be given by a function h from the variables of M to the variables of M_1: $y = h(x)$; (4) the pre-condition of the methods in M_1 may change as well as the definition of the methods. M_1 correctly refines M if:

- the initial state of M_1 is compatible with the initial state of M, i.e., $h(v) = w$, where v is the initial state of M and w is the initial state of M_1;
- for every method of M which changes the states, if the invariant and the pre-condition of the method hold in a state e, then the invariant of M_1 and the pre-condition of the corresponding method in M_1 hold for the state $h(e)$, and if the method of M changes state e into state e', then the corresponding method in M_1 must change state $h(e)$ into $h(e')$;
- for every method of M which returns a result, if the invariant of the method and the pre-condition of the method hold in a state e, then the invariant of M_1 and the pre-condition of the corresponding method in M_1 hold for the state $h(e)$, and the result returned by the corresponding method of M_1 must be *equal* to the result returned by the method of M.

It is not necessary that all computations of the methods of M have a low-level counterpart. The refinement of a method has a weaker pre-condition than its high-level counterpart, it can be used in any context where the high-level method can be used, and also in contexts where the high-level method cannot be used. In addition, the low-level method is less non-deterministic than the high-level method. The refinement is correct if the low-level method, used in any context where the high-level method is used, yields the same results, and if the internal states are compatible via the change clause.

An *implementation* is a machine that refines either an abstract machine or a refinement. An implementation cannot be refined further, it has no abstract variables and the operations must be "implementable" (direct translation into a programming language is possible). An implementation may import other abstract machines, whose operations are used to define the operations of the implementation. These machines can be refined further.

2.3.4 Refinement Calculus

The refinement calculus of Back and von Wright [8] views a program as a *predicate transformer*. A predicate $p : \Sigma \to Bool$ is a function from Σ, a set of states, to $Bool = \{T, F\}$, the boolean values. The predicate mentions for each state whether it satisfies or not the predicate. $Pred(\Sigma)$ is the set of all predicates over Σ. Given two sets of states: Σ and Γ, a program is a predicate transformer, $S : Pred(\Sigma) \to Pred(\Gamma)$.

$Pred(\Sigma)$ is a complete lattice (a partial order with least upper bound and greatest lower bound for every subset of $Pred(\Sigma)$). The order relation over $Pred(\Sigma)$ corresponds to the implication ordering: $p \leq q$ if $p \Rightarrow q$, it is defined point-wise, i.e., $p \leq q$ if $p(\sigma) \leq q(\sigma)$ for every $\sigma \in \Sigma$. It defines a refinement ordering on the programs as follows: T refines

S, noted $S \leq T$, if $S(q) \leq T(q)$ for every $q \in Pred(\Sigma)$. The set of all programs from $Pred(\Sigma)$ to $Pred(\Gamma)$ is a complete lattice wrt this order relation.

This notion of refinement models the notion of correctness given by a pre-condition/post-condition pair (or assumption/guarantee): for every pre-condition P and post-condition Q, if S validates post-condition Q, assuming pre-condition P, then T, refining S, validates also post-condition Q, assuming pre-condition P. This definition is extended to data refinement by the means of *encoding* and *decoding* commands, E and F. S is refined by S' through encoding E and decoding F if $S \leq E; S'; F^{-1}$, where the ";" operator is the composition of predicate transformers. Modularity is supported in the following way: if $T(S)$ is a program containing S as a subprogram then $S \leq S' \Rightarrow T(S) \leq T(S')$.

The refinement calculus is extended by Back [7] to parallel and reactive programs and by Back and von Wright [9] to action systems. Among others, the following results are presented: (1) the parallel composition is monotonic wrt refinement, i.e., $A \leq A'$ and $B \leq B'$ implies $A||B \leq A'||B'$; (2) if A' refines A then replacing A by A' in any context using A leads to a refinement, i.e., $A \leq A'$ implies $C[A] \leq C[A']$, where C is the context using A; (3) all temporal properties, validated by $C[A]$, are still validated by $C[A']$.

Utting [58] has extended the refinement calculus to object-oriented programming. This refinement allows modular reasoning about sub-typing, i.e., if c is a sub-type of d, then replacing c by d in a system leads to a refinement.

2.3.5 TLA

The Temporal Logic of Actions (TLA), due to Lamport [46], specifies both closed systems and their properties. Verification tasks are reduced to verification of logical implications: a system satisfies a property if the formula specifying the system implies (logically) the formula specifying the desired property; a system refines another system if the formula specifying the former system implies the formula specifying the latter.

TLA formulae are essentially constructed over *actions*. An action is a relation between an old state and a new state (before and after the action has taken place). The canonical form of a formula specifying a system is made by the conjunction of: (1) an initial predicate, which gives initial conditions on states; (2) a next-state action part, which gives the action (disjunction or conjunction of smaller actions) that must be performed at each step, this part also specifies stuttering steps, i.e., allows that some states may remain unchanged. The next-state action part can be seen as an invariant to be preserved at each step; (3) a fairness part, which allows to express liveness properties. A low-level formula ϕ refines a higher-level one ψ if $\phi \Rightarrow \psi$. There are three points that need to be proved: the initial predicate of ϕ implies the initial predicate of ψ; a step of ϕ simulates a step of ψ (same sequence of states after removing stuttering steps), and ϕ implies the fairness condition of ψ.

In addition, a TLA formula may have visible and internal variables. Internal variables are existentially quantified. In the case of a refinement of a formula with internal variables, the proof that the lower-level system implies the higher-level one can be made easier if we exhibit a *refinement mapping*, which maps the internal variables of the lower-level system to those of the higher-level one.

More generally, for other formalisms, in order to prove that a low-level specification refines a higher-level specification, it is in some cases sufficient to prove the existence of a refinement mapping. A refinement mapping is a function that maps executions (sequences of states) of the low-level specification to executions of the higher-level one (possibly with stuttering). However, the existence of a refinement mapping is sufficient but not necessary to prove a refinement: indeed, it may happen that no refinement mapping from the low-level specification to the higher-level one exists, but the low-level specification is actually a refinement of the higher-level one. The existence of refinement mappings and the way to find a refinement mapping by adding variables to the low-level specification have been discussed by Abadi and Lamport [1].

An extension of TLA to open systems using an assumption/guarantee style is given by Abadi and Lamport [2]. An assumption/guarantee expresses what services are guaranteed by a component, provided its environment (the other components) satisfies some assumptions. A whole system made of several components is specified by the conjunction of the specifications of the components. The conjunction of assumption/guarantees does not trivially imply the conjunction of the assumptions, the conjunction of the guarantees, or another assumption/guarantee, when assumptions are not safety properties.

2.3.6 Refinement as Properties

Jacob [44] advocates that each refinement relation defines a property. He gives the following informal definition of refinement: "a product refines another means that the former product is *no worse with respect to some property of interest* than the latter." This means that the refined model satisfies more specifications than the initial model.

A specification is a *contract* between a customer and an implementor. A specification is defined as the set of all products that would satisfy the customer. A product p satisfies a specification S if $p \in S$. Such a product is called an implementation. A specification S is a reification of a specification T if any implementation of S is also an implementation of T, i.e., $S \subseteq T$. Jacob shows that any property defines a refinement relation on products and vice-versa. A property P is defined as a set of specifications (closed under union and intersection). These specifications stand for all the specifications that satisfy the property.

Given a property, the corresponding refinement relation on products $r \subseteq Products \times Products$ is defined such that: a product p is refined by a product q , noted $(p, q) \in r$, if q appears in any specification where p appears. Conversely, given a refinement relation r on products, the set of specifications forming the property is given by the sets of products S such that: $r(S) = S$; where $r(S) = \{q \in Products \mid (p, q) \in r \wedge p \in S\}$. Indeed, as r is a refinement relation, every product $p \in S$ must be refined by a product in S or in a subset of S, thus $r(S) \subseteq S$, in addition r is reflexive, thus $r(S) = S$. Conversely, $r(S) \nsubseteq S$ means that there are products of S refined by products which are not in S, thus S is too small to be part of the property, S must be enlarged to T with $r(T) = T$.

If several properties are required simultaneously, the refinement relation is obtained by the intersection of the refinement relations of each property. If the properties are contradictory, this intersection may lead to the empty set.

2.4 Discussion

Let us have a look at some informal definitions that apply to the refinements reported above:

> A specification T refines a specification S if all experiments of S are also experiments of T and the results obtained when performing these experiments in T are related to results that can be obtained when performing these experiments in S (FOOPS).

> If D is a refinement of C it must not be possible for a user of the common interface to be able to devise an experiment which would allow him to deduce whether he had an instance of C or of D (VDM^{++}).

> A concrete method, implementing an abstract method, has a weaker pre-condition than the abstract method (it is applicable in at least the same states as the abstract method) and a stronger post-condition (the concrete method returns the same results as the abstract one) (B, Refinement calculus).

A common idea emerges from these definitions: the concrete specification is *different* from the abstract specification, but it must be *compatible* with the abstract specification. The exact meaning of compatible varies from one definition to the other, as well as how far the concrete specification can be from the abstract specification. Several different techniques are used to prove the compatibility of the abstract and the concrete specification, their differences being given. The aim of this section is to discuss the following points. First, the differences allowed between the concrete and the abstract specification are investigated. These differences are constrained by syntactical conditions. Second, we list the semantical conditions that define the compatibility between the concrete and the abstract specifications. Third, we list properties of the definition of a refinement. Then, we discuss the differences between an implementation and a refinement, as well as the use of temporal logic in definitions of refinement, and we report some development guidelines. Finally, we devise a "generic" definition of refinement, based on the preservation of properties. Throughout this section, emphasis is given on model-oriented specifications languages.

2.4.1 Formal Definitions of Refinement: Syntactical Conditions

A concrete specification is a transformation of an abstract specification. It can change syntactical visible elements: names of operations or methods, exported types and sorts (interaction refinement); or hidden elements: states, attributes (data refinement), definition of operations or methods (action refinement).

There are two policies for the visible part: either the abstract and the concrete specifications have a common identical visible part, or they are allowed to have different visible parts. Usually, the abstract and concrete specifications have different hidden parts.

The preservation of signatures (sorts, operations) is a technique that forces the abstract and the concrete specifications to have a common identical visible part. When visible and/or hidden parts are different, the refinement requires that abstract operations are *renamed* to concrete operations, that abstract elements are *refined* to concrete elements, or that abstract states are *retrieved* from concrete ones.

Preservation of Signatures

The preservation of the signature is required when the concrete specification has to allow the *same* observations (experiment, or property) as the abstract specification. The following cases occur: (1) the abstract and the concrete specifications must have the *same* signature, i.e., the concrete specification is not allowed to introduce new visible sorts or operations; (2) the signature of the concrete specification *contains* that of the abstract specification, i.e., the concrete specification may introduce new visible elements, but must keep those of the abstract specification; (3) the concrete specification contains a *part* of the signature of the abstract specification, i.e., both specifications have a common signature part, which will be used for the observations; (4) the concrete specification has no obligations towards the abstract signature, i.e., it is not necessary to preserve any element of the signature.

Algebraic specifications require that the abstract and the concrete specifications have the same signature. CO-OPN requires that the abstract and the concrete specifications have the same events. FOOPS requires that all experiments, and primary sorts (sorts needed for experiments) of the abstract specification are also experiments and sorts of the concrete specification. The B method requires that the high-level machine and the lower-level one have the same name and the same operation names (with the same types).

Use of Retrieve, Refine and Renaming Functions

Some formalisms allow visible or hidden elements of the abstract specification to be different from the visible or the hidden elements of the concrete specification. Thus, essentially for proof purpose, it is necessary to relate abstract and concrete elements, e.g., to translate the former into the latter. Retrieve, refine and renaming functions are used to map abstract and concrete elements. Usually, functions are used. However, in some cases, it is not possible or desirable to use functions. Thus, relations are used instead.

A *retrieve* function is a function from elements of the concrete specification to those of the abstract one. It is usually defined on object-oriented specifications, and it maps concrete attributes to abstract attributes or concrete states to abstract states. A *refine* function is a function from elements of the abstract specification to those of the concrete one. They may be defined either on syntactic and visible elements or on hidden elements, i.e., defined on elements of the signature of the specification, or on the attributes or states of the specification. A *renaming* function is a function from methods of the abstract specification to methods of the concrete specification; it is sometimes part of a refine function.

The definition of refinement implies the following constraints, according to whether these functions are injective, surjective or total functions:

If the refine (or renaming) function is injective this means that: two distinct abstract elements are still refined to two distinct concrete elements. For methods it means that two different methods cannot be refined by the same method. Otherwise, the refine (or renaming) function is non-injective, and a concrete element can refine two distinct abstract elements. If the refine (or renaming) is surjective it means that every concrete element has an abstract counterpart, and no new element can be added. Conversely, if it is non-surjective, new elements (e.g., new methods) can be added. The use of a total refine (renaming) function means that *every* abstract element has exactly one concrete counterpart. It is not possible that an abstract element has no concrete counterpart, and it cannot have more than one.

If the retrieve function is injective, it means that two distinct concrete elements have two distinct abstract counterparts. Otherwise, two or more concrete methods could refine the same abstract method. It is then necessary to stress in the definition of the refinement what it means if two or more concrete methods refine the same abstract method. For instance in timed Petri nets with a TRIO axiomatisation, several concrete transitions can refine the same abstract transition. This means that several firings of the same abstract transition are distributed over the firings of the concrete transitions that refine the abstract transition. If the retrieve function is surjective, then *every* abstract element has a concrete counterpart. Usually this is required for elements taking part into observations, since all possible abstract observations have to be translated into concrete observations. The use of a total retrieve function means that every concrete element has exactly one abstract counterpart. It is not possible for a concrete method to refine two abstract methods, and it is not possible for a concrete element to be a new element not related to an abstract element.

The event function of timed Petri nets with a TRIO axiomatisation is a partial, surjective retrieve function, mapping transitions. The morphisms of the rule-based refinement are a kind of refine function. The reification function of TROLL is a total refine function coupled with a renaming function, mapping object identifiers, attributes and actions.

The change function of B is a retrieve function, mapping attributes. VDM^{++} uses both a retrieve function mapping instance variables, and a total renaming function. A refinement mapping is a retrieve function on states. ASTRAL uses a refine function mapping types, constants, variables and transitions.

2.4.2 Formal Definitions of Refinement: Semantical Conditions

We have seen that syntactically, the concrete specification must be related to the abstract one in some way. Given these syntactic changes, the behaviour of the concrete specification must be "compatible" with the behaviour of the abstract specification.

The semantical conditions of refinement define what "compatible" means. They are defined on the basis of the refine, retrieve, or renaming functions seen before; and they work on the underlying models of both the abstract and the concrete specification. Compatibility often means *preservation of behaviour*. The behaviour of a system is devised through the *observations* that can be made on the system, and the *abstract* view that the user has of the system's state.

There are two kinds of behaviour preservation: the *input/output behaviour preservation*, which is mostly concerned with the result obtained when a method is invoked, and the *whole behaviour preservation*, i.e., the compatibility of traces of the concrete and the abstract systems. The algebraic specifications and the refinement calculus are based solely on input/output behaviour. The other formalisms reported in this section use the behaviour preservation as well.

A supplementary aspect, interesting for object-oriented languages, concerns the use of *object identifiers*, and the obligations of the concrete specification wrt the object identifiers of the abstract specification.

Observations

A system can be seen as a black box that has an interaction with a user (another system or a human being). The user of the system expects some result or behaviour from the system. An observation is a property that the interaction with the system must have. We will use as synonyms the terms observation and observable property.

The notion of observation, or observable property, is present in every definition of refinement: in some cases, the properties are part of the specification and they must be preserved by a refinement; in some other cases, the proof of refinement constructs explicitly the observable properties to be preserved; finally, in other cases, the preservation of observable properties is only implicitly required by the refinement.

For algebraic specifications, the observations are explicitly given by the equations on the operations of the signature. For Petri nets the observations are either properties asserting that the net is safe, live or bound, or properties built on firings of the net. For object-oriented specifications languages, observations are built on method calls. In the case of the B method, pre-conditions, results and invariants are the observations. For refinement calculi, the assumptions and the guarantees are the observations. TLA is based on a next-state action to be preserved, thus observations are built on sequences of states.

Abstract States

An abstract state is the view of the actual system's state observed by the user. In some cases, the user observes only a small part of the actual state: the abstract state is the visible part of the state; the hidden part may be freely modified by a refinement. In other cases, the user does not observe a part of the state, but some input/output parameter whose value depends on the actual value of the state: the abstract state is given by these parameters; the actual state is completely hidden, and a refinement may change it.

The abstractors, used in algebraic specifications, explicitly define abstract states. For the other formalisms reported here, the abstract state is either explicitly given by visible attributes, or implicitly given by the parameters of method calls, or by firable transitions.

Input/Output Behaviour Preservation

The definition of refinement is based on input/output behaviour preservation, when the user of the system is mostly interested by the (isolated) requests it can ask the system. When some (input) conditions hold, a request feasible in the abstract system must be feasible in the concrete one, and the result (output) returned by the concrete system must be compatible or equal to the one returned by the abstract system. The user is not interested by the way the result has been obtained (number of steps used, method called, etc) or by the sequences of requests it can perform.

The input/output behaviour preservation uses the weaker pre-condition/stronger post-condition technique. Indeed, the refinement relation may require that the operations of the concrete specification be used in any situation when the operations of the abstract specification are used. This is known as the "weaker pre-condition". It is coupled with a condition on the result: each time the concrete operation is used, it yields the same (or compatible) result as its abstract counterpart. This is known as the "stronger post-condition". This means that the concrete specification may be used in more situations than the abstract one, but when used in the same situations as the abstract one, it must return the same result, or one of the results that the abstract specification would return. The stronger post-condition is coupled with less non-determinism. Indeed, the concrete operation usually has less non-determinism than the abstract operation, since it is allowed to return one of the results of the abstract operation. It is not necessary that it returns all the possible results of the abstract operation.

Specifications whose model is not a transition system, as well as specifications defined with an assumption/guarantee style, employ this kind of refinement. In the latter case, the assumption is the pre-condition, and the guarantee is the post-condition. Other specifications languages use both the input/output behaviour preservation and the whole behaviour preservation.

Algebraic specifications, B, FOOPS, VDM^{++}, and the refinement calculus use the weaker pre-condition/stronger post-condition.

Whole Behaviour Preservation

The definition of refinement is based on behaviour preservation, when the user is not only interested in the results returned by the system, but also by the sequences of requests it can ask the system, the sequences of states reached by the system, or the choices offered by the system at each point. For instance, the user wants to be able to perform in the concrete system the same choices, or the same sequences of actions as those it can perform in the abstract system.

Systems whose refinement requires behaviour preservation have a semantics based on events and states, e.g. transition systems, event structures or traces.

Simulation notions are used to define behaviour preservation. Simulations are oriented: an abstract behaviour is simulated by a concrete behaviour; or a concrete behaviour is simulated by an abstract behaviour. When both simulations are required, we say that it is a bisimulation. Simulation notions are focused either on events or on states, and the simulation may be weaker or stronger. Among others, we may have the following cases: (1) the concrete and the abstract behaviour must be equal; (2) the concrete and the abstract behaviour must be equal modulo stuttering, i.e., the concrete behaviour may use more steps than the abstract behaviour to reach the same result (or vice-versa); (3) abstract and concrete behaviours are identical on the event part, but states may be different; (4) the concrete and the abstract behaviours must have the same failure set.

The definitions of refinement are usually based on a simulation notion, and requests that *every* abstract behaviour must be simulated by a concrete behaviour. These definitions usually request as well that every concrete behaviour has an abstract counterpart, i.e., no new concrete behaviour that cannot be considered a refinement of an abstract behaviour can be added.

Except the algebraic specifications, B, and the refinement calculus, all the formalisms reported in this chapter use a whole behaviour preservation. Refinements of Petri nets are based on equivalence relations given on the abstract and the concrete transition systems. The refinement is correct, if the abstract and the concrete transition system are equivalent. The CO-OPN formalism uses the bisimulation equivalence, which forces the concrete and abstract trees derived from their respective transition systems to be equal on the event parts. Timed Petri nets using TRIO require the possible abstract firings (sequences or choices of firings) to be also possible (translated) concrete firings. FOOPS requires every abstract experiment to be a concrete experiment, and the concrete results obtained (states) to be related to the abstract results. TROLL allows every possible interleaving of concrete transactions (several actions) to be a refinement of an atomic action. VDM^{++} requires every abstract experiment (sequences or any composition of method calls) to be also a concrete experiment (possibly with renaming) and every concrete experiment using new methods to be obtained as a concrete experiment using only the abstract (possibly renamed) methods. ASTRAL requires identical firings of high-level transitions to correspond to firings of lower-level transitions, i.e., same starting time, same duration, and same result. TLA refinement requires the abstract and the concrete sequences of (visible) states to be equal modulo stuttering, i.e., the abstract trace is allowed to have a sequence of the same visible state.

Management of Object Identifiers

The semantics of object-oriented specifications languages imply that instances of objects are created/destroyed at run-time. Usually, every abstract object identifier has to be related to a concrete object identifier (using a retrieve or a refine function). This is essential if the refinement requires that the same or translated observations be performed in both the abstract and the concrete system, since observations are built with calls of objects' methods.

FOOPS requires every object identifier of the abstract class to be also an object identifier of the concrete class. TROLL uses a refine function that maps abstract object identifiers to concrete object identifiers. VDM^{++} uses a retrieve function from the attributes of the concrete class to those of the abstract class.

2.4.3 Properties of the Refinement Relation

Clearly, in order to perform a stepwise refinement, it is necessary that the definition of refinement is a pre-order relation, otherwise the last step of a sequence of refinements cannot be considered itself as a refinement of the most abstract specification.

In addition, if the system decomposes into smaller parts, it would be interesting to refine every smaller part separately, and then assemble the concrete smaller parts into a concrete specification. If the refinement relation is compositional, the concrete specification, obtained by the composition of concrete smaller parts, is actually a refinement of the abstract specification. However, every refinement relation is not compositional, and the above result is not always guaranteed.

Refinement is a Pre-Order

The refinement relation has to be reflexive, i.e., any specification can be replaced by itself; and transitive, i.e., if P refines to Q and Q refines to R then P refines to R. This is the fundamental requirement that enables the refinement relation to be used for *stepwise* refinement. Transitivity is also called *vertical composition*.

A relation which is reflexive and transitive is a pre-order. A pre-order is an order if it is also anti-symmetric, i.e., if P refines to Q and Q refines to P implies that $P = Q$. This requirement cannot be fulfilled by every specifications language and every refinement relation.

Indeed, if the specifications language allows information hiding, and if the refinement relation is concerned with the visible information only, both P and Q could lead to observable behaviours that are refinement of each other, but they could be different specifications (especially on the hidden parts). If the specifications language does not allow information hiding, but the refinement relation allows different syntaxes related by refine, retrieve and renaming functions, it may happen that two different specifications have identical models or models that are refinement of each other.

However, if the specification does not allow information hiding, and if the refinement relation is concerned with the preservation of all properties (all properties are observable since no information is hidden), and if it does not allow renamings, then the refinement relation is anti-symmetric.

In the specifications languages described in this chapter, the refinement relation is an order for the refinement calculus, but only a pre-order for the others.

Compositional Refinement

A refinement is said to be *compositional*, or to be a *congruence* wrt compositional operators, or compositional operators are said to be *monotonic* wrt refinement, if: the refinement of a composed system is obtained by the refinement of its components. This property of refinement is also called *horizontal composition*. It deals with the proof of refinement: if an abstract component, part of an abstract compound system, is refined by a concrete component, then the replacement of the abstract component by the concrete one, leads to a concrete compound system which is a refinement of the abstract system. The horizontal composition of the refinement relation depends on a *compositional operator*. Compositional operators are not necessarily monotonic wrt a refinement relation, thus the refinement relation is not always compositional. In addition, compositional operators are of different kinds: the use of parameters; the synchronisation with the method of a CO-OPN object; the use of a class (client-ship); or a parallel, sequence or choice operator.

In the formalisms discussed above, some of the refinement relations are compositional: the refinement of parameterised algebraic specifications is a congruence wrt the use of parameters; in the field of structured Petri nets, the CO-OPN refinement of an object is a congruence wrt the use of a Petri net; the union of two nets is monotonic wrt rule-based refinement; FOOPS method combiners (parallel, sequence, choice) are monotonic wrt the refinement of FOOPS methods; the use of a VDM^{++} class is monotonic wrt VDM^{++} refinement; B refinement is a congruence wrt the client-ship, and defines as well several operators that are monotonic wrt B refinement; extensions of the refinement calculus are congruences wrt the parallel operator, and the contexts are monotonic wrt the refinement.

2.4.4 Implementation vs Refinement

For our part, we think that refinement and implementation should be two different things. A *refinement* should be seen as the replacement of a specification by another specification (expressed with the same specifications language). Each refinement step produces a new specification. The replacement has to follow certain rules in order to be correct. The refinement process produces a chain of specifications, with $Spec_1$ begin the most abstract one, and $Spec_n$ the most concrete one; each specification is a correct refinement of the previous one. The refinement process ends when the obtained specification is sufficiently detailed to be immediately translated into a programming language, or has a known implementation (by test or other techniques). An *implementation* is the replacement of the last specification $Spec_n$ of the refinement process by an actual program, expressed in a programming language (different from the specifications language).

In some of the specifications languages discussed in this chapter, implementation is not mentioned at all. In other languages, the words implementation and refinement are used as synonyms, thus there is no distinction between them. VDM^{++} and B make a distinction between refinement and implementation and explain how to reach an actual implementation. VDM^{++} defines implementation classes - which are directly translatable into a procedural language, and which have no abstract type - and gives translation rules to implement specifications by programs. In B, an implementation machine is an abstract machine with no abstract variables and whose operations can be translated into a programming language. An implementation machine cannot be refined further, but if it uses other abstract machines, these machines can be refined further (provided they are not already implementation machines). Both VDM^{++} and B consider the last specification of the refinement process, i.e., specification $Spec_n$, as the implementation; the program is further derived from this implementation.

2.4.5 About the Use of Temporal Logic

Temporal logic is often used for defining and/or proving a refinement. Some of the formalisms reported above use temporal logic for that purpose. TRIO is a temporal logic used to give an axiomatisation to timed Petri nets; observable properties are expressed with the logic, and the refinement is defined as the preservation of these properties. TROLL and VDM^{++} make use of a temporal logic; properties to be preserved by a refinement step are expressed in the logic. In these three cases, the temporal logic is used in addition to the considered specifications language. ASTRAL uses logical implications in order to prove the correctness of a refinement step. In the case of TLA, the specifications language is itself a temporal logic, thus a specification is a property, and the verification of refinement is reduced to the proof of implication.

2.4.6 Development Methodologies

The stepwise refinement process is the part of the development of a software system, where design decisions directed by implementation constraints are taken into account. In our opinion, the refinement process should begin with a very abstract view of the system, describing only the essential functionality of the system. Gradually, complexity is added to this view, so that the more concrete specification, produced by the refinement process, integrates the original functional requirements, as well as some non-functional requirements, and constraints imposed by the chosen programming language.

A development methodology should help the specifier in making design decisions, i.e., it should give *guidelines* for integrating design decisions or implementation constraints in the refinement process. None of the investigated definitions of refinement give guidelines for integrating design decisions into the refinement process.

In the case of a formal specifications language, allowing the *structuring* (inheritance, sub-typing or client-ship relations) of specifications, a development methodology should answer the following questions as well: Is the structure of the specification describing the system, allowed to vary during the refinement process? If yes, how does the structure vary? Is it necessary to refine abstract components into concrete components preserving the same inheritance, sub-typing or client-ship relations? Does the program have to follow the same structure than the last specification of the refinement process?

Except for VDM^{++} and B, the definitions of refinement for the specifications languages reported in this chapter do not discuss the evolution of the structure of the system's specification during the development process.

Lano in [47] discusses two ways of refining the structure of a VDM^{++} specification: *independent structure* and *continuity of structure*. The independent structure does not force the structure of the lower-level specification to be identical to that of the higher-level specification. This kind of development is used when the more concrete level makes use of already developped components, which cannot fit into the new abstract structure. In addition, it allows the structure to grow, since a concrete class, refining an abstract class, may be in a client-ship relation with more classes than the abstract class (annealing). The continuity of structure imposes the following constraints: if an abstract class C is a client of an abstract class S, then a class C_1 refining class C would also be a client of S; if an abstract class C is a sub-type of D, then a class C_1 refining class C would also be a sub-type of D or a sub-type of D_1 a class refining D.

In both cases however, the class that is at the top of the abstract structure hierarchy is refined by a class that is also at the top of the concrete structure hierarchy. The difference is that in the case of independent structure, the classes used in the rest of the concrete hierarchy can be completely different from those of the abstract hierarchy (e.g., they do not have to refine a class of the abstract hierarchy), and the abstract and concrete structure (inheritance, sub-typing, client-ship) can be completely different. In the case of continuity of structure, the abstract and concrete structures must be the same, e.g., a type and its sub-type in the abstract structure are refined by a type and its sub-type in the concrete structure.

In some cases, the definition of refinement is such that it implicitly leaves or not some degrees of freedom for the structure of a lower-level specification wrt the structure of the higher-level one.

A FOOPS specification contains several classes and their relationships, the refinement of a FOOPS specification requires only the experiments of the abstract specification to be also experiments of the concrete specification. It seems that the relationships between the abstract and the concrete classes may be different.

A TROLL system is a collection of objects, the refinement maps abstract objects to concrete objects, as well as their attributes and actions. Thus, the set of objects constituting the abstract system can be totally different (smaller, bigger) from the set of objects constituting the concrete system.

2.4.7 Refinement Preserves Observable Properties

The semantical conditions of refinement define: (1) the observations, i.e., observable properties, that can be made on a system; and (2) the preservation of these observations during a refinement step.

Two cases occur, either the *same* properties, without any change, have to be validated by the concrete specification, or properties of the abstract specification are *translated* into properties of the concrete specification, and those properties have to be validated by the concrete specification. The first case occurs when the syntactical conditions of the refinement impose the same signature on both the abstract and the concrete specifications. The second case occurs when the abstract and the concrete specifications may have different signatures, and refine, retrieve or renaming functions are used. When properties are expressed as formulae, extensions of the refine, retrieve and renaming functions to the formulae are used to actually translate the abstract properties into concrete properties.

Properties are explicitly given by the specification as properties of interest (algebraic specifications, and TLA), or built for proof purpose (TRIO, TROLL, VDM^{++}), or implicitly required by the refinement relation (CO-OPN, FOOPS, B, refinement calculus). We will now explain for each formalism described in this chapter, how the refinement relation preserves properties and what are the kind of properties that are preserved.

Algebraic specifications are given as pairs of signatures and equations. These equations define properties that the models of the specifications must satisfy. The refinement of algebraic specifications implies that the concrete specification preserves the *same* properties of interest as the abstract one. The properties of interest are either the whole set of properties of the abstract specification, or the observable set of properties of the abstract specification, this is the case when abstractors are used. In addition, the concrete specification usually introduces more properties of interest to be preserved by subsequent refinements.

In the case of Petri nets, the refinement is defined on the preservation of properties or on the preservation of equivalences. The refinement of a transition preserves properties asserting that the net is safe, live and bound. The refinement of places via parallel composition preserves failures. The refinement of a timed Petri net using a TRIO axiomatisation preserves all temporal formulae built on firings and that are verified by every execution of the net. These three cases preserve *translated* properties.

The CO-OPN refinement implies that the abstract specification and the concrete specification have the same events, thus the *same* properties have to be preserved. In the case of CO-OPN, properties are all the possible sequences and choices of events' firing, given in the transition system.

In the case of object-oriented specifications, the refinement of FOOPS implies that the experiments that can be performed in the abstract specification are also experiments that can be performed in the concrete specification, and they lead to *related* results. The same experiments can be performed, they do not lead necessarily to the same result (state), but they lead to states that allow same experiments to be performed. The properties are the sequences and choices of experiments, or composition of experiments. The refinement requires that the *same* properties are preserved.

To each TROLL specification is associated a set of temporal logic formulae. These properties represent the set of distributed life cycles of the abstract TROLL system. A refine function is used, that translates every property of the abstract specification into a property of the concrete specification. The refinement implies the preservation of *translated* properties.

To each VDM^{++} class is associated a theory, expressing the semantics of the class in a temporal logic language. The properties are all the possible sequences of method calls, or composition of method calls, and their results. A retrieve function and a renaming function translate every property validated by the theory of the abstract class into a property of the concrete class. The refinement implies that the theory of the concrete class validates the *translated* properties.

An ASTRAL specification is correctly refined if the lower-level transition has the same starting time, the same duration, and provides the same result. Since logical implications on entry and exit assertions are used in order to actually prove a refinement step, the refinement of ASTRAL specification implies that the *translated* properties, i.e., starting time, duration, and result, expressed with entry and exit assertions are preserved.

A B class defines invariants, and methods that either change attributes or return a result (possibly changing the attributes). Methods cannot be renamed, and those returning a result are refined to methods producing the same results. The properties are all possible calls of methods and their results (when there is any). A method call is possible if the pre-condition holds, the new values for the attributes validate the invariant. The low-level specification validates the *same* set of properties as the high-level specification; the same calls are possible, and when there is a result, the same result is returned.

The refinement calculus implies that for every pre-condition P and post-condition Q, if program S validates post-condition Q, assuming pre-condition P, then program T, refining S, validates also post-condition Q, assuming pre-condition P. The properties are all these pairs of pre-condition and post-condition for S, and the refinement preserves the *same* pairs. Back [7] extends the refinement calculus to reactive programs, and shows that the simulation refinement of reactive program preserves any temporal logic property insensitive to stuttering.

The specification of a system in TLA *is* a temporal logic formula, i.e., it is a property. This property is made of some invariant (the next-state part) and some liveness property (the fairness part). A concrete system refines an abstract system if the former implies the latter. Thus, the refinement implies the preservation of the same properties.

2.4.8 Conclusion

We have shown that the refinements described in this chapter are all based on the preservation of (possibly translated) properties (either implicitly, or explicitly by the means of additional logical formulae). This joins the ideas of Jacob [44], who shows that every refinement defines a set of properties and vice-versa.

The definitions of refinement discussed in this chapter can all be described by the informal following definition:

> *A specification $Spec'$ refines a specification $Spec$ if* ***the properties of interest*** *of $Spec$ are preserved by $Spec'$.*

The preservation of these properties with or without syntactical changes forces a concrete specification to satisfy some syntactical requirements. If the *same* properties must be preserved, then the concrete specification and the abstract specification have a part of the signature in common. Otherwise, *translated* properties must be preserved and retrieve, refine or rename functions are used to relate the abstract and the concrete specification.

The kind of properties to preserve will affect the semantical requirement of the definition of refinement. If the property deals with the returned results, the refinement requires an input/output behaviour preservation; if the property deals with a sequence of experiments, the refinement requires a whole behaviour preservation.

In addition, the refinement must be a pre-order, in order to perform sequences of refinements leading to a very concrete specification, which is actually a refinement of the most abstract specification. However, it is not necessary for the refinement to be an order.

If the refinement can be performed on smaller parts of a system, and the composition of the concrete smaller parts builds a concrete specification, which is actually a refinement of the abstract specification, then the refinement is compositional.

Finally, an implementation is the last step before the program is obtained, or it is the program itself. Therefore, it should be distinguished from a refinement.

Chapter 3

A Theory of Refinement and Implementation

At the end of Chapter 2, we drew the conclusion that a low-level specification always preserves some properties of interest of a higher-level specification. Thus, any definition of refinement can be captured by the following informal definition:

> *A specification $Spec'$ refines a specification $Spec$ if the* ***properties of interest*** *of $Spec$ are preserved by $Spec'$.*

Our goal is to define a general theory of refinement of model-oriented specifications, that relies *explicitly* on properties of interest. Therefore, the set of properties of interest is joined to every specification; it is a subset of the set of all properties that the specification guarantees. This subset is called a *contract*. Formulae of the contract are expressed using a logical language. Pairs of model-oriented specifications and contracts are called *contractual specifications*. A lower-level contractual specification is thus a correct refinement of a higher-level contractual specification, if it preserves the contract of the higher-level contractual specification. This approach to refinement lies then within the two languages framework described by Pnueli [54]; and integrates built-in features, for correctness as advocated by Meyer [50], since correctness is based on the contracts.

A series of refinement steps is followed by an implementation phase. The implementation is defined in a way similar to the refinement: a contractual program, i.e., a pair made of a program and a contract, implements correctly a contractual specification if it preserves the contract of the contractual specification.

First this chapter defines contractual specifications and their refinement. Second, it defines contractual programs and the implementation of contractual specifications by contractual programs. Third, the conditions that enable to perform a stepwise refinement followed by an implementation are discussed. Fourth, the compositional refinement and the compositional implementation of contractual specifications are defined. Finally, this chapter ends with a discussion aiming at a better understanding of the use of contracts in a development process.

3.1 Refinement Based on Contracts

As we intend to make explicit the use of properties in order to constrain the refinement, we require every specification to be linked with a set of properties. This set of properties is called a *contract*. The pair formed by a specification and a contract is called a *contractual specification*. Since we are interested more particularly by formal specifications languages that are model-oriented, we advocate the use of a logic, in order to express properties on specifications. Indeed, model-oriented specifications languages are well suited to model a system, but they are not well suited to express properties of a system. Therefore, the contract is actually a set of formulae expressed on the specification, that is satisfied by all models of the specification.

The basic idea of refinement consists in replacing a high-level contractual specification by a lower-level contractual specification whose models preserve the contract guaranteed by the higher-level specification.

In order to remain on a general level, we will not constrain syntactically the lower-level contractual specifications wrt the higher-level ones, i.e., syntactical changes are allowed. A *refine relation* associates one or more elements of the low-level contractual specification to elements of the high-level contractual specification. The refine relation explains the syntactical evolution of the high-level specification towards the low-level specification.

The use of a refine relation, allowing syntactical changes, implies the translation of the high-level contract into a set of formulae expressed on the lower-level specification. The translation is performed by the means of a *formula refinement*, i.e., a function, univocally defined on the basis of the refine relation, which maps every high-level property of the contract into a low-level formula. The formula refinement explains the semantical evolution of the high-level specification to the low-level specification, e.g., when a high-level element is related to several lower-level elements, the formula refinement has to explain how the lower-level elements replace the single higher-level element in a formula.

The refinement is then defined as the replacement of a high-level contractual specification by a lower-level contractual specification whose contract *contains* the *translated* contract of the higher-level contractual specification. In this way, every model of the lower-level specification satisfies the translated contract of the higher-level specification, since it satisfies the contract of the lower-level specification.

First this section defines contractual specifications, then presents the refine relation, and the formula refinement, and finally gives the definition of the refinement of contractual specifications.

3.1.1 Contractual Specifications

Contractual specifications are pairs of specifications and contracts. A contract is a set of formulae satisfied by all the models of a specification. In a contractual specification, the specification part stands for the complete description of the system, functionality and behaviour. The contract stands for the essential requirements of the specification that must be satisfied by a refinement step or an implementation step. The contract is not a means to make a selection between models of a specification in order to retain only those models satisfying the contract; it is a means to make a selection between all the specifications in order to retain those that correctly refine the high-level specification. Therefore, the contract does *not* correspond to an *extra* set of requirements, it is a subset of all the properties satisfied by all the models of the specification.

We assume that we have a given formalism that formally defines the syntax and semantics of specifications.

Notation 3.1.1 *Specifications, Models.*
We denote by SPEC *the set of all specifications that can be expressed in the formalism, by* MOD *the universe of all models, by* $Mod \in \text{MOD}$ *a model, and by* $\text{MOD}_{Spec} \subseteq \mathcal{P}(\text{MOD})$ *the set of all models of a specification* $Spec \in \text{SPEC}$.

We are mostly interested in systems having models based on events and states. These systems usually have only one model, i.e., a transition system, an event structure or a set of traces. However, in order to as general as possible, we consider MOD_{Spec} as a set, even if in most cases, this set reduces to a singleton.

We assume as well that we have a given logic which enables to express formulae on the specifications of the given formalism; and a satisfaction relation between the models of a specification and the formulae.

Notation 3.1.2 *Formulae, Satisfaction Relation, Properties.*
We denote by PROP *the set of all formulae that can be written in the given logic and that are expressed on specifications of the given formalism, and by* $\text{PROP}_{Spec} \subseteq \text{PROP}$ *the set of all formulae that can be expressed on* $Spec \in \text{SPEC}$.

We denote $\models$ *the satisfaction relation:* $\models \subseteq \text{MOD} \times \text{PROP}$. *It is such that* $(Mod, \phi) \in \models$ *iff* Mod *is a model that satisfies* ϕ. *We note* $Mod \models \phi$ *when* $(Mod, \phi) \in \models$.

Given the satisfaction relation $\models$, *we extend the notation to sets of formulae and sets of models of specifications. We write* $\text{MOD}_{Spec} \models \phi$, *if* $Mod \models \phi$ *for every* $Mod \in \text{MOD}_{Spec}$; $Mod \models \Phi$, *if* $Mod \models \phi$ *for every* $\phi \in \Phi$; *and* $\text{MOD}_{Spec} \models \Phi$, *if* $\text{MOD}_{Spec} \models \phi$ *for every* $\phi \in \Phi$. *The models of* $Spec$ *satisfy the empty set of formulae:* $\text{MOD}_{Spec} \models \emptyset$, *for every* $Spec \in \text{SPEC}$.

We denote by Φ_{Spec} *the set of all formulae satisfied by all the models of* $Spec$: $\Phi_{Spec} = \{\phi \in \text{PROP}_{Spec} \mid \text{MOD}_{Spec} \models \phi\}$.

A formula ϕ, *satisfied by all models of* $Spec$, *i.e.,* $\phi \in \Phi_{Spec}$, *is called a property of* $Spec$. *The set* Φ_{Spec} *is called the set of properties of* $Spec$.

A contract on a specification *Spec* is a set of properties of *Spec*, i.e., a set of formulae satisfied by all the models of *Spec*.

Definition 3.1.3 *Contract.*
Let Spec be a specification. A contract on Spec, denoted Φ*, is a set of properties of Spec:*

$$\Phi \subseteq \Phi_{Spec}.$$

As we said before, the contract does not make a selection between models of a specification. The contract is defined in such a way that it is satisfied by all models; it is only a subset of the set of all properties satisfied by the models of the specification, i.e., it may even be a strict subset $\Phi \subset \Phi_{Spec}$. When $\Phi = \Phi_{Spec}$, we say that the contract is *total*, when $\Phi \subset \Phi_{Spec}$, we say that the contract is *partial*.

A contractual specification is a pair formed by a specification and a contract on the specification.

Definition 3.1.4 *Contractual Specifications.*
Let Spec be a specification, and $\Phi \subseteq \Phi_{Spec}$ *be a contract on Spec. A contractual specification is a pair:*

$$CSpec = \langle Spec, \Phi \rangle.$$

Notation 3.1.5 CSPEC *denotes the set of all contractual specifications.*

The models of $\langle Spec, \Phi \rangle$ are simply given by the models of *Spec*.

Definition 3.1.6 *Models of a Contractual Specification.*
Let $CSpec = \langle Spec, \Phi \rangle$ *be a contractual specification, and* MOD_{Spec} *be the models of Spec. The set of models of CSpec, denoted* MOD_{CSpec}*, is given by:*

$$\text{MOD}_{CSpec} = \text{MOD}_{Spec}.$$

3.1.2 Refine Relation

We allow syntactical changes between a high-level and a low-level specification. As we have seen in Chapter 2, syntactical changes imply either the use of refine, and renaming functions, in order to be able to map elements of the higher-level specification to elements of the lower-level one; or the use of a retrieve function, in order to map elements of the lower-level specification to elements of the higher-level one. By elements, we mean any syntactical term of a specification. Elements can appear in formulae.

If we use a refine function, we will not be able to allow a single high-level element to be refined by two or more low-level elements. Conversely, if we use a retrieve function, we will not be able to allow two distinct high-level elements to be refined by the same low-level element. In order to encompass functional requirements, we will use a *relation* instead of a function. We will call this relation, the *refine relation*.

Since elements may appear in formulae, the only restriction that the refine relation must satisfy is that every abstract element of the specification that *takes part* in properties of the contract must have at least one concrete counterpart. Indeed, we want to be able to translate every property of the high-level contract into a formula of the lower-level specification.

Notation 3.1.7 *Elements of a Specification.*
We denote by $\textsc{Elem}_{CSpec}$ *the elements of a contractual specification* $CSpec$*.*

Definition 3.1.8 *Refine Relation.*
Let $CSpec$*,* $CSpec'$ *be two contractual specifications. A refine relation on* $CSpec$ *and* $CSpec'$*, denoted* λ*, is a relation on elements of* $CSpec$ *and elements of* $CSpec'$*:*

$$\lambda \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec'} ,$$

such that for every $e \in \textsc{Elem}_{CSpec}$ *that takes part in properties of the contract of* $CSpec$*, there is* $e' \in \textsc{Elem}_{CSpec'}$ *and* $(e, e') \in \lambda$*.*

Remark 3.1.9 *The identity refine relation, denoted* $Id_{\textsc{Elem}_{CSpec}} \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec}$*, is such that:* $(e, e') \in Id_{\textsc{Elem}_{CSpec}}$ *iff* $e = e'$*.*

During a refinement process, a high-level contractual specification is refined by a lower-lever contractual specification, which in turn is refined by a lower-level specification, etc. We want to be able to follow the syntactical changes applied to the elements of the high-level contractual specification during the whole refinement process. The following composition of refine relation is a means to follow these changes.

Definition 3.1.10 *Composition of Refine Relations.*
Let $CSpec$*,* $CSpec'$*, and* $CSpec''$ *be three contractual specifications,* $\lambda \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec'}$ *be a refine relation on* $CSpec$ *and* $CSpec'$*, and* $\lambda' \subseteq \textsc{Elem}_{CSpec'} \times \textsc{Elem}_{CSpec''}$ *be a refine relation on* $CSpec'$ *and* $CSpec''$*. The composition of* λ *and* λ'*, noted* $\lambda; \lambda'$ *is a relation on* $CSpec$ *and* $CSpec''$*:*

$$\lambda; \lambda' \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec''} ,$$

such that $(e, e'') \in \lambda; \lambda'$ *iff there exists* $e' \in \textsc{Elem}_{CSpec'}$ *with* $(e, e') \in \lambda$ *and* $(e', e'') \in \lambda'$*.*

Remark 3.1.11 *Composition $\lambda;\lambda'$ is a relation on elements of CSpec and elements of CSpec'', but it may happen that it is not a refine relation, i.e., it is not total*[1] *on elements of the contract of CSpec.*

3.1.3 Formula Refinement

As we said before, we want to define a refinement that preserves the contract. The use of a refine relation implies the translation of the formulae.

Given a refine relation, a *formula refinement* is univocally[2] defined. The formula refinement is a function that maps a formula, expressible on the high-level specification, into a formula expressible on the low-level specification. The formula refinement may be partial, but must be total on properties of the high-level contract. Indeed, if a property of the high-level contract has no corresponding low-level formula, this means that during the refinement we lost this property, and that it will be guaranteed neither by the lower-level specification nor by further refinement steps. The formula refinement is not necessarily injective, since two or more abstract elements can be related to the same concrete element, and thus different abstract formulae are translated into the same concrete formula. Similarly, the formula refinement is not necessarily surjective, since the refine relation does not necessarily relate every concrete element with an abstract one, thus there are concrete formulae that cannot be considered as refinement of an abstract formula.

When the refine relation can be seen as a function, i.e., every abstract element has at most one counterpart, the formula refinement is a trivial extension of the refine relation to the formulae. When the refine relation associates several concrete elements to a single abstract element, the formula refinement must clearly describe how the abstract formula, containing the abstract element, is refined into a concrete formula. We will not impose any formula refinement here, since it depends both on the specifications language and the logic used for specifying the contracts. We will only impose several conditions on the formula refinement in order to ensure that the refinement relation, defined in the sequel, is a pre-order.

Definition 3.1.12 *Formula Refinement.*
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$ be two contractual specifications, $\lambda \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec'}$ a refine relation on CSpec and CSpec'. A formula refinement, denoted Λ, is a function, univocally defined from λ, which maps formulae expressed on Spec into formulae expressed on Spec':

$$\Lambda : \textsc{Prop}_{Spec} \rightarrow \textsc{Prop}_{Spec'} \ ,$$

such that:

[1] a relation $r \subseteq A \times B$ is said to be total on A if every element of A is related by r to some element of B.

[2] we assume that from any refine relation it is possible to obtain, in an unambiguous way, a formula refinement.

- *Λ maps every property of the contract of $CSpec$ to formulae of $Spec'$, i.e., $\Lambda(\phi)$ is defined for every $\phi \in \Phi$;*

- *the formula refinement Λ derived from $\lambda = Id_{\mathrm{ELEM}_{CSpec}}$ must be the identity on PROP_{Spec}, i.e. $\Lambda(\phi) = \phi$, for every $\phi \in \mathrm{PROP}_{Spec}$. It is noted $Id_{\mathrm{PROP}_{Spec}}$;*

- *given two refine relations λ and λ' such that their composition is defined $\lambda'' = \lambda; \lambda'$ and is a refine relation, the formula refinement Λ'' derived from λ'' is such that $\Lambda'' = \Lambda' \circ \Lambda$; where Λ', Λ are the formula refinements derived from λ' and λ respectively, and $\circ$ is the composition operator on functions.*

Notation 3.1.13 *Refinement of a Set of Formulae.*
Given $\Lambda : \mathrm{PROP}_{Spec} \rightarrow \mathrm{PROP}_{Spec'}$ a formula refinement, we denote by $\Lambda(\Phi)$ the image of Φ under Λ. $\Lambda(\Phi) = \{\phi' \in \mathrm{PROP}_{Spec'} \mid \exists \phi \in \Phi \text{ s.t. } \Lambda(\phi) = \phi'\}$.

3.1.4 Refinement Relation

A low-level contractual specification is a correct refinement of a higher-level contractual specification if the former preserves the contract of the latter. As syntactical changes are allowed, this means that the contract of the lower-level contractual specification *contains* the *translated* contract of the higher-level contractual specification. The translation of the contract is obtained by the means of the formula refinement that is univocally defined from the refine relation.

Definition 3.1.14 *Refinement of Contractual Specifications via λ.*
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$ be two contractual specifications, $\lambda \subseteq \mathrm{ELEM}_{CSpec} \times \mathrm{ELEM}_{CSpec'}$ be a refine relation on $CSpec$ and $CSpec'$, and Λ be the formula refinement univocally defined from λ. $\langle Spec', \Phi' \rangle$ is a refinement of $\langle Spec, \Phi \rangle$ via λ, noted $\langle Spec, \Phi \rangle \sqsubseteq^{\lambda} \langle Spec', \Phi' \rangle$, iff

$$\Lambda(\Phi) \subseteq \Phi'.$$

If $\langle Spec', \Phi' \rangle$ refines $\langle Spec, \Phi \rangle$ then every model of $\langle Spec', \Phi' \rangle$ satisfies at least $\Lambda(\Phi)$. Indeed, every model of $\langle Spec', \Phi' \rangle$ satisfies the contract Φ', thus every model satisfies $\Lambda(\Phi)$. A lower-level specification has no obligation towards the properties of the higher-level specification that are not in the contract, i.e., towards $\Phi_{Spec} - \Phi$.

Definition 3.1.15 *Refinement Relation.*
The refinement relation, noted $\sqsubseteq$, is a relation on contractual specifications:

$$\sqsubseteq \subseteq \mathrm{CSPEC} \times \mathrm{CSPEC}\ ,$$

such that for every $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle \in$ CSPEC, $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec', \Phi' \rangle$ *iff*

$\exists \lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ *a refine relation on* $CSpec$ *and* $CSpec'$, *s.t.* $\langle Spec, \Phi \rangle \sqsubseteq^{\lambda} \langle Spec', \Phi' \rangle$.

Remark 3.1.16 *The definitions of refinement given for* TROLL, *timed Petri nets using a TRIO axiomatisation, and VDM*$^{++}$, *are very close to the definition of refinement using contracts. Indeed, each of them uses a temporal logic to express formulae on the specifications. A lower-level specification is a correct refinement of a higher-level specification if the translated properties of a whole given class are logically implied by lower-level properties.*

Remark 3.1.17 *Definition 3.1.14 requires an inclusion of the translated high-level contract into the lower-level contract. The reason for requiring an inclusion, instead of a logical implication, lies in the fact that a set of formulae* Φ *on* $Spec$ *is actually a contract iff every model of* $Spec$ *satisfies* Φ. *Therefore, logical implication* $\Phi \Rightarrow \Phi_{Spec}$ *holds, since every model satisfying* Φ *is also a model satisfying* Φ_{Spec}. *If we require* $\Phi' \Rightarrow \Phi$ *(assuming that* $\Lambda = Id_{\text{PROP}_{Spec}}$*), then we have* $\Phi' \Rightarrow \Phi_{Spec}$. *This is clearly what we want to avoid.*

The use of inclusion takes as well its motivation from the application of the general theory of refinement to the CO-OPN/2 language and the HML logic, presented in the following chapters. For such a simple logic, inclusion naturally provides the requirements needed for establishing the definition of refinement.

However, in order to fully assess the choice of inclusion of the contracts wrt that of implication, it is necessary to further apply the general theory, presented in this chapter, to another model-oriented specifications language, and to another logic.

3.1.5 Properties of the Refinement Relation

A refinement relation is useful for stepwise refinement if it is reflexive and transitive. We will now state and show this result for the refinement relation defined above.

Proposition 3.1.1 *Refinement Relation is a Pre-Order.*
The refinement relation $\sqsubseteq \subseteq$ CSPEC $\times$ CSPEC *is a pre-order.*

Proof.
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$ and $CSpec'' = \langle Spec'', \Phi'' \rangle$ be three contractual specifications. Relation $\sqsubseteq$ is a pre-order if it is: (1) reflexive, i.e, $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec, \Phi \rangle$ for every $\langle Spec, \Phi \rangle \in$ CSPEC; and (2) transitive, i.e, $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec', \Phi' \rangle$ and $\langle Spec', \Phi' \rangle \sqsubseteq \langle Spec'', \Phi'' \rangle$ implies $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec'', \Phi'' \rangle$, for every $\langle Spec, \Phi \rangle$, $\langle Spec', \Phi' \rangle$, $\langle Spec'', \Phi'' \rangle \in$ CSPEC.

- Reflexivity.
For every contractual specification $CSpec = \langle Spec, \Phi \rangle$, we consider $\lambda = Id_{\text{ELEM}_{CSpec}}$ as the refine relation. The formula refinement obtained is given by $\Lambda = Id_{\text{PROP}_{Spec}}$, and $\Lambda(\Phi) = \Phi$. It follows trivially that $\Lambda(\Phi) \subseteq \Phi$, thus $\langle Spec, \Phi \rangle \sqsubseteq^{Id_{\text{ELEM}_{CSpec}}} \langle Spec, \Phi \rangle$. This implies $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec, \Phi \rangle$.

- Transitivity.
$\langle Spec', \Phi' \rangle \sqsubseteq \langle Spec'', \Phi'' \rangle$ implies that there exists $\lambda' \subseteq \text{ELEM}_{CSpec'} \times \text{ELEM}_{CSpec''}$ a refine relation such that $\langle Spec', \Phi' \rangle \sqsubseteq^{\lambda'} \langle Spec'', \Phi'' \rangle$. Λ', the formula refinement univocally defined from λ', is such that $\Lambda'(\Phi') \subseteq \Phi''$.

  $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec', \Phi' \rangle$ implies that there exists $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ a refine relation such that $\langle Spec, \Phi \rangle \sqsubseteq^{\lambda} \langle Spec', \Phi' \rangle$. Λ, the formula refinement univocally defined from λ, is such that $\Lambda(\Phi) \subseteq \Phi'$.

  λ and λ' can be composed in order to form $\lambda'' = \lambda; \lambda' \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec''}$. λ'' is actually a refine relation, i.e., it is total on the contract Φ. Indeed, first, λ is total on elements of contract Φ, and $CSpec'$ refines $CSpec$ via λ, thus all elements of contract Φ are related to elements of contract Φ'. Second, λ' is total on elements of contract Φ', thus all elements of contract Φ are related to elements of contract Φ'' by $\lambda; \lambda'$. Consequently, $\lambda'' = \lambda; \lambda'$ is a refine relation. By definition, if λ'' is a refine relation, then Λ'', the formula refinement, univocally defined from λ'', is such that: $\Lambda'' = \Lambda' \circ \Lambda$.

  Therefore, we have: $\Lambda'(\Lambda(\Phi)) \subseteq \Lambda'(\Phi')$. Since $\Lambda'(\Phi') \subseteq \Phi''$, we derive that $\Lambda'(\Lambda(\Phi)) \subseteq \Phi''$. Thus, $\Lambda'(\Lambda(\Phi)) \subseteq \Phi''$ implies $\Lambda''(\Phi) \subseteq \Phi''$, which in turn implies $\langle Spec, \Phi \rangle \sqsubseteq^{\lambda;\lambda'} \langle Spec'', \Phi'' \rangle$, which finally implies $\langle Spec, \Phi \rangle \sqsubseteq \langle Spec'', \Phi'' \rangle$.

■

3.2 Implementation Based on Contracts

A refinement step consists of replacing a high-level specification by a lower-level specification, both specifications being expressed within the *same* language. The implementation step replaces a specification by a program, expressed in a programming language, which is usually *different* from the specifications language. The implementation links the world of specifications to the world of programs. Thus, the implementation shares a lot of similarities with the refinement, even though, due to this change of world, it slightly differs from the refinement.

The basic idea of implementation consists of replacing a contractual specification by a *contractual program* whose models preserve the contract of the contractual specification. A contractual program is defined like a contractual specification, it is a pair made of a program and a contract, i.e., a set of properties that the program guarantees.

We do not constrain syntactically a low-level specification wrt a high-level specification. Due to the change of language, the gap between the program and the specification is bigger than that between two specifications. Thus, we will neither constrain syntactically the program wrt the contractual specification. An *implement relation* associates elements of the contractual specification to elements of the contractual program. Formulae of the specifications are translated to formulae expressed on the programs, by the means of a function called *formula implementation*.

The implementation is then defined as the replacement of a contractual specification by a contractual program whose contract *contains* the *translated* contract of the contractual specification.

This section presents contractual programs, the implement relation, the formula implementation, and finally the implementation of a contractual specification by a contractual program.

3.2.1 Contractual Programs

A given program $Prog$, written in a given source code of a given programming language, has as many models as the number of target machines. Indeed, the same source code may be compiled by different compilers (one for each target machine), and thus we obtain different machine codes. Once we have a machine code, we can associate it to a transition system, i.e., the set of all possible executions of the machine code. This transition system is considered as *a* model of the original source code $Prog$. Thus, one source code may have several models (one for each target machine). In the case of virtual machines, we consider the model in the virtual machine, instead of every model in every actual machine. The correspondence between the virtual and the actual machine is ensured by the interpreter, which respects the semantics of the virtual machine.

In the rest of this chapter, we associate a set of models to a program source. This set of models contains only the models associated to machines on which the program will actually be executed. Then, a contractual program is a pair made of a program and a set of formulae that every model of this set satisfies.

We assume that we have a given programming language, which formally defines the syntax of programs; to every program is attached a set of models, one for each envisaged target machine.

Notation 3.2.1 *Programs, Models.*
We denote by $\textsc{Prog}$ *the set of all programs (source code) that can be written with the given programming language, by* $\textsc{Mod}_{\textsc{Prog}}$ *the set of all their models, by* $Mod \in \textsc{Mod}_{\textsc{Prog}}$ *a model, and by* $\textsc{Mod}_{Prog} \subseteq \mathcal{P}(\textsc{Mod}_{\textsc{Prog}})$ *the set of the considered models of a program* $Prog \in \textsc{Prog}$.

We also assume that we have a given logic that makes it possible to express formulae on the programs of the given programming language; and a satisfaction relation between the models of the programs and the formulae. This logic can be different from that used for the specifications, since the formal specifications language is different from the programming language.

Notation 3.2.2 *Formulae, Satisfaction Relation, Properties.*
We denote $\textsc{Prop}$ *the set of all formulae that can be written in the given logic and that are expressed on the programs of the given programming language, and* $\textsc{Prop}_{Prog} \subseteq \textsc{Prop}$ *the set of all formulae that can be expressed on* $Prog \in \textsc{Prog}$*. It will be clear from the context if a formula is expressed on a program or on a specification.*

We denote $\models$ *the satisfaction relation:* $\models \subseteq \textsc{Mod}_{\textsc{Prog}} \times \textsc{Prop}$*. It is such that* $(Mod, \psi) \in \models$ *iff* Mod *is a model that satisfies* ψ*. We denote* $Mod \models \psi$ *when* $(Mod, \psi) \in \models$*.*

Given the satisfaction relation $\models$*, we extend the notation to sets of formulae and sets of models of programs. We write* $\textsc{Mod}_{Prog} \models \psi$*, if* $Mod \models \psi$ *for every* $Mod \in \textsc{Mod}_{Prog}$*;* $Mod \models \Psi$*, if* $Mod \models \psi$ *for every* $\psi \in \Psi$*; and* $\textsc{Mod}_{Prog} \models \Psi$*, if* $\textsc{Mod}_{Prog} \models \psi$ *for every* $\psi \in \Psi$*. The models of* $Prog$ *satisfy the empty set of formulae:* $\textsc{Mod}_{Prog} \models \varnothing$*, for every* $Prog \in \textsc{Prog}$*.*

We denote Ψ_{Prog} *the set of all formulae satisfied by all the models of* $Prog$*:* $\Psi_{Prog} = \{\psi \in \textsc{Prop}_{Prog} \mid \textsc{Mod}_{Prog} \models \psi\}$*.*

A formula ψ*, satisfied by all models of* $Prog$*, i.e.,* $\psi \in \Psi_{Prog}$*, is called a property of* $Prog$*. The set* Ψ_{Prog} *is called the set of properties of* $Prog$*.*

As for contractual specifications, a contractual program is a pair made of a program and a contract, i.e., a set of properties of $Prog$.

Definition 3.2.3 *Contract.*
Let $Prog$ *be a program. A contract on* $Prog$*, denoted* Ψ*, is a set of properties of* $Prog$*:*

$$\Psi \subseteq \Psi_{Prog}.$$

Definition 3.2.4 *Contractual Programs.*
Let $Prog$ *be a program, and* $\Psi \subseteq \Psi_{Prog}$ *be a contract on* $Prog$*. A contractual program is a pair:*

$$CProg = \langle Prog, \Psi \rangle.$$

Notation 3.2.5 $\textsc{CProg}$ *denotes the set of all contractual programs.*

The models of $\langle Prog, \Psi \rangle$ are simply given by the models of $Prog$.

Definition 3.2.6 *Models of a Contractual Program.*
Let $CProg = \langle Prog, \Psi \rangle$ *be a contractual program, and* MOD_{Prog} *be the models of* $Prog$. *The set of models of* $CProg$, *denoted* MOD_{CProg}, *is given by:*

$$\text{MOD}_{CProg} = \text{MOD}_{Prog}.$$

As for contractual specifications, the contract of a program does not limit the set of models, since it is a set of formulae "naturally" satisfied by all models of the program.

3.2.2 Implement Relation

The refine relation relates elements of a high-level contractual specification to elements of a lower-level contractual specification, because syntactical changes are allowed during a refinement step. In the case of the implementation step, syntactical changes are necessary between a specification and a program, since the formal specifications language is usually not a programming language. While a refine relation is a relation on elements of contractual specifications, an *implement relation* is a relation on elements of a contractual specification and elements of a contractual program. By elements of a contractual program, we mean any syntactical term related to the program, for example, a Class name or a method name (in the case of object-oriented programming languages).

Notation 3.2.7 *Elements of a Program.*
We denote by ELEM_{CProg} *the elements of a program* $Prog$.

Definition 3.2.8 *Implement Relation.*
Let $CSpec$ *be a contractual specification, and* $CProg$ *be a contractual program. An implement relation on* $CSpec$ *and* $CProg$, *denoted* λ^I, *is a relation on elements of* $CSpec$ *and elements of* $CProg$:

$$\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg},$$

such that for every $e \in \text{ELEM}_{CSpec}$ *that takes part in the properties of the contract of* $CSpec$, *there is* $e' \in \text{ELEM}_{CProg}$ *and* $(e, e') \in \lambda^I$.

During a refinement process, we follow the syntactical changes of the elements of a contractual specification by composing refine relations. An implementation step occurs at the end of a series of refinement steps. The implementation of the most concrete specification should be as well an implementation of the most concrete. In order to examine the syntactical changes that occur during a refinement step followed by an implementation step, we define the composition of refine relations and implement relations.

Definition 3.2.9 *Composition of Refine Relations and Implement Relations.*
Let $CSpec$, $CSpec'$, be two contractual specifications, and $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ be a refine relation on $CSpec$ and $CSpec'$. Let $CProg$ be a contractual program, and $\lambda^I \subseteq \text{ELEM}_{CSpec'} \times \text{ELEM}_{CProg}$ an implement relation on $CSpec'$ and $CProg$. The composition of λ and λ^I, noted $\lambda; \lambda^I$ is a relation on elements of $CSpec$ and elements of $CProg$:

$$\lambda; \lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg} \ ,$$

such that $(e, e'') \in \lambda; \lambda^I$ iff there exists $e' \in \text{ELEM}_{CSpec'}$ with $(e, e') \in \lambda$ and $(e', e'') \in \lambda^I$.

Remark 3.2.10 *The composition of refine relations is not always a refine relation. Similarly, the composition of a refine relation and an implement relation is a relation which is not necessarily an implement relation.*

3.2.3 Formula Implementation

In the case of refinement, the use of a refine relation on elements of a high-level contractual specification and elements of a low-level contractual specification, implies the use of a formula refinement, mapping high-level formulae to low-level formulae. It is identical in the case of the implementation. The use of an implement relation, on a contractual specification and a contractual program, leads to the use of a function, called formula implementation, that maps formulae expressed on the specification to formulae expressed on the program. The formula implementation is used to translate the contract of the contractual specification into formulae on the program. Thus, the formula implementation may be partial on formulae expressed on the specification, but must be total on the contract of the specification.

Formula refinements are submitted to conditions necessary to ensure that the refinement relation is a pre-order. Formula implementations are submitted only to the conditions necessary to ensure that the implementation relation, defined in the next subsection, is compatible with the refinement relation; i.e., an implementation step that follows a refinement process is such that the program which implements the most concrete specification implements the higher-level specifications as well.

Definition 3.2.11 *Formula Implementation.*
Let $CSpec = \langle Spec, \Phi \rangle$ be a contractual specification, $CProg = \langle Prog, \Psi \rangle$ be a contractual program, $\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$ be an implement relation on $CSpec$ and $CProg$. A formula implementation, denoted Λ^I, is a function, univocally defined from λ^I, which maps formulae expressed on $Spec$ into formulae expressed on $Prog$:

$$\Lambda^I : \text{PROP}_{Spec} \rightarrow \text{PROP}_{Prog} \ ,$$

such that:

- Λ^I *maps every property of the contract of* $CSpec$ *to formulae of* $Prog$*, i.e.,* $\Lambda^I(\phi)$ *is defined for every* $\phi \in \Phi$*;*

- *given* λ *a refine relation,* λ^I *an implement relation such that their composition,* $\lambda'^I = \lambda; \lambda^I$*, is defined, and is an implement relation; the formula implementation* Λ'^I*, derived from* λ'^I*, is such that* $\Lambda'^I = \Lambda^I \circ \Lambda$*; where* Λ^I*,* Λ *are the formula implementation and formula refinement derived from* λ^I *and* λ *respectively, and* $\circ$ *is the composition of functions.*

Notation 3.2.12 *Implementation of a Set of Formulae.*
Given $\Lambda^I : \textsc{Prop}_{Spec} \rightarrow \textsc{Prop}_{Prog}$ *a formula implementation, we denote by* $\Lambda^I(\Phi)$ *the image of* Φ *under* Λ^I*.* $\Lambda^I(\Phi) = \{\psi \in \textsc{Prop}_{Prog} \mid \exists \phi \in \Phi \text{ s.t. } \Lambda^I(\phi) = \psi\}$.

3.2.4 Implementation Relation

The implementation relation is defined in the same way as the refinement relation. A contractual program is a correct implementation of a contractual specification if the contract of the program contains the translated contract of the specification. While the refinement relation is a relation on specifications, the implementation relation is a relation on specifications and programs.

Definition 3.2.13 *Implementation of Contractual Specifications via* λ^I*.*
Let $CProg = \langle Prog, \Psi \rangle$ *be a contractual program,* $CSpec = \langle Spec, \Phi \rangle$ *be a contractual specification,* $\lambda^I \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CProg}$ *be an implement relation on* $CSpec$ *and* $CProg$*, and* Λ^I *be the formula implementation univocally defined from* λ^I*.* $\langle Prog, \Psi \rangle$ *is an implementation of* $\langle Spec, \Phi \rangle$ *via* λ^I*, noted* $\langle Spec, \Phi \rangle \rightsquigarrow^{\lambda^I} \langle Prog, \Psi \rangle$*, iff*

$$\Lambda^I(\Phi) \subseteq \Psi.$$

If $\langle Prog, \Psi \rangle$ implements $\langle Spec, \Phi \rangle$, then every model of $\langle Prog, \Psi \rangle$ satisfies $\Lambda^I(\Phi)$. The program has no specific obligation towards properties that are not in the contract of $CSpec$.

Definition 3.2.14 *Implementation Relation.*
The implementation relation, noted $\rightsquigarrow$*, is a relation on contractual specifications and contractual programs:*

$$\rightsquigarrow \subseteq \textsc{CSpec} \times \textsc{CProg} \ ,$$

such that for every $CSpec = \langle Spec, \Phi \rangle \in \textsc{CSpec}$*, and every* $CProg = \langle Prog, \Psi \rangle \in \textsc{CProg}$*, then* $\langle Spec, \Phi \rangle \rightsquigarrow \langle Prog, \Psi \rangle$ *iff*

$\exists \lambda^I \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CProg}$ *an implement relation on* $CSpec$ *and* $CProg$*, s.t.* $\langle Spec, \Phi \rangle \rightsquigarrow^{\lambda^I} \langle Prog, \Psi \rangle$.

3.3 Refinement Process and Implementation

We intend to perform a stepwise refinement process, followed by an implementation phase. The refinement process leads to a chain of contractual specifications $\langle Spec_1, \Phi_1\rangle, \ldots,$ $\langle Spec_n, \Phi_n\rangle$: the first contractual specification, $\langle Spec_1, \Phi_1\rangle$, stands for the most abstract specification and the last specification, $\langle Spec_n, \Phi_n\rangle$, stands for the most concrete one. In the chain, each contractual specification refines its predecessor. Since the refinement relation is a pre-order (see Proposition 3.1.1), every specification is a refinement of the higher-level specifications of the chain, e.g., $\langle Spec_1, \Phi_1\rangle \sqsubseteq \langle Spec_n, \Phi_n\rangle$.

The last contractual specification is considered to be the most concrete one, it should be easily translated into a contractual program $CProg = \langle Prog, \Psi\rangle$, and this program should actually implement the contractual specification, i.e., $\langle Spec_n, \Phi_n\rangle \leadsto \langle Prog, \Psi\rangle$. Since the implementation phase is a final step after a series of refinement steps, it must be compatible with the refinement relation, i.e., the program which implements the most concrete specification implements all the specifications of the chain as well.

This section defines: the refinement process, the implementation step, the compatibility of a refinement relation and an implementation relation. Finally, it shows that the refinement and implementation relations based on contracts are actually compatible.

The following definitions formally define the refinement process and the implementation step.

Definition 3.3.1 *Chain of Contractual Specifications.*
A chain of contractual specifications is an ordered set of contractual specifications:

$$\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_i, \Phi_i\rangle, \ldots, \langle Spec_n, \Phi_n\rangle ,$$

such that each contractual specification refines its predecessor in the chain:

$$\langle Spec_i, \Phi_i\rangle \sqsubseteq \langle Spec_{i+1}, \Phi_{i+1}\rangle, \quad 1 \leq i \leq n-1.$$

Definition 3.3.2 *Refinement Step, Refinement Process.*
A refinement step is the act of replacing a contractual specification by another contractual specification which refines the former contractual specification. A refinement process is a series of consecutive refinement steps leading to a chain of contractual specifications.

Definition 3.3.3 *Implementation.*
Given a chain of contractual specifications, $\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_i, \Phi_i\rangle, \ldots, \langle Spec_n, \Phi_n\rangle$, *the implementation is the replacement of the most concrete contractual specification of the chain by a contractual program which implements this contractual specification:*

$$\langle Spec_n, \Phi_n\rangle \leadsto \langle Prog, \Psi\rangle.$$

The refinement process ends by the implementation of the most concrete contractual specification. The program, implementing the most concrete contractual specification, should be an implementation of every contractual specification of the chain as well, in particular of the most abstract one. It is formalised by the following definition:

Definition 3.3.4 *Compatible Refinement and Implementation Relations.*
Let $\sqsubseteq$ be the refinement relation on contractual specifications, and $\leadsto$ be the implementation relation on contractual specifications and contractual programs. $\sqsubseteq$ and $\leadsto$ are compatible iff for every pair of contractual specifications $\langle Spec', \Phi' \rangle$, $\langle Spec, \Phi \rangle$, and every contractual program $\langle Prog, \Psi \rangle$ the following holds:

$$\langle Spec, \Phi \rangle \sqsubseteq \langle Spec', \Phi' \rangle \wedge \langle Spec', \Phi' \rangle \leadsto \langle Prog, \Psi \rangle \quad \Rightarrow \quad \langle Spec, \Phi \rangle \leadsto \langle Prog, \Psi \rangle.$$

The refinement relation and the implementation relation defined in the previous sections are compatible.

Proposition 3.3.1 *Compatibility of the Refinement and the Implementation Relations.* *The refinement relation on contractual specifications, $\sqsubseteq$, and the implementation relation on contractual specifications and contractual programs, $\leadsto$, are compatible.*

Proof.
Let $CSpec = \langle Spec, \Phi \rangle$, and $CSpec' = \langle Spec', \Phi' \rangle$ be contractual specifications, and $CProg = \langle Prog, \Psi \rangle$ be a contractual program.

$\langle Spec, \Phi \rangle \sqsubseteq \langle Spec', \Phi' \rangle$ implies that there exists $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$, a refine relation such that $\langle Spec, \Phi \rangle \sqsubseteq^{\lambda} \langle Spec', \Phi' \rangle$. Λ, the formula refinement univocally defined from λ, is such that: $\Lambda(\Phi) \subseteq \Phi'$.

$\langle Spec', \Phi' \rangle \leadsto \langle Prog, \Psi \rangle$ implies that there exists $\lambda^I \subseteq \text{ELEM}_{CSpec'} \times \text{ELEM}_{CProg}$ an implement relation such that $\langle Spec', \Phi' \rangle \leadsto^{\lambda^I} \langle Prog, \Psi \rangle$. Λ^I, the formula implementation, univocally defined from λ^I, is such that: $\Lambda^I(\Phi') \subseteq \Psi$.

λ and λ^I can be composed in order to form $\lambda'^I = \lambda; \lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$. λ'^I is actually an implement relation, i.e., it is total on the contract Φ. Indeed, first, λ is total on elements of contract Φ, and $CSpec'$ refines $CSpec$ via λ, thus all elements of contract Φ are related to elements of contract Φ'. Second, λ^I is total on elements of contract Φ', thus all elements of contract Φ are related to elements of contract Ψ by $\lambda; \lambda^I$. Consequently $\lambda'^I = \lambda; \lambda^I$ is an implement relation. By definition, if λ'^I is an implement relation, then Λ'^I, the formula implementation, univocally defined from λ'^I, is such that: $\Lambda'^I = \Lambda^I \circ \Lambda$.

Therefore, $\Lambda(\Phi) \subseteq \Phi'$ implies $\Lambda^I(\Lambda(\Phi)) \subseteq \Lambda^I(\Phi')$. As $\Lambda^I(\Phi') \subseteq \Psi$, we have $\Lambda^I(\Lambda(\Phi)) \subseteq \Psi$. This implies $\langle Spec, \Phi \rangle \leadsto^{\lambda;\lambda^I} \langle Prog, \Psi \rangle$, which in turn implies $\langle Spec, \Phi \rangle \leadsto \langle Prog, \Psi \rangle$. ■

A consequence of this property is that, given two contractual specifications $\langle Spec', \Phi' \rangle$ and $\langle Spec, \Phi \rangle$, with $\langle Spec', \Phi' \rangle$ refining $\langle Spec, \Phi \rangle$, then every program that implements

$\langle Spec', \Phi' \rangle$ implements $\langle Spec, \Phi \rangle$ too. Thus the set of programs implementing $\langle Spec', \Phi' \rangle$ is included in the set of programs implementing $\langle Spec, \Phi \rangle$.

A contractual program, implementing the most concrete contractual specification of a chain of specifications, satisfies (via the formula implementation) the whole set of properties of this contractual specification. Due to the compatibility of the refinement and the implementation relations, and due to the transitivity of the refinement relation, this contractual program satisfies the contract of each of the other contractual specifications of the chain as well, and thus is an implementation of every contractual specification of the chain.

Corollary 3.3.1 *Compatible Refinement Process and Implementation.*
Let $\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_i, \Phi_i \rangle, \ldots, \langle Spec_n, \Phi_n \rangle$ *be a chain of contractual specifications. If* $\langle Prog, \Psi \rangle$ *is an implementation of* $\langle Spec_n, \Phi_n \rangle$*, then* $\langle Prog, \Psi \rangle$ *is an implementation of all the contractual specifications of the chain:*

$$\langle Spec_n, \Phi_n \rangle \rightsquigarrow \langle Prog, \Psi \rangle \quad \Rightarrow \quad \langle Spec_i, \Phi_i \rangle \rightsquigarrow \langle Prog, \Psi \rangle, \quad 1 \leq i \leq n-1.$$

Proof.
Due to the transitivity of $\sqsubseteq$, $\langle Spec_n, \Phi_n \rangle$ refines every contractual specification in the chain:

$$\langle Spec_i, \Phi_i \rangle \sqsubseteq \langle Spec_n, \Phi_n \rangle, \quad 1 \leq i \leq n-1.$$

$\langle Prog, \Psi \rangle$ implements $\langle Spec_n, \Phi_n \rangle$, i.e., $\langle Spec_n, \Phi_n \rangle \rightsquigarrow \langle Prog, \Psi \rangle$. The compatibility between $\sqsubseteq$ and $\rightsquigarrow$ implies:

$$\langle Spec_i, \Phi_i \rangle \rightsquigarrow \langle Prog, \Psi \rangle, \quad 1 \leq i \leq n-1.$$

■

Summary

Figure 3.1 shows a refinement process followed by an implementation phase, and depicts the proofs necessary to ensure that the whole process is correct.

The refinement process starts with the pair $CSpec_0 =< Spec_0, \Phi_0 >$ as the most abstract contractual specification. A first refinement leads to the pair $CSpec_1 =< Spec_1, \Phi_1 >$; the refinement process continues and reaches the pair $CSpec_n =< Spec_n, \Phi_n >$. Finally, the implementation phase provides the contractual program $CProg =< Prog, \Psi >$.

Horizontal proofs ensure that every pair $CSpec_i =< Spec_i, \Phi_i > (0 \leq i \leq n)$ obtained during the refinement process is actually a contractual specification, and that the $CProg =< Prog, \Psi >$ is actually a contractual program. Therefore, it is necessary to show:

$$Mod_{Spec_i} \models \Phi_i \ (0 \leq i \leq n), \text{ and}$$

$$Mod_{Prog} \models \Psi.$$

Vertical proofs assert the correctness of the refinement steps, by requesting:

$$\Phi_i \subseteq \Phi_{i+1} \ (0 \leq i \leq n-1).$$

Finally implementation proof ensures that the contractual program $CProg =< Prog, \Psi >$ correctly implements the contractual specification $CSpec_n =< Spec_n, \Phi_n >$, and hence every contractual specification $CSpec_i$ $(0 \leq i \leq n)$. It requests, similarly to vertical proof, that:

$$\Phi_n \subseteq \Psi.$$

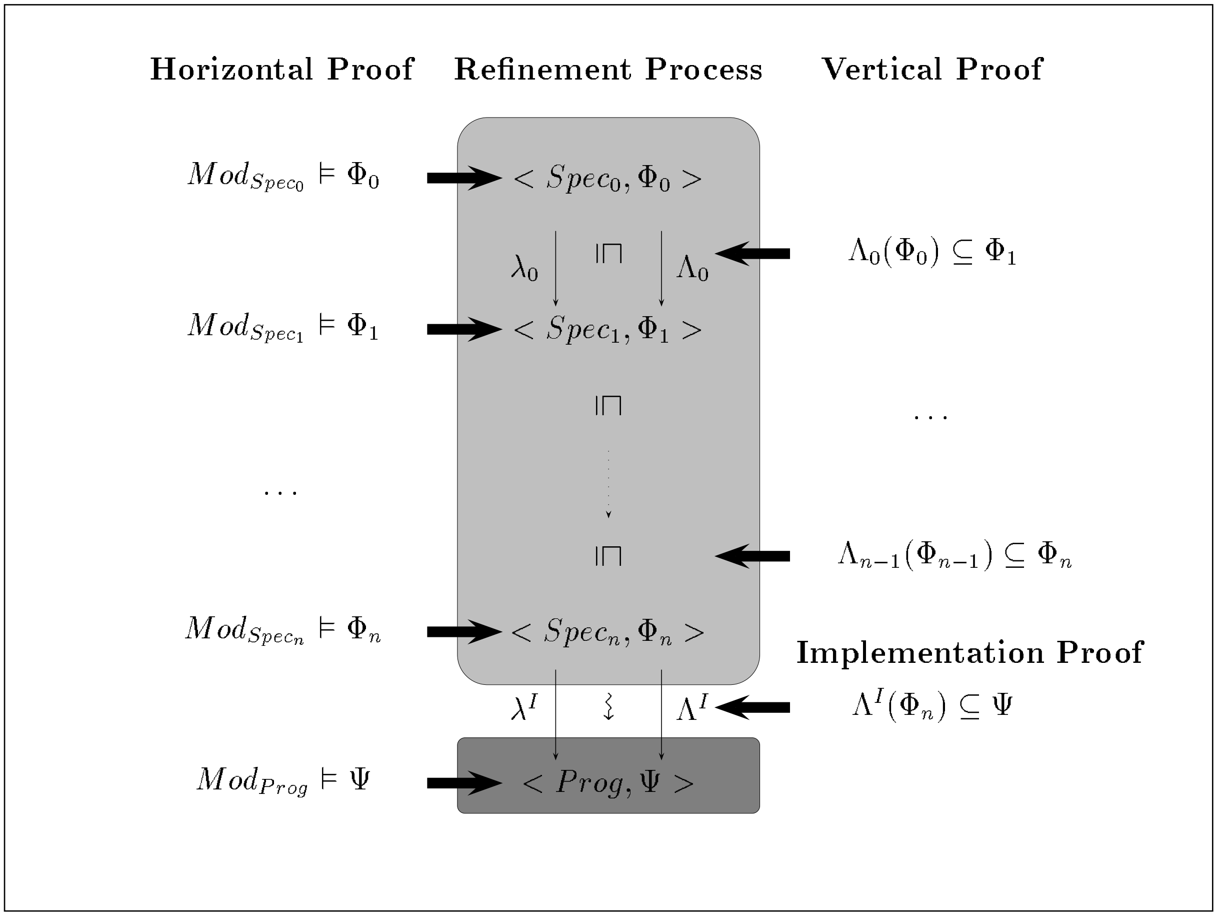


Figure 3.1: Refinement Process, Implementation and Proofs

3.4 Compositional Refinement and Implementation

When the considered formal specifications language is such that there exists a compositional operator that enables a specification to be considered as the composition of several sub-specifications (also called components), and when the refinement of components is defined, then we can consider it to be a compositional refinement.

A refinement is said to be compositional wrt a compositional operator, or to be a congruence wrt a compositional operator; or a compositional operator is said to be monotonic wrt the refinement relation, if:

> *Given a high-level specification made of the composition of several components, the replacement of each component, by a lower-level component refining it, leads to a lower-level specification which is a refinement of the higher-level one.*

If, in addition, the programming language defines a compositional operator that enables a program to be considered as the composition of several sub-programs (also called components), and if the implementation of components is defined, a compositional implementation can be considered.

An implementation is said to be compositional, or to be a congruence wrt a compositional operator on the specifications and a compositional operator on the programs, if:

> *Given a specification made of the composition of several components, the replacement of each component, by a program implementing it, leads to a program which is an implementation of the specification.*

First this section defines compositional contractual specifications, and the compositional refinement of contractual specification. Second, it defines compositional contractual programs, and the compositional implementation of contractual specification. Finally, it discusses different ways of achieving the composition of contracts and the composition of specifications.

Compositional Contractual Specification

As this chapter does not consider a particular formal specifications language, we will not discuss any particular compositional operator. We will assume the existence of a compositional operator that applies to a set of specifications. The composition of the contracts depends on the composition of the specifications. Thus, we assume the existence of a compositional operator that is able to return from a set of contractual specifications a compound contractual specification, whose specification part is the composition of the specification parts and whose contract is the composition of the contract parts.

Definition 3.4.1 *Compositional Operator on Contractual Specifications.*
A k-ary compositional operator, denoted f, is a partial function on contractual specifications:

$$f : \mathrm{CS{\small PEC}}^k \to \mathrm{CS{\small PEC}}.$$

A k-ary compositional operator is not necessarily a total function, since any set of k contractual specifications cannot be composed to form a compound contractual specification.

Definition 3.4.2 *Compositional Contractual Specification.*
Let $\langle Spec_i, \Phi_i \rangle$, $1 \leq i \leq k$, be k contractual specifications. Let $f : \text{CSPEC}^k \rightarrow \text{CSPEC}$ be a k-ary compositional operator on contractual specifications. A compositional contractual specification is a contractual specification given by the composition of $\langle Spec_i, \Phi_i \rangle$, $1 \leq i \leq k$, by f:

$$f(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_n, \Phi_k \rangle).$$

According to this definition, components are themselves contractual specifications. Thus, the refinement of a component is defined as the refinement of a contractual specification, and the implementation of a component is defined as the implementation of a contractual specification.

Compositional Refinement

The refinement of contractual specifications is a congruence wrt a k-ary compositional operator on contractual specifications if, given a high-level compositional contractual specification, the lower-level contractual specification, obtained by replacing each high-level contractual component by a lower-level component, is a refinement of the higher-level contractual specification.

Definition 3.4.3 *Compositional Refinement.*
Let $f : \text{CSPEC}^k \rightarrow \text{CSPEC}$ be a k-ary compositional operator on contractual specifications. Let $\langle Spec_i, \Phi_i \rangle$, $\langle Spec'_i, \Phi'_i \rangle$, $1 \leq i \leq k$ be contractual specifications. The refinement relation on contractual specifications, $\sqsubseteq$, is a congruence wrt f, iff:

$$\begin{array}{l} \langle Spec_i, \Phi_i \rangle \sqsubseteq \langle Spec'_i, \Phi'_i \rangle, 1 \leq i \leq k \quad \Rightarrow \\ f(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_k, \Phi_k \rangle) \sqsubseteq f(\langle Spec'_1, \Phi'_1 \rangle, \ldots, \langle Spec'_k, \Phi'_k \rangle). \end{array}$$

Compositional Contractual Program

We assume the existence of a compositional operator on contractual programs. Like the compositional operator on contractual specifications, so the compositional operator on contractual programs is a partial function, since any set of programs cannot be composed in order to form a compound program.

Definition 3.4.4 *Compositional Operator on Contractual Programs.*
A k-ary compositional operator, denoted g, is a partial function on contractual programs:

$$g : \text{CPROG}^k \rightarrow \text{CPROG}.$$

Definition 3.4.5 *Compositional Contractual Program.*
Let $\langle Prog_i, \Psi_i \rangle$, $1 \leq i \leq k$, *be* k *contractual programs. Let* $g : \text{CProg}^k \rightarrow \text{CProg}$ *be a compositional operator on contractual programs. A compositional contractual program is a contractual program given by the composition of* $\langle Prog_i, \Psi_i \rangle$, $1 \leq i \leq k$, *by* g:

$$g(\langle Prog_1, \Psi_1 \rangle, \ldots, \langle Prog_n, \Psi_k \rangle).$$

Compositional Implementation

The implementation of contractual specifications is a congruence wrt a k-ary compositional operator on contractual specifications and a k-ary compositional operator on contractual programs if, given a compositional contractual specification, the contractual program, obtained by replacing each contractual component by a program implementing the component, is an implementation of the compositional contractual specification.

Definition 3.4.6 *Compositional Implementation.*
Let $f : \text{CSpec}^k \rightarrow \text{CSpec}$ *be a* k*-ary compositional operator on contractual specifications, and* $g : \text{CProg}^k \rightarrow \text{CProg}$ *be a* k*-ary compositional operator on contractual programs. Let* $\langle Spec_i, \Phi_i \rangle$, $1 \leq i \leq k$, *be* k *contractual specifications, and* $\langle Prog_i, \Psi_i \rangle$, $1 \leq i \leq k$, *be* k *contractual programs. The implementation relation on contractual specifications and contractual programs,* $\rightsquigarrow$, *is a congruence wrt* f *and* g *iff:*

$$\begin{array}{l} \langle Spec_i, \Phi_i \rangle \rightsquigarrow \langle Prog_i, \Psi_i \rangle, 1 \leq i \leq k \quad \Rightarrow \\ f(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_k, \Phi_k \rangle) \rightsquigarrow g(\langle Prog_1, \Psi_1 \rangle, \ldots, \langle Prog_k, \Psi_k \rangle). \end{array}$$

Refinement Process and Implementation

When the refinement relation is a congruence wrt f a compositional operator on contractual specifications, and the implementation relation is a congruence wrt f and to g, a compositional operator on contractual programs, then a compositional program implementing, component by component, a low-level compositional specification implements as well component by component any higher-level compositional specification that the lower-level one refines.

Corollary 3.4.1 *Compatible Compositional Refinement and Implementation.*
Let $f : \text{CSpec}^k \rightarrow \text{CSpec}$ *be a* k*-ary compositional operator on contractual specifications. Let* $g : \text{CProg}^k \rightarrow \text{CProg}$ *be a* k*-ary compositional operator on contractual programs. Let* $\langle Spec_i, \Phi_i \rangle$, $\langle Spec_i', \Phi_i' \rangle$, $1 \leq i \leq k$, *be contractual specifications, and* $\langle Prog_i, \Psi_i \rangle$, $1 \leq i \leq k$, *be* k *contractual programs.*

If $\sqsubseteq$ *is a congruence wrt* f, *and* $\rightsquigarrow$ *is a congruence wrt* f *and* g, *then the following holds:*

$$\begin{array}{l} \langle Spec_i, \Phi_i \rangle \sqsubseteq \langle Spec_i', \Phi_i' \rangle \wedge \langle Spec_i', \Phi_i' \rangle \rightsquigarrow \langle Prog_i, \Psi_i \rangle, 1 \leq i \leq k \quad \Rightarrow \\ f(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_n, \Phi_k \rangle) \rightsquigarrow g(\langle Prog_1, \Psi_1 \rangle, \ldots, \langle Prog_n, \Psi_k \rangle). \end{array}$$

Proof.
The compatibility between $\sqsubseteq$ and $\leadsto$ implies that: $\langle Spec_i, \Phi_i \rangle \leadsto \langle Prog_i, \Psi_i \rangle, 1 \leq i \leq k$, since $\langle Spec_i, \Phi_i \rangle \sqsubseteq \langle Spec_i', \Phi_i' \rangle \wedge \langle Spec_i', \Phi_i' \rangle \leadsto \langle Prog_i, \Psi_i \rangle, 1 \leq i \leq k$. The fact that $\leadsto$ is a congruence wrt f and g implies the result. ■

Remark 3.4.7 *Fiadeiro [35] shows that it is not sufficient that a component program satisfies its specification to ensure that the composition of the component programs satisfies the composition of their respective specifications. It is necessary to have a functor from the category of programs to the category of specifications. Thus, the compositional refinement or compositional implementation are not guaranteed for any formal specifications language, programming language, refinement relation, and implementation relation.*

Compositional Operators

As mentioned above, we did not choose a particular operator for composing either the specifications or the contracts. Abadi and Lamport [3] give a method for deducing properties of a system by reasoning about its components: every component is specified by a TLA formula, the parallel composition is represented by the conjunction of the formulae. If contracts are given by TLA formulae, the conjunction of the contracts could be the compositional operator.

Wirsing [61] distinguishes the *structured* specifications from the *parameterised* specifications. Structured specifications are obtained with specification-building operators (abstractors and constructors of section 2.3.1). These operators are necessarily monotonic wrt the refinement relation, thus the fact that the refinement relation is compositional follows immediately. Hierarchical specifications are structured specifications obtained with a particular specification-building operator. In order to form a hierarchical specification, a specification is extended with an incomplete specification, i.e., all the elements used in the specification are not defined in the specification. The monotonicity of the operator ensures that if an algebraic specification SP_1 refines an algebraic specification SP_2, then the hierarchical specification extending SP_1 with an incomplete specification refines that extending SP_2 with the *same* incomplete specification. The refinement of the incomplete specification is not considered.

Parameterised specifications $P(SP)$ are not obtained with specification-building operators. The refinement of a parameterised specification is defined in the following way: P refines P_1 if for any actual parameter SP_A, then $P(SP_A)$ refines $P_1(SP_A)$. It is interesting to note that, even though the P part of a parameterised specification is an incomplete specification, its refinement is defined.

We apply these definitions of compositional refinement to contractual specifications. When contractual specifications are complete, i.e., all the elements used in the specification are defined in the specification, then the compositional refinement presented in this section can be compared to the refinement of structured specifications. Indeed, in this case, the refinement of incomplete contractual specifications is not defined. The compositional operator f on contractual specification may freely add an incomplete contractual specification to a k-tuple of complete contractual specification in order to form a new complete contractual specification. The complete contractual specification obtained with f is considered for the refinement.

When contractual specifications are allowed to be incomplete, the compositional refinement of contractual specifications can be compared to the refinement of parameterised specifications. Indeed, the refinement of incomplete components is defined, and a k-tuple of contractual specifications may contain incomplete components.

Remark 3.4.8 *Chapters 5 and 6 define a compositional CO-OPN/2 refinement and a compositional CO-OPN/2 implementation in a way similar to the refinement of hierarchical specifications.*

3.5 Discussion

The previous sections have lead to the definition of a theory of refinement based on the preservation of properties explicitly collected in what we have called a contract. They also lead, with similar definitions, to the implementation of specifications by programs satisfying the properties of interest of the specifications.

This section is devoted to a deeper understanding of the use of a contract in a development process. It discusses: the syntactical and the semantical requirements implied by a refinement constrained by properties; correct and incorrect refinements; the evolution of the contract during a refinement process and the implementation phase; the way the evolution of the contracts restricts the set of programs implementing the most abstract contractual specification; and some advantages and disadvantages due to the use of contracts.

3.5.1 Syntactical Conditions

The refine relation conveys the syntactical requirements of the refinement, and has an impact on whether the structure of specifications will be preserved. Indeed, during the refinement process, the syntactical obligations of a lower-level contractual specification towards a higher-level contractual specification, are reduced to the existence of a refine relation, which ensures that every abstract element that takes part in the contract is in relation with at least one concrete element.

The theory presented in this chapter does not constrain the refine relation. However, when the theory is practically applied to a specifications language, the refine relation is submitted to specific constraints (partial, total, functional, injective or surjective, on observable elements only, etc). Therefore, the refine relation implies structural constraints on lower-level contractual specifications. For instance, a refine relation which is a total function forces the structure of a high-level specification to be totally maintained by a lower-level specification, even though it authorises the lower-level specification to add new components. On the contrary a refine relation which is a partial, surjective function does not preserve the whole high-level structure in its entirety, and prevents the lower-level specification to add new components.

The same discussion applies for the the implement relation, since it is very similar to a refine relation.

3.5.2 Semantical Conditions

The semantical requirements of the definitions of refinement and implementation are conveyed by the contract. Indeed, the obligations of the low-level specification wrt the higher-level one are restricted to the preservation of the contract only. If a property of high-level specification *is part* of the contract, then, the translation of this property *is a property* of the lower-level specification, i.e., it is satisfied by every model of the lower-level specification. If a property of a high-level specification is *not part* of the contract, then, the translation of this property is a formula expressed on the lower-level specification which is *not necessarily satisfied* by all the models of the lower-level specification.

Therefore, we can say that a high-level contractual specification and a lower-level contractual specification, which correctly refines it, are *equivalent modulo the contract*. Indeed, the contract is the only part of the behaviour of the high-level contractual specification, that is ensured to be part of the behaviour of the lower-level contractual specification.

Classes of Properties

We have seen in Chapter 2 that the definitions of refinement usually require two kinds of semantical obligations: input/output behaviour preservation; and whole behaviour preservation. A contract may contain properties of different classes:

- Functional Properties.
  These properties relate to the essential functionality expected by the system. They can be seen as a kind of input/output behaviour. For instance, the system functionality consists of computing sums.

- Non-Functional Properties.
  The functionality is a small part of the whole behaviour of the system. The non-functional properties describe the rest of the behaviour. They encompass dependability constraints (fault-tolerance, error recovery, ...), as well as performance constraints (high degree of parallelism, time taken for a computation, ...), or architectural constraints (client/server, ...).

- Refinement choices.
  Some properties of the contract reflects refinement choices performed during the refinement process. For instance, the introduction of a client/server architecture.

- Visible or not.
  Some properties may be observable for a user: given an input, a certain output is obtained; or a given sequence of operations can be performed while another cannot; etc. Some properties may be non observable: if the underlying architecture of the system is a client/server architecture, the user of the client system cannot know if requests are made to the server, or if the system computes everything itself.

Refinement Depends on the Logic

We have seen that the contract decides on the kind of refinement, e.g., a refinement which preserves input/output behaviour or a refinement which preserves the whole behaviour. The contract is made of properties expressed in a given logic. Depending on the kind of logic used (classic, modal, temporal), and depending on the expressivity of the logic wrt the formal specifications language, it is not possible to express every property that the specification satisfies. Thus, it is not possible to define every kind of refinement. A logic which is more expressive enables to discriminate more finely the specifications wrt the refinement relation.

For a given logic and a specification $Spec$, the strongest refinement is obtained with the maximal contract, i.e., $\Phi = \Phi_{Spec}$. If the logic is such that Φ is able to describe very precisely behavioural details of $Spec$, the number of contractual specifications which are able to refine $Spec$ will be rather low. If the logic is such that Φ is able to give only rough information on $Spec$, then the number of contractual specifications that are able to refine $Spec$ will be greater than that obtained in the first case.

The use of a temporal logic, instead of a classical logic, is best suited for expressing formulae on specifications languages whose semantics is based on events and states, since temporal logics provide a means to assert if a formula is true at a given point (state) of the execution of the system. Moreover, temporal logics are traditionally used in addition to process algebra in order to express essential requirements of a process. They are also used to express the semantics of object-oriented specifications languages (TROLL, VDM^{++}).

Weak and Strong Forms of Refinement and Implementation

Depending on the size of the set of properties that must be preserved between a specification and its refinement, the refinement relation will be more or less constrained. At one end of this spectrum, we find a refinement relation imposing that *all* the properties of the specification to refine must be preserved; this is the strongest refinement relation: only *few* specifications can refine the given specification. At the other end, we find a refinement relation where *no* properties at all have to be preserved; this is the weakest refinement relation: *every* specification refines the given specification. In between, we have refinement relations imposing that *some* properties (or some properties of a given *class* of properties) have to be preserved: *some* specifications refine the given specification.

The weak or strong form of the refinement depends as well on the kind of logic used, since the set of properties that can be expressed on a specification depends on the logic.

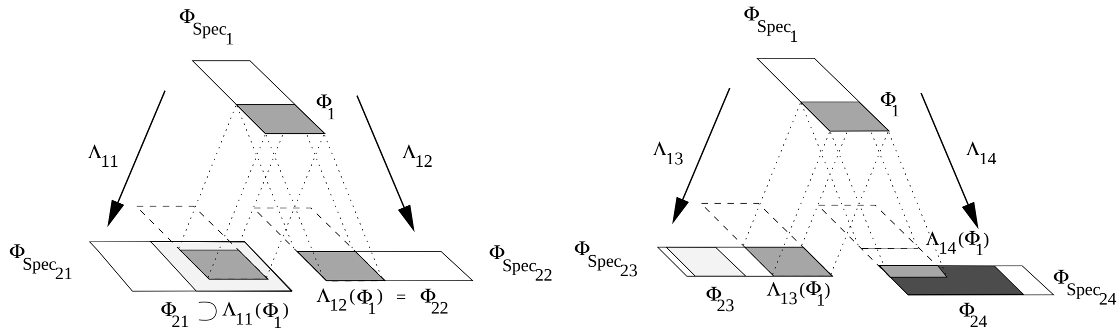
3.5.3 Correct and Incorrect Refinements

A refinement is correct if either the translated contract is equal to the lower-level contract, or is a strict subset. In both cases, the translated contract is also part of the set of all properties of the lower-level specification. A refinement is incorrect if either the translated contract is satisfied by the models of the lower-level specification - but is not included into the lower-level contract -, or the translated contract is not satisfied by all models of the lower-level specification. In the last case, the translated contract is not part of the set of all properties of the lower-level specification. In all case, the set of high-level properties that are not in the contract may be totally, partially or not at all satisfied by all models of the lower-level specification.

Figure 3.2 depicts these four cases. The left part of the figure shows two correct refinements while the right part shows two incorrect ones. In the examples of this figure, the set of high-level properties that are not in the contract is not at all satisfied by all models of the lower-level specification.

In the left part of the figure, a high-level contractual specification $\langle Spec_1, \Phi_1 \rangle$ is refined by two different contractual specifications: $\langle Spec_{21}, \Phi_{21} \rangle$ and $\langle Spec_{22}, \Phi_{22} \rangle$. Φ_{Spec_1} denotes the set of properties of $Spec_1$, i.e., the set of all formulae satisfied by the models of $Spec_1$. Similarly, $\Phi_{Spec_{2i}}$, $1 \leq i \leq 2$, denotes the set of properties of $Spec_{2i}$. The formula refinement Λ_{11} translates every property of $Spec_1$ into a formula of $Spec_{21}$, and the formula refinement Λ_{12} translates every property of $Spec_1$ into a formula of $Spec_{22}$. The formula refinement Λ_{11} translates the contract Φ_1 of $Spec_1$ into a part of the contract of $Spec_{21}$, and hence into a strict subset of $\Phi_{Spec_{21}}$. Thus, contractual specification $\langle Spec_{21}, \Phi_{21} \rangle$ is a correct refinement of $\langle Spec_1, \Phi_1 \rangle$. The formula refinement Λ_{12} translates the contract Φ_1 of $Spec_1$ into Φ_{22}. Thus, contractual specification $\langle Spec_{22}, \Phi_{22} \rangle$ is a correct refinement of $\langle Spec_1, \Phi_1 \rangle$.

In the right part of the figure, the same high-level contractual specification $\langle Spec_1, \Phi_1 \rangle$ is refined by: $\langle Spec_{23}, \Phi_{23} \rangle$ and $\langle Spec_{24}, \Phi_{24} \rangle$. The formula refinement Λ_{13} translates the contract Φ_1 of $Spec_1$ into a subset of $\Phi_{Spec_{23}}$. This means that every model of $Spec_{23}$ satisfies $\Lambda_{13}(\Phi_1)$. However, the contract Φ_{23} does not contain $\Lambda_{13}(\Phi_1)$, thus a subsequent refinement will not be obliged to preserve $\Lambda_{13}(\Phi_1)$. Therefore, contractual specification $\langle Spec_{23}, \Phi_{23} \rangle$ is *not* a correct refinement of $\langle Spec_1, \Phi_1 \rangle$. The formula refinement Λ_{14} translates the contract Φ_1 of $Spec_1$ into a set of formulae of $Spec_{24}$ which is not completely a subset of $\Phi_{Spec_{24}}$, thus the part of the translated contract which is not in $\Phi_{Spec_{24}}$ is not satisfied by all models of $Spec_{24}$. Therefore, the contractual specification $\langle Spec_{24}, \Phi_{24} \rangle$ is *not* a correct refinement of $\langle Spec_1, \Phi_1 \rangle$.



(a) Correct (b) Incorrect

Figure 3.2: Correct and Incorrect Refinements

Figure 3.3 explains why a lower-level specification, whose set of properties contains the translated contract of a higher-level specification, but whose contract does not contain the translated contract of the higher-level specification, cannot be a correct refinement of the higher-level specification.

Contractual specification $\langle Spec_1, \Phi_1 \rangle$ is "refined" by contractual specification $\langle Spec_2, \Phi_2 \rangle$. The models of $\langle Spec_2, \Phi_2 \rangle$ satisfy the translated contract of the higher-level specification, since $\Lambda_1(\Phi_1) \subseteq \Phi_{Spec2}$. However, Φ_2 does not contain $\Lambda_1(\Phi_1)$. Thus, if we consider $\langle Spec_2, \Phi_2 \rangle$ to be a correct refinement of $\langle Spec_1, \Phi_1 \rangle$, and if we perform a subsequent refinement step, we may reach the lower-level contractual specification $\langle Spec_3, \Phi_3 \rangle$ whose models do not satisfy $\Lambda_2(\Lambda_1(\Phi_1))$, since $\Lambda_2(\Lambda_1(\Phi_1)) \nsubseteq \Phi_{Spec_3}$. Thus the original contract has not been preserved. Therefore, even though the models of $\langle Spec_2, \Phi_2 \rangle$ satisfy the original contract, $\langle Spec_2, \Phi_2 \rangle$ is *not* a correct refinement since Φ_2 breaks the preservation of the original contract.

Figure 3.4 shows the case of a low-level contractual specification $\langle Spec_2, \Phi_2 \rangle$, that refines a high-level contractual specification $\langle Spec_1, \Phi_1 \rangle$ but not $\langle Spec_1, \Phi_1' \rangle$, even though the two high-level contractual specifications have the same specification part ($Spec_1$).

3.5.4 Evolution of the Contract during the Refinement Process

When they are necessary for the final implementation, refinement choices will be indicated in the contract. For instance, a refinement process starts with a high-level specification whose contract mentions only the basic functionality. If the final implementation has to be built according to the client/server paradigm, then at some moment in the refinement process it will be necessary to specify the system in that way. If the contract does not require the client/server architecture, then any subsequent refinement step and the final

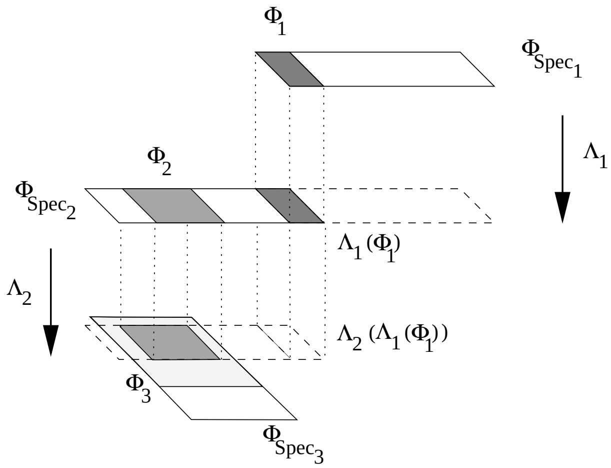


Figure 3.3: Loss of the Contract during an Incorrect Refinement Process

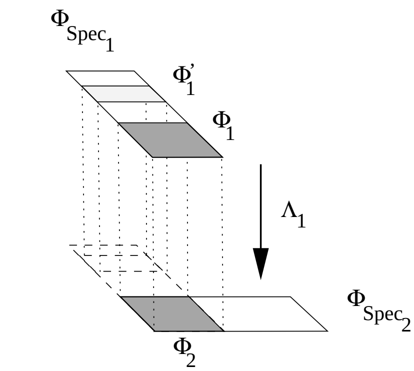


Figure 3.4: Correct Refinement Depends on the High-Level Contract

implementation will not have to follow the client/server architecture. If, on the contrary, it is essential for the final implementation to follow a client/server architecture, the contract will require it. Complexity, necessary for the final implementation, is added at each step, and the growth of the contract reflects the essential complexity.

The growth of the contract can also be seen as a means to measure the degree of refinement reached. Basically, the more the contract grows, the more the lower-level specifications are fine grained, or conversely the higher-level specification are coarse grained wrt the contract.

Let $\langle Spec_1, \Phi_1 \rangle$ be a contractual specification and $\langle Spec_2, \Phi_2 \rangle$ be a refinement of $\langle Spec_1, \Phi_1 \rangle$. We will say the the contract Φ_2 is *bigger* than Φ_1 if $\Lambda_1(\Phi_1) \subset \Phi_2$. The contract Φ_2 is the *same* as Φ_1 if $\Lambda_1(\Phi_1) = \Phi_2$.

During a refinement step two cases occur: either the contract of the lower-level specification is bigger than the contract of the higher-level specification, or it the same. The contract cannot decrease, otherwise it is not a correct refinement step, i.e., if $\Lambda(\Phi_1) \not\subseteq \Phi_2$ then $\langle Spec_2, \Phi_2 \rangle$ is not a correct refinement of $\langle Spec_1, \Phi_1 \rangle$.

When the contract grows, the models of a lower-level contractual specification, refining a higher-level contractual specification, satisfy entirely the translated contract of the higher-level contractual specification, *plus* properties of their own. The growth of the contract indicates refinement choices made at each step of the refinement process. The added properties, i.e., $\Phi_2 - \Lambda_1(\Phi_1)$ represent refinement choices that have been made at this step, and that *must be kept* in subsequent refinement steps. When the contract grows, we say the the lower-level specification is more precise than the higher-level specification wrt the contract. The growth of the contract can be used to measure the degree of refinement. If the low-level contract is bigger than the higher-level contract, then the high-level specification is coarser grained wrt the low-level specification, or the low-level specification is finer grained wrt the higher-level specification.

When the contract remains the same, the models of a lower-level specification, refining a higher-level specification, satisfy at least the translated high-level contract, and probably other properties of their own, but further specifications in the refinement process are not required to satisfy these extra properties, so that these properties will not be maintained till the implementation. In this case, on the basis of the contract alone, we cannot say if the low-level specification is finer grained than the higher-level one.

Figure 3.5 shows an example of the evolution of the contract during a refinement process leading to a chain of specification made of three contractual specifications $\langle Spec_1, \Phi_1\rangle, \langle Spec_2, \Phi_2\rangle, \langle Spec_3, \Phi_3\rangle$. The example chosen here is such that at each step the translated contract is a strict subset of the lower-level contract $\Lambda_i(\Phi_i) \subset \Phi_{i+1}$, $1 \leq i \leq 2$; thus the lower-level contract is bigger than the higher-level one. At each step the contract grows. The part of the high-level properties which is not in the contract is not preserved by the lower-level specification $(\Phi_{Spec_i} - \Phi_i) \not\subseteq \Phi_{Spec_{i+1}}$, $1 \leq i \leq 2$. According to the methodology, the contract of the most concrete contractual specification $\langle Spec_3, \Phi_3\rangle$ is given by $\Phi_{Spec_3} = \Phi_3$. Thus, the implementation requires that the program must satisfy the whole set of properties Φ_{Spec_3} of the most concrete specification. In this example, the contract of the program Ψ contains this set of properties: $\Phi_{Spec_3} \subset \Psi$; the contract of the program is bigger and the program has properties of its own that are not properties of $Spec_3$.

3.5.5 Evolution of Programs

We consider a chain of specifications obtained by a refinement process: $\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_n, \Phi_n\rangle$; and the sets of programs implementing each specification. We will call $\textsc{Prog}_i$ the set of programs that correctly implement $\langle Spec_i, \Phi_i\rangle$, $1 \leq i \leq n$. If the contract grows at each step, then the sets of programs $\textsc{Prog}_{i+1} \subset \textsc{Prog}_i$, $1 \leq i \leq n-1$, since the compatibility between the implementation and the refinement relation of Proposition 3.3.4 imply that any program implementing a low-level specification implements also the higher-level specifications. Thus, the number of programs decreases at each step. If the contract remains the same at each step, then $\textsc{Prog}_{i+1} = \textsc{Prog}_i$, $1 \leq i \leq n-1$, since nothing in the contract is added, that can specialise the program.

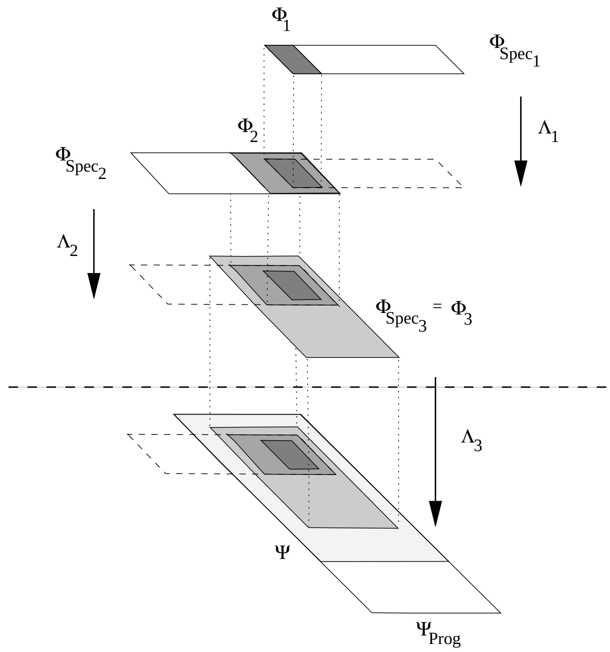


Figure 3.5: Evolution of Contract during the Refinement Process and Implementation

Figure 3.6 depicts the reduction of the number of programs implementing the contractual specification during the refinement process. For the scope of this example, we assume that exactly three contractual programs are able to implement $\langle Spec_1, \Phi_1\rangle$, since $\Lambda_{1i}(\Phi_1) \subseteq \Psi_i$, $1 \leq i \leq 3$. In order to be concise, the figure depicts a special case, where $\Lambda_{1i}(\Phi_1)$ are all equal, $1 \leq i \leq 3$. However, they could be completely different, e.g., with no intersection at all.The refinement process leads to $\langle Spec_2, \Phi_2\rangle$, which is a refinement of $\langle Spec_1, \Phi_1\rangle$, and whose contract Φ_2 is bigger than the translated contract of $\langle Spec_1, \Phi_1\rangle$. At this point of the refinement process, only two contractual programs are able to implement $\langle Spec_2, \Phi_2\rangle$: $\langle Prog_2, \Psi_2\rangle$, and $\langle Prog_3, \Psi_3\rangle$. $\langle Prog_1, \Psi_1\rangle$ cannot implement $\langle Spec_2, \Phi_2\rangle$, because Ψ_1 does not contain $\Lambda_{21}(\Phi_2)$. Finally, the refinement process leads to a third contractual specification, $\langle Spec_3, \Phi_3\rangle$, the contract Φ_3 is bigger than Φ_2, and a unique implementation is given by $\langle Prog_3, \Psi_3\rangle$.

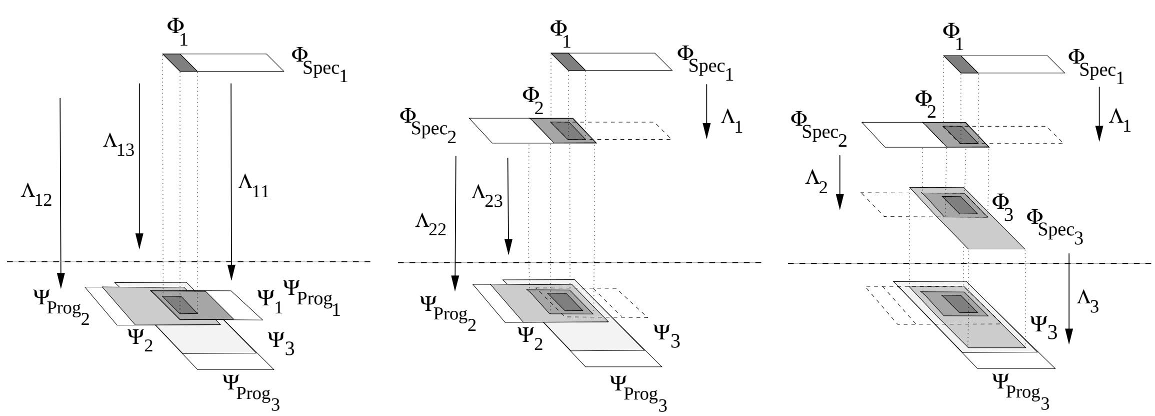


Figure 3.6: Reduction of the Set of Programs During the Refinement Process

Figure 3.7 shows another example, where the lower-level contracts are not bigger than the higher-level ones. The set of programs, implementing every contractual specification obtained during the refinement process, does not change.

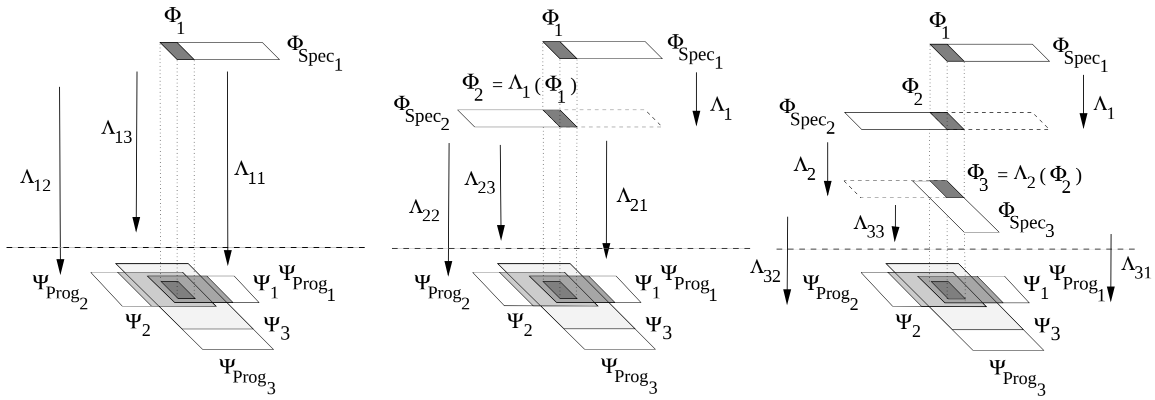


Figure 3.7: Immutable Set of Programs During the Refinement Process

As for the previous example, we assume that exactly three contractual programs are able to implement $\langle Spec_1, \Phi_1\rangle$. The refinement process leads to $\langle Spec_2, \Phi_2\rangle$. It is a refinement of $\langle Spec_1, \Phi_1\rangle$, whose contract is the same as those of $\langle Spec_1, \Phi_1\rangle$: $\Phi_2 = \Lambda_1(\Phi_1)$. Thus, every program implementing $\langle Spec_1, \Phi_1\rangle$ implements $\langle Spec_2, \Phi_2\rangle$ as well. $\langle Spec_3, \Phi_3\rangle$ is a refinement of $\langle Spec_2, \Phi_2\rangle$ with the same contract, thus the same set of programs implements $\langle Spec_3, \Phi_3\rangle$.

3.5.6 Advantages of the Use of Contracts

Contracts may be used during the whole software life cycle; they correspond to pragmatic refinement and implementation processes; they are useful for proof purposes; and they provide a more general theory of refinement and implementation.

Software Life Cycle

During the *analysis phase*, the requirements are formally expressed with a first contractual specification. The contract part stands for the requirements, while the whole specification stands for an abstract solution that enables the requirements to be fulfilled.

During the *design phase*, the abstract solution is progressively replaced by more concrete solutions; it is the refinement process. The contract guides each refinement step: it guarantees that the requirements of the previous step are maintained, and enables to integrate new requirements (i.e., new design constraints).

Finally, during the *implementation phase*, a program replaces the most concrete specification obtained during the previous phase. The contracts ensure that the program fulfils the requirements of the most concrete specification, and hence of the most abstract one.

Practical Refinement and Implementation

Due to the choice of the formal specifications language, a system is specified in such a way that its models exhibit a certain behaviour. It is not always necessary or possible that a lower-level specification or a program, refining or implementing the specification respectively, exhibits exactly the same behaviour.

For instance, a formal specifications language, may have a semantics - given by a transition system - that allows parallel operations to be events of the transition system. For practical reasons, the program cannot be implemented on a parallel machine, but only on a sequential machine. If the implementation phase requires that the whole behaviour of the specification must be kept by the program, then, if only a sequential machine is available, no program can be considered as a correct implementation. Another example is provided by a specifications language whose syntax and semantics are such that a specification becomes complex not because the system itself is complex but because the specifications language does not allow simple formulation of the problem. If a programming language allows more expressivity than the formal language, then a more concise program will implement the specification. In this case, a complex program sticking to the specification is not necessary.

The use of the contract alleviates the refinement process and the implementation phase, since it allows both the program and the lower-level specifications to take certain freedom wrt higher-level specifications. The contract conveys exactly the part of the high-level specification that must not be forgotten in a lower-level specification. For instance, the specifier is free to change the architecture of the system, to change algorithms used, provided these changes do not interfere with the preservation of the contract.

Proof

In most definitions of refinement, the proof of refinement is stated informally. The contracts enable formal proofs to be realized both *vertically*, i.e., during a refinement step, and *horizontally*, i.e., for a given specification.

Vertically, the use of contracts enables to prove that a lower-level specification is a correct refinement of a higher-level one. The proof of refinement is reduced to the proof of inclusion of the translated high-level contract into a lower-level one.

Horizontally, given a formal specification, a proof is performed, that enables to state that a set of formulae is actually a contract, i.e., it is satisfied by all the models of the specification. The contract ensures that a proof has been performed, and enables the user of the specification (a human being or another system) to know the behaviour which is guaranteed by the system.

Practically, these proofs are realized by model-checking, formal proofs on the basis of the formal specifications (in the case of a sound and complete logic), or tests (for partial proofs).

The use of contracts provides a built-in feature for correctness, and makes our approach similar to that proposed by Meyer [50].

A More General Theory

As observed in Chapter 2, the definitions of refinement can always be reduced to the preservation of properties. Since the theory of refinement based on contracts is founded on the preservation of explicit properties, this theory is, in some aspects, more general than other existing theories of refinement:

- Meta-Refinement.
  The theory of refinement presented in this chapter is a kind of "meta-refinement", since the contract decides upon the refinement performed. Given a formal specifications language, and a high-level specification *Spec*, there are as many possible contracts satisfied by this specification as the number of sets in the power set: $\mathcal{P}(\Phi_{Spec})$. This means that there are as many different definitions of refinements as the number of different sets forming the contracts. In the case of a CO-OPN specification, we can use a contract specifying the bisimulation between the transitions systems. Thus, the refinement leads to the same set of possible lower-level specifications as the one we obtain when we use the refinement defined by the CO-OPN formalism; or we can use a contract specifying only input/output behaviour, and the refinement leads to a set of possible lower-level specifications completely different from those obtained with the bisimulation. Similarly to the implementation, given two contractual specifications with the same specification part, but two different contracts, the set of programs implementing correctly one of the two contractual specification is different from the one implementing the other contractual specification;

- Nature of the Contract.
  Properties of a contract may be of different classes, and it is not necessary that a whole class is part of a contract. In addition, the nature of the contracts can change during the refinement process. For instance, the refinement process may start with a high-level contractual specification whose contract specifies only its functionality, say computing sums. Due to refinement choices or to implementation constraints, non-functional requirements, e.g., dependability constraints or high-parallelisation of the computations, are integrated. Thus the final system has to perform the original functionality, and in addition, it must be able to recover from certain faults or the sums must be computed in parallel as much as possible. The existing definitions of refinement imply that the same class of properties be preserved during the whole refinement process.

- Tuning.
  The use of contracts enables the specifier to adapt the refinement to each system.

Emphasis is put on specific needs and requirements of the system to develop, and not on semantical requirements generally stated by the specifications language.

3.5.7 Disadvantages of the Use of Contracts

The specifier is aware of the semantical requirements of each refinement step. This awareness allows the advantages we have discussed above, however it implies some disadvantages.

More effort has to be produced at each step, since the specifier must build not only the specification, but also the contract, and he must prove that the models of the current specification satisfy the contract. In addition, the specifier must prove at each step that the lower-level contract contains the translated high-level contract.

If the contract stands for a whole class of properties, it may contain an infinite number of formulae. Thus, practically, it may be impossible to write them down, unless the logic used allows to express infinite properties with a finite number of formulae.

Even with the use of an expressive logic, it may happen that the number of formulae of the contract is huge. In this case, a specifier cannot write all the formulae himself. A tool assisting the specifier is necessary to write the formulae and to prove them. The contract becomes huge especially when non-functional properties are part of the contract, e.g., all the traces of the models of the high-level specification must be kept by the models of the lower-level one.

However, these disadvantages are present in other definitions of refinement as well, since the use of contracts enables to simulate existing definitions of refinements. The use of contracts explicitly points out problems (like the proof of refinement when the contract is infinite) that already exist in other definitions of refinement.

Loss of Original Requirements

Refine relations enable to *rename* high-level elements. This feature can be useful in certain cases. However, the possibility of renaming, combined with a small contract, can lead to a semantical change of the original formulae. We consider the following example: a system whose purpose is to make sums. Formulae of the contract are built with the "+" operator, which adds up two integers. During a refinement step, the "+" operator is renamed to the "−" operator. If the "−" operator actually behaves like the subtraction of integers, and if the contract contains no formula of the kind $0 + 1 = 1$, which ensures that the semantics of the "+" operator is preserved, then formulae built with the addition are translated to formulae built with the subtraction.

This effect can be ignored if the important point is the ability to make operations on integers (it is not important whether the operation is an addition or a subtraction). On the contrary, if the operation has to be the addition, then the specifier must be very careful, and must put into the contract all formulae necessary to ensure that, even though a renaming is performed, the semantics of the addition is preserved.

Chapter 4

CO-OPN/2

Chapter 3 defines a theory of refinement and implementation based on contracts, which advocates the joint use of a model-oriented formal specifications language, and a logical language. The following chapters carry out this general theory to an object-oriented formal specifications language, called CO-OPN/2. The current chapter is dedicated to the description of the syntax and the semantics of CO-OPN/2 specifications.

CO-OPN/2 is an object-oriented formal specifications language based on partial order-sorted algebraic specifications [61] and Petri nets which are combined in a way that is similar to algebraic nets [56]. Algebraic specifications are used to describe the data structures and the functional aspects of a system, while Petri nets allow to model the system's concurrent features. To compensate for algebraic Petri nets' lack of structuring capabilities, CO-OPN/2 provides a structuring mechanism based on a synchronous interaction between algebraic nets, as well as notions specific to object-orientation such as the notions of class, inheritance, and sub-typing. A system is considered as being a collection of independent objects (algebraic nets) which interact and collaborate together in order to accomplish the various tasks of the system. The formal semantics of a CO-OPN/2 specification is given in terms of a concurrent transition system expressing all the possible evolutions of objects' states.

CO-OPN/2 is the object-oriented version of CO-OPN [21]. CO-OPN provides the same mechanism of synchronous interaction between algebraic nets, but is simply object-based (no dynamic creation of instances, no inheritance, no sub-typing). A definition of refinement for CO-OPN has been defined, which is based on strong bisimulation between the states of transition systems. A series of tools is available for CO-OPN [15]; it includes a syntax checker, a simulator, a property verifier based on temporal logic, a graphical editor, and a transformation tool supporting the derivation of specifications.

First the current chapter presents the syntax of CO-OPN/2 specifications and then their semantics.

The definitions, theorem, propositions, examples, as well as explanations of this chapter are all taken from Biberstein's Ph.D. thesis [14].

4.1 Syntax

The CO-OPN/2 formalism introduces the notion of modules. Two kinds of modules are provided: *ADT modules* and *Class modules*. The ADT modules are used for the specification of the abstract data types involved in a CO-OPN/2 specification while the Class modules correspond to the description of the objects obtained by instantiation. Both these kinds of modules are composed of a part which groups the elements accessible by other modules, called the *ADT module signature* or the *Class module interface*, according to the type of module. The other elements, which compose the module, describe the properties of the module; they are grouped in a *body* part, and are not accessible by other modules.

Throughout this chapter, as well as in the following chapters, we use the notation below:

Notation 4.1.1 *Universe of all names.*
We consider a given universe $\mathcal{U}$ *which includes the disjoint sets:* $\mathsf{S}, \mathsf{F}, \mathsf{M}, \mathsf{P}, \mathsf{V}, \mathsf{O}$. *These sets correspond, respectively, to the sets of all sort, operation, method, place, variable and static object names.*

The set S *is divided into two disjoint sets* $\mathsf{S^A}$ *and* $\mathsf{S^C}$, $\mathsf{S} = \mathsf{S^A} \cup \mathsf{S^C}$ *with* $\mathsf{S^A} \cap \mathsf{S^C} = \varnothing$. *The former is dedicated to all the usual sort names involved in the algebraic description part, whereas the latter consists in all the type names of the classes.*

First we present ADT module signatures and Class module interfaces; and describe how global signatures and global interfaces are derived from a set of ADT module signatures and Class module interfaces. Second, we define ADT modules and Class modules. Then, we present CO-OPN/2 specifications.

4.1.1 ADT Module Signature

The elements of an ADT module that can be used from the outside are defined in the ADT module signature. It 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 $\mathsf{S^A}$, and not only over the set of sorts S^{A} defined in the module itself.

Definition 4.1.2 *ADT module signature.*
An ADT module signature *(ADT signature for short) (over* S *and* F*) is a triple* $\Sigma^{\mathsf{A}} = \langle S^{\mathsf{A}}, \leq^{\mathsf{A}}, F \rangle$, *where*

- S^{A} *is a set of sort names of* $\mathsf{S^A}$;

- $\leq^{\mathsf{A}} \subseteq (S^{\mathsf{A}} \times \mathbf{S}^{\mathbf{A}}) \cup (\mathbf{S}^{\mathbf{A}} \times S^{\mathsf{A}})$ *is a partial order (partial sub-sort relation);*
- $F = (F_{w,s})_{w \in \mathbf{S}^*, s \in \mathbf{S}}$ *is a* $(\mathbf{S}^* \times \mathbf{S})$*-sorted set*[1] *of function names of* $\mathbf{F}$.

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

We often denote a function name $f \in F_{s_1 \cdots s_n, s}$ by $f : s_1, \ldots, s_n \to s$ or by $f_{s_1 \ldots s_n, s}$, and a constant $f \in F_{\epsilon,s}$ by $f : \to s$ or by f_s (ϵ represents the empty string). The index $(s_1 \cdots s_n, s)$ is called the *arity* of the members of $F_{s_1 \cdots s_n, s}$.

The profile of the operations is built over $\mathbf{S}$, therefore some elements with such profiles can imply sorts of $\mathbf{S}^{\mathbf{C}}$. Thus, ADT modules can describe data structures containing object identifiers, for example: stack or arrays of object identifiers.

Remark 4.1.3 *When a signature only uses elements locally defined we say that the signature is* complete.

CO-OPN/2 provides abstract definitions as well as textual representations. Figure 4.1 gives the textual representation of an ADT module defining three sorts: `chocolate`, `praline` and `truffle`. Sorts `praline` and `truffle` are both sub-sorts `chocolate`. This ADT defines only two generators `P` and `T` producing pralines and truffles respectively.

```
Adt Chocolate;
Interface
  Sorts chocolate, praline, truffle;
  Subsort
    praline < chocolate;
    truffle  < chocolate;
  Generators
    P : praline;
    T : truffle;
End Chocolate;
```

Figure 4.1: CO-OPN/2 `Chocolate` ADT Module

Example 4.1.4 *ADT Module Signature.*
The ADT module signature corresponding to Figure 4.1 is given by:

$$\Sigma^{\mathsf{A}}_{\text{Chocolate}} = \Big\langle \{\,\text{chocolate}, \text{praline}, \text{truffle}\}, \{(\text{praline}, \text{chocolate}),\ (\text{truffle}, \text{chocolate})\}, \{\,\text{P}_{\text{praline}}, \text{T}_{\text{truffle}}\} \Big\rangle.$$

[1] a S-sorted set A is a family of sets indexed by S, we write $A = (A_s)_{s \in S}$.

4.1.2 Class Module Interface

A Class module describes a collection of objects with the same structure by means of an encapsulated algebraic net. 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: (1) the type of the class; (2) a sub-type relation with other classes; (3) the set of methods that corresponds to the services provided by the class, methods being particular transitions of the net; (4) and the set of static objects provided by the Class, static objects are always available independently of the number of instances of the Class that have been created.

Definition 4.1.5 *Class module interface.*
A class module interface *(class interface for short) (over* S*,* M*, and* O*) is a 4-tuple*[2] $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, O\rangle$*, where:*

- $c \in \mathsf{S}^{\mathsf{C}}$ *is the type*[3] *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*;*
- $O = (O_c)_{c \in \mathsf{S}^{\mathsf{C}}}$ *is a finite* S^{C}*-sorted set of static object names of* O*.*

A method is not a function, but a parameterised 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 $m_c : s_1, \ldots, s_n$ or $m_{c,s_1,\ldots,s_n}$, while a method without any argument $m \in M_{c,\epsilon}$ is written m_c (ϵ denotes the empty string). Set M contains also non-default generators of instances of the class.

From a set of ADT signatures $\mathbf{\Sigma} = \{\Sigma_i^{\mathsf{A}} \mid 1 \leq i \leq n\}$ and a set of class interfaces $\mathbf{\Omega} = \{\Omega_j^{\mathsf{C}} \mid 1 \leq j \leq m\}$ such that $\Sigma_i^{\mathsf{A}} = \langle S_i^{\mathsf{A}}, \leq_i^{\mathsf{A}}, F_i\rangle$ for $1 \leq i \leq n$ and $\Omega_j^{\mathsf{C}} = \langle \{c_j\}, \leq_j^{\mathsf{C}}, M_j, O_j\rangle$ for $1 \leq j \leq n$, we construct a *global sub-sort/sub-type relation* noted $\leq_{\mathbf{\Sigma},\mathbf{\Omega}}$ which is the reflexive and transitive closure of the union of the partial sub-sort and sub-type relations of the elements of $\mathbf{\Sigma}$ and $\mathbf{\Omega}$:

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

Since a class interface includes two elements closely related to the algebraic part, namely the type of the class and the sub-type relation, a class interface $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, O\rangle$ induces an ADT signature that contains the operations necessary for the management of the objects identifiers, as well as one constant for each static object.

[2]here the C superscript stresses the belonging to the class (algebraic net) dimension.

[3]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.

Definition 4.1.6 *ADT signature induced by Class interface.*
Let $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, O \rangle$ *be a Class module interface, the ADT signature induced by* Ω^{C}*, noted* $\Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}}$*, is such that* $\Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}} = \langle \{c\}, \leq^{\mathsf{C}}, F_{\Omega^{\mathsf{C}}} \rangle$*, and:*

$$\begin{aligned} F_{\Omega^{\mathsf{C}}} = \; & \{o_{c'} : \rightarrow c' \mid o : c' \in O\} \cup \\ & \{\mathrm{init}_c : \rightarrow c, \mathrm{new}_c : c \rightarrow c\} \cup \\ & \{\mathrm{sub}_{c,c'} : c \rightarrow c', \mathrm{super}_{c,c''} : c \rightarrow c'' \mid c' \leq_{\mathbf{\Sigma},\mathbf{\Omega}} c, c \leq_{\mathbf{\Sigma},\mathbf{\Omega}} c''\}. \end{aligned}$$

Function $o_{c'}$ provides object identifiers of static objects. Function init_c provides the object identifier of the first object of type c that is created either statically or dynamically. Function new_c generates a new (the next) object identifier from a given object identifier. Functions $\mathrm{sub}_{c,c'}$ and $\mathrm{super}_{c,c''}$ map object identifiers of type c with object identifiers whose type is a sub-type or a super-type of c respectively.

Figure 4.2 gives the textual representation of the Class module interface of a Class module called `Packaging`. This Class module defines chocolate boxes of type `packaging`. Such boxes offer two services: `fill` for putting a chocolate inside a box, and `full-praline` which is used to know when the box is full of chocolates. A non-default generator of instances is provided `create-packaging`. Class module `Packaging` defines no sub-type and no static object.

```
Class Packaging;
Interface
  Use Chocolate;
  Type packaging;
  Methods
    fill _ : chocolate;
    full-praline;
  Creation
    Create-packaging;
Body
   ...
End Packaging;
```

Figure 4.2: CO-OPN/2 `Packaging` Class Module Interface

Example 4.1.7 *Class Module Interface.*
The Class module interface of Class module ***Packaging*** *given by Figure 4.2 is the following:*

$$\Omega^{\mathsf{C}}_{\text{Packaging}} = \Big\langle \{\,\text{packaging}\}, \varnothing,\ \{\text{fill}_{\text{packaging,chocolate}}, \text{full-praline}_{\text{packaging}}\}, \varnothing \Big\rangle.$$

The ADT signature induced by this Class interface is given by:

$$\Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}_{\text{Packaging}}} = \langle \{\text{packaging}\}, \varnothing, F_{\Omega^{\mathsf{C}}_{\text{Packaging}}} \rangle,$$

and:

$$F_{\Omega^{\mathsf{C}}_{\text{Packaging}}} = \{\text{init}_{\textbf{packaging}} : \rightarrow \text{packaging}, \text{new}_{\textbf{packaging}} : \text{packaging} \rightarrow \text{packaging}\}.$$

4.1.3 Global Signature and Global Interface

From a set of ADT module signatures and a set of a Class module interfaces, it is possible to build a *global signature* and a *global interface*. Intuitively, a global signature groups the sorts and types, the sub-sort and sub-type relations, as well as the operations of ADT signatures and Class interfaces. As for a global interface, it groups the types, the sub-type relations, the methods, and the static objects of a set of class interfaces.

Definition 4.1.8 *Global signature and global interface.*
Let $\boldsymbol{\Sigma} = (\Sigma^{\mathsf{A}}_i)_{1\leq i\leq n}$ *be a set of ADT signatures and* $\boldsymbol{\Omega} = (\Omega^{\mathsf{C}}_j)_{1\leq j\leq m}$ *be a set of class interface such that* $\Sigma^{\mathsf{A}}_i = \langle S^A_i, \leq^{\mathsf{A}}_i, F_i\rangle$ *and* $\Omega^{\mathsf{C}}_j = \langle\{c_j\}, \leq^{\mathsf{C}}_j, M_j, O_j\rangle$.
The global signature *over* $\boldsymbol{\Sigma}$ *and* $\boldsymbol{\Omega}$ *is:*

$$\Sigma_{\boldsymbol{\Sigma},\boldsymbol{\Omega}} = \left\langle \bigcup_{1\leq i\leq n} S^A_i \cup \bigcup_{1\leq j\leq m} \{c_j\},\ \leq_{\boldsymbol{\Sigma},\boldsymbol{\Omega}},\ \bigcup_{1\leq i\leq n} F_i \cup \bigcup_{1\leq j\leq m} F_{\Omega^{\mathsf{C}}_j} \right\rangle.$$

The global interface *over* $\boldsymbol{\Omega}$ *is:*

$$\Omega_{\boldsymbol{\Omega}} = \left\langle \bigcup_{1\leq j\leq m} \{c_j\},\ (\bigcup_{1\leq j\leq m} \leq^{\mathsf{C}}_j)^*,\ \bigcup_{1\leq j\leq m} M_j,\ \bigcup_{1\leq j\leq m} O_j \right\rangle.$$

In order to ensure that the global signature is an order-sorted signature, some conditions are required on signatures such as monotonicity, regularity and coherence. The following definitions introduce these notions.

Definition 4.1.9 *Many-sorted and order-sorted signature.*
A many-sorted signature *(upon* S *and* F*)* $\Sigma = \langle S, F\rangle$ *consists of a set of sorts* $S \subseteq \mathsf{S}$ *and a* $S^* \times S$*-sorted family of operation or function names* $F = (F_{w,s})_{w\in S^*, s\in S}$ *with* $F \subseteq \mathsf{F}$. *An* order-sorted signature *is a triple* $\langle S, \leq, F\rangle$ *such that* $\langle S, F\rangle$ *is a many-sorted signature,* $\langle S, \leq\rangle$ *is a poset*[4], *and the operation names satisfy the following monotonicity condition,*

$$\textit{if } f \in F_{w_1,s_1} \cap F_{w_2,s_2} \textit{ and } w_1 \leq w_2 \textit{ then } s_1 \leq s_2.$$

[4]the pair $(S, \leq)$ is a *partially ordered set*, or *poset* for short, if $\leq\, \subseteq S \times S$ is a partial order relation (reflexive, transitive and antisymmetric).

Pre-regularity is equivalent to the existence of a least sort for every term. Regularity is a stronger condition which allows both ad-hoc polymorphism and sub-sort polymorphism. Regularity implies pre-regularity. Coherence is needed to force an equation to be valid in all isomorphic models.

Definition 4.1.10 *Pre-regular, regular, and coherent signature.*
An order-sorted signature $\Sigma = \langle S, \leq, F\rangle$ *is* pre-regular *iff for any* $f \in F_{w_1,s_1}$ *and any* $w_0 \leq w_1$ *in* S^**, there is a least sort* $s \in S$ *such that* $f \in F_{w,s}$ *and* $w_0 \leq w$ *for some* $w \in S^*$.

Σ *is* regular *iff there is a least* $(w, s) \in S^* \times S$ *such that* $w_0 \leq w$ *and* $f \in F_{w,s}$. Σ *is* coherent *iff it is regular and each sort* s *has a maximum in* S.

Lemma 4.1.1 below provides a combinatorial condition that is equivalent to regularity.

Lemma 4.1.1 *Let* $\Sigma = \langle S, \leq, F\rangle$ *be an order-sorted signature over a finite set of sorts.* Σ *is regular iff whenever* $f \in F_{w_1,s_1} \cap F_{w_2,s_2}$ *and there is some* $w_0 \leq w_1, w_2$*, then there is* (w, s) *such that* $w \leq w_1, w_2$ *and* $s \leq s_1, s_2$ *and* $f \in F_{w,s}$ *and* $w_0 \leq w$.

Proposition 4.1.1 ensures that the global signature is an order-sorted signature.

Proposition 4.1.1 *Let* $\mathbf{\Sigma}$ *be a set of ADT signatures and* $\mathbf{\Omega}$ *be a set of class interfaces. If the global signature* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$ *is complete and satisfies the monotonicity condition, then* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$ *is an order-sorted signature.*

In a similar way, a set of class interfaces must satisfy the contra-variance condition that guarantees, at the syntactic level, the substitutability principle of an object of type c' by any object of type c when c is a sub-type of c'.

Definition 4.1.11 *Contra-variance condition.*
A set of class interfaces $\mathbf{\Omega}$ *satisfies the* contra-variance *condition iff for any class interface* $\langle\{c\}, \leq^{\mathsf{C}}, M, O\rangle$ *and* $\langle\{c'\}, \leq^{\mathsf{C}'}, M', O'\rangle$ *in* $\mathbf{\Omega}$ *the following property holds. If* $c \leq_{\varnothing,\mathbf{\Omega}} c'$ *then for each method* $m_{c'} : s'_1, \ldots, s'_n$ *in* M' *there exists a method* $m_c : s_1, \ldots, s_n$ *in* M *such that* $s'_i \leq s_i$ $(1 \leq i \leq n)$.

Given a signature and a set of variables, we can construct the set of terms in the following way:

Definition 4.1.12 *Set of all terms.*
Let $\Sigma = \langle S, \leq, F\rangle$ *be a signature and* X *be a* S*-sorted variable subset of* V. *The set of all terms over* Σ *and* X *with sort* $s \in S$*, noted* $(T_{\Sigma,X})_s$*, is the least set with the following properties:*

i) $x \in (T_{\Sigma,X})_s$ *for all* $x \in X_{s'}, s' \leq s$;

ii) $f \in (T_{\Sigma,X})_s$ *for all* $f : \rightarrow s' \in F$, *such that* $s' \leq s$;

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

We define $T_{\Sigma,X} \stackrel{def}{=} ((T_{\Sigma,X})_s)_{s \in S}$ *as the* S*-sorted set of all terms over* Σ *and* X, *and* $T_\Sigma \stackrel{def}{=} T_{\Sigma,\emptyset}$ *as the set of all ground terms.*

Remark 4.1.13 *If type* s' *is a sub-type of* s, *i.e.,* $s' \leq s$, *then every term of type* s' *is also a term of type* s.

When Σ *is a global signature, and* $S = S^A \cup S^C$, *with* S^A *the set of ADT sorts and* S^C *the set of Class types, then terms of sort* $s \in S^A$ *stand for data values, while terms of type* $c \in S^C$ *are object identifiers.*

4.1.4 ADT Modules

An ADT module consists of a visible part, which is the ADT signature; and a hidden part, which is given by a set of variables, and a set of formulae also called axioms.

Definition 4.1.14 *Equation, atomic formula, formula, axiom.*
Let $\Sigma = \langle S, \leq, F \rangle$ *be a regular signature and* X *be a* S*-disjointly-sorted set of variables.*

1. *A* Σ-equation *is a pair* (t, t') *of terms in* $T_{\Sigma,X}$ *such that the sort of* t *and that of* t' *are related by the reflexive and transitive closure of* $\leq$. *We denote a* Σ*-equation* (t, t') *by* $t = t'$.

2. *An* atomic formula *is either a* Σ*-equation or a* definedness formula *of a term* t *in* $T_{\Sigma,X}$ *noted* $\boldsymbol{D}\, t$.

3. *A* formula *(or* axiom*) is either an atomic formula or a family of atomic formulae* $\{\phi_i, \phi \mid 1 \leq i \leq n\}$. *We note such a family by* $\phi_1 \wedge \cdots \wedge \phi_n \Rightarrow \phi$.

Definition 4.1.15 *ADT module.*
Let $\mathbf{\Sigma}$ *be a set of ADT signatures and* $\mathbf{\Omega}$ *be a set of class interfaces such that the global signature* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle S, \leq, F \rangle$ *is complete. An* ADT module *is a triple* $Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle \Sigma^{\mathsf{A}}, X, \Phi \rangle$, *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 formulae (axioms) over* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$ *and* X.

Remark 4.1.16 *In the context of structured specifications, an ADT module may obviously use elements not locally defined, i.e., defined in other modules.*

Figure 4.3 provides a more complex ADT module. If defines a FIFO (first in, first out) structure, able to store boxes of type `packaging` defined by Class module `Packaging` (see Figure 4.2). It defines two sorts: `fifo-packaging` and `ne-fifo-packaging` (for non-empty FIFOs). It provides two generators: `[]` for creating empty FIFOs; and `insert` for adding a box of type `packaging` at the end of a FIFO, the FIFO obtained after this operation is a non-empty one. The operations defined by this ADT module are: `first`, which returns the object identifier of the box at the head of the FIFO; `extract`, which removes this object identifier; and `size`, which returns the size of the FIFO.

The `Axioms` field gives formulae Φ; they formally defied the generators and the operations. The set of variables used for establishing the formulae is $X = \{\text{box}_{\text{packaging}}, \text{f}_{\text{ne-fifo-packaing}}\}$.

```
Adt FifoPackaging;
Interface
  Use Naturals, Packaging;
  Sorts ne-fifo-packaging, fifo-packaging;
  Subsort ne-fifo-packaging < fifo-packaging;
  Generators
    [] : -> fifo-packaging;
    insert _ _ : packaging fifo-packaging ->
                 ne-fifo-packaging;
  Operations
    first   _ : ne-fifo-packaging -> packaging;
    extract _ : ne-fifo-packaging -> fifo-packaging;
    size    _ : ne-fifo-packaging -> natural;
Body
  Axioms
    first (insert box [])  = box;
    first (insert box f)   = first f;
    extract (insert box []) = [];
    extract (insert box f)  =
                insert box (extract f);
    size [] = 0;
    size (insert box f) = 1 + (size f);

    Where
      box : packaging;
      f   : ne-fifo-packaging;
End FifoPackaging;
```

Figure 4.3: CO-OPN/2 `FifoPackaging` ADT Module

4.1.5 Class Module

The purpose of a Class module is to describe a collection of objects having 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 is made of a visible part, i.e., a Class module interface; and a body part, which actually defines the algebraic net. It consists of: a set of places, some variables, the initial values of the places, and a set of behavioural formulae which describe the behaviour of instances of the class, when events occur.

The CO-OPN/2 formalism provides two different categories of events: the *invisible* events, and the *observable* events. Both of them can involve an optional *synchronisation expression*. The invisible events describe the spontaneous reactions of an object to some stimuli. They correspond to the *internal transitions* which we will denote by τ. The observable events correspond to the *methods*, defined in the Class module interface, and which are then accessible from the outside. A synchronisation expression offers an object the means of choosing how to be synchronised with other partners (even itself). In the textual representation of a CO-OPN/2 specification, the keyword `with` introduces the synchronisation expression. Three synchronisation operators are provided: '//' for simultaneity, '..' for sequence, and '$\oplus$' for alternative. In order to select a particular method of a given object, the usual dot notation has been adopted.

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

Definition 4.1.17 *Set of all events.*
Let $(S, \leq)$ be a poset, where $S = S^A \cup S^C$ is a set of sorts such that $S^A \in \mathbf{S^A}$ and $S^C \in \mathbf{S^C}$. Let us consider $A = (A_s)_{s \in S}$, a set of terms, $M = (M_{s,w})_{s \in S^C, w \in S^*}$ a set of method names, $O = (O_s)_{s \in S^C}$ a set of terms for object identifiers, and a set of types of classes $C \subseteq S^C$. The set of all events *(over A, M, O, C), noted $\mathbf{E}_{A,M,O,C}$, is made of events Event, built according to the following syntax:*

$$\begin{array}{rcl} \mathit{Event} & \rightarrow & \mathit{Inv} \mid \mathit{Inv}\ \text{with}\ \mathit{Sync} \mid \mathit{Obs} \mid \mathit{Obs}\ \text{with}\ \mathit{Sync} \\ \mathit{Inv} & \rightarrow & \mathit{self}.\tau \\ \mathit{Obs} & \rightarrow & \mathit{self}.m(a_1, \ldots, a_n) \mid \mathit{Obs}\ //\ \mathit{Obs} \mid \mathit{Obs}\ ..\ \mathit{Obs} \mid \mathit{Obs} \oplus \mathit{Obs} \\ \mathit{Sync} & \rightarrow & o.m(a_1, \ldots, a_n) \mid o.\text{create} \mid o.\text{destroy} \mid \\ & & \mathit{Sync}\ //\ \mathit{Sync} \mid \mathit{Sync}\ ..\ \mathit{Sync} \mid \mathit{Sync} \oplus \mathit{Sync} \end{array}$$

where $s \in S^C$, $s_i, s'_i \in S$ $(1 \leq i \leq n)$, $a_1, \ldots, a_n \in A_{s'_1} \times \cdots \times A_{s'_n}$, $m \in M_{s,s_1 \cdots s_n}$, $o \in O_s$, $s \in C$, and $\mathit{self} \in O_s$ and such that (s'_i, s_i) $(1 \leq i \leq n)$ belongs to the transitive and reflexive closure of $\leq$.

Since behavioural formulae handle terms of sort multi-set, we first define the multi-set extension of signatures. It consists of extending the signature: (1) by adding a sort noted

$[s]$, for every sort s of the signature, which stands for the sort multi-set of s; (2) by extending the sub-sort relation to the multi-sets; (3) by adding three functions for every $[s]$ that respectively generate: an empty multi-set, create a multi-set with a single element of sort s, and make the union of two multi-sets.

Definition 4.1.18 *Syntactic multi-set extension of signatures.*
Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature. The syntactic multi-set extension of Σ is noted $[\Sigma]$ and defined by:

$$[\Sigma] = \langle S \cup \bigcup_{s\in S} \{[s]\}, \leq \cup \bigcup_{\substack{s, s' \in S \\ s \leq s'}} \{([s],[s'])\}, F \cup \bigcup_{s\in S} \left\{ \begin{array}{c} \emptyset_s : \to [s], \\ [\_]_s : s \to [s], \\ +_s : [s],[s] \to [s] \end{array} \right\} \rangle.$$

Behavioural formulae are used to describe the properties of observable and invisible events (respectively, methods and internal transitions) of a net. A behavioural formula consists of an event, a condition expressed by means of a set of equations over algebraic values, and the usual pre/post-conditions of the event. Both pre/post-conditions are sets of terms (of sort multi-set) indexed by the places of the net. An event can occur (or using the Petri nets jargon, the method or the internal transition can be fired) if and only if the condition on the algebraic values is satisfied and enough resources can be consumed/produced from/in the places of the module.

Definition 4.1.19 *Behavioural formula.*
Let $\Sigma = \langle S, \leq, F\rangle$ be an order-sorted signature such that $S = S^A \cup S^C$ ($S^A \in \mathbf{S^A}$ and $S^C \in \mathbf{S^C}$). For a given $(S^C \times S^*)$-sorted set of methods M, a S-disjointly-sorted set of places P, a set of types $C \subseteq S^C$, and a S-disjointly-sorted set of variables X. A behavioural formula *is a 4-tuple $\langle Event, Cond, Pre, Post\rangle$, where:*

- *$Event \in \mathbf{E}_{(T_{\Sigma,X}),M,(T_{\Sigma,X})_s,C}$ such that $s \in S^C$;*
- *$Cond$ is a set of equations*[5] *over Σ and X;*
- *$Pre = (Pre_p)_{p\in P}$ is a family of terms over $[\Sigma], X$ indexed by P, such that*

$$(\forall s \in S)\ (\forall p \in P_s)\ (Pre_p \in (T_{[\Sigma],X})_{[s]});$$

- *$Post = (Post_p)_{p\in P}$ is a family of terms over $[\Sigma], X$ indexed by P, such that*

$$(\forall s \in S)\ (\forall p \in P_s)\ (Post_p \in (T_{[\Sigma],X})_{[s]}).$$

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

$$Event :: Cond \Rightarrow Pre \to Post.$$

[5]see Definition 4.1.14

Finally, a Class module consists of: a class interface, a set of places, which corresponds to the state of the class instances, some variables, the initial values of the places (also called the *initial marking* of the module), and a set of behavioural formulae which describe the properties of the methods and of the internal transitions.

Definition 4.1.20 *Class module.*
Let $\mathbf{\Sigma}$ *be a set of ADT signatures,* $\mathbf{\Omega}$ *be a set of class interfaces such that the global signature* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle S, \leq, F\rangle$ *is complete. A* Class module *is a 5-tuple* $Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}} = \langle \Omega^{\mathsf{C}}, P, I, X, \Psi\rangle$, *where:*

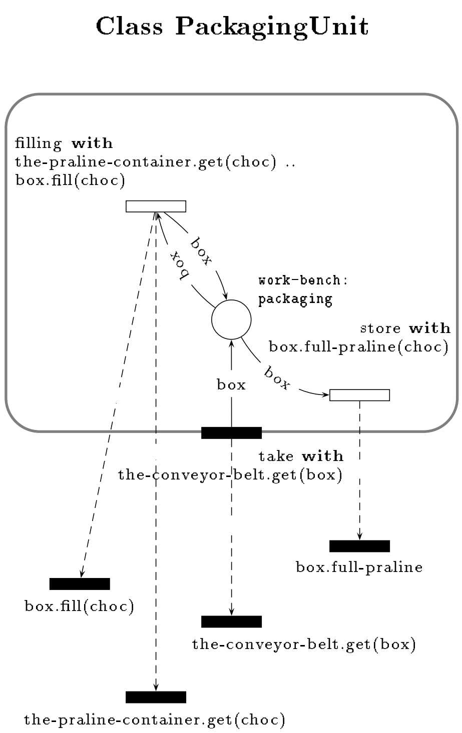
- $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M\rangle$ *is a class interface;*
- $P = (P_s)_{s\in S}$ *is a finite* S*-disjointly-sorted set of place names of* P*;*
- $I = (I_p)_{p\in P}$ *is an initial marking, a family of terms indexed by* P *such that*
$$(\forall s \in S)\ (\forall p \in P_s)\ (I_p \in (T_{[\mathbf{\Sigma}],X})_{[s]});$$
- $X = (X_s)_{s\in S}$ *is a* S*-disjointly-sorted set of variable of* V*;*
- Ψ *is a set of behavioural formulae over the global signature* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$*, a set of methods composed of* M *and all the methods of* $\mathbf{\Omega}$*, the set of places* P*, the type of the class* $\{c\}$*, and* X*.*

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 $\mathsf{S^C}$.

CO-OPN/2 provides a textual representation of ADT modules and Class modules. In addition, it provides a graphical representation of Class modules. Figure 4.4 defines Class module `PackagingUnit`. Left part of the figure shows the graphical representation, while right part gives the textual representation.

Class module `PackagingUnit` defines a unique method `take` which removes a box of type `packaging` from a static object called `the-conveyor-belt` provided by Class module `ConveyorBelt`, and stores it into place `work-bench`. A synchronous request introduced with keyword `with` is used for actually obtaining boxes from `the-conveyor-belt`. Class module `ConveyorBelt` simply stores `packaging` boxes using a `fifo-packaging` structure. In addition to method `take`, Class module `PackagingUnit` defines two transitions `filling` and `store`. Transition `filling` takes chocolates from a static object called `the-praline-container`, defined in Class module `PralineContainer`; and sequentially (using operator "..") inserts this chocolate into one of the available boxes, currently stored into place `work-bench`. Transition `store` removes a box from place `work-bench` once it has been completely filled with chocolates.

Appendix A gives the CO-OPN/2 specification of Class modules `PralineContainer` and `ConveyorBelt`.




```
Class PackagingUnit;
Interface
  Type packaging-unit;
  Method Take;
Body
  Use Chocolate, ConveyorBelt,
      Packaging, PralineContainer;
  Transitions
    filling, store;
  Place
    work-bench _ : packaging;
  Axioms
    Take With the-conveyor-belt.get box ::
      -> work-bench box;
    filling With
      the-praline-container. get choc ..
      box.fill choc ::
      work-bench box -> work-bench box;
    store With box.full-praline choc ::
      work-bench box -> ;
    Where
      box: packaging;
      choc: chocolate;
End PackagingUnit;
```

Figure 4.4: CO-OPN/2 PackagingUnit Class Module

4.1.6 CO-OPN/2 Specification

Finally, a CO-OPN/2 specification is a collection of ADT and Class modules.

Definition 4.1.21 *CO-OPN/2 specification.*
Let $\mathbf{\Sigma}$ *be a set of ADT signatures,* $\mathbf{\Omega}$ *be a set of class interfaces such that* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$ *is complete and coherent, and such that* $\Omega_{\mathbf{\Omega}}$ *satisfies the contra-variance condition. A* CO-OPN/2 specification *consists of a set of ADT and class modules:*

$$Spec_{\mathbf{\Sigma},\mathbf{\Omega}} = \{(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_i \mid 1 \le i \le n\} \cup \{(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_j \mid 1 \le j \le m\}.$$

We denote a CO-OPN/2 specification $Spec_{\mathbf{\Sigma},\mathbf{\Omega}}$ *by Spec and the global sub-sort/sub-type relation* $\le_{\mathbf{\Sigma},\mathbf{\Omega}}$ *by* $\le$ *when* $\mathbf{\Sigma}$ *and* $\mathbf{\Omega}$ *are, respectively, included in the global signature and in the global interface of the specification. In this case, the specification is considered* complete.

Two dependency graphs can be constructed from a CO-OPN/2 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 client-ship relationship between the class modules. Both these graphs are composed of the specification *Spec* and a binary relation over *Spec* noted D^{A}_{Spec} for the algebraic dependency graph, and D^{C}_{Spec} for the client-ship dependency graph. The relation D^{A}_{Spec} is constructed as follows: for any module Md, Md' of *Spec* ($Md \neq Md'$), (Md, Md') is in D^{A}_{Spec} if and only if the ADT module Md or the ADT signature induced by the class module Md uses some elements defined in the ADT signature of Md' or in the ADT signature induced by the class module Md'. As for the relation D^{C}_{Spec}, it is constructed as follows: for any class module Md, Md' ($Md \neq Md'$), (Md, Md') is in D^{C}_{Spec} if and only if there is a synchronisation expression of a behavioural formula of Md which involves a method of Md'.

Thus, a *well-formed* CO-OPN/2 specification is 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 the class module dimension of our formalism, as will be shown in the next section.

Definition 4.1.22 *Well-formed CO-OPN/2 specification.*
A complete CO-OPN/2 specification Spec is well-formed *iff:*

i) the algebraic dependency graph $\langle Spec, D^{\mathsf{A}}_{Spec}\rangle$ *has no cycle;*

ii) the client-ship dependency graph $\langle Spec, D^{\mathsf{C}}_{Spec}\rangle$ *has no cycle.*

In the rest of the current chapter, and in the following chapters, we use the notations below:

Notation 4.1.23 *Let Spec be a well-formed CO-OPN/2 specification, and* $\Sigma_{\mathbf{\Sigma},\mathbf{\Omega}}$ *be the global signature of Spec, and* $\Omega_{\mathbf{\Omega}}$ *be the global interface of Spec, obtained by Definition 4.1.8. We denote:*

$$S^A = \bigcup_{1 \le i \le n} S_i^A \qquad S^C = \bigcup_{1 \le j \le m} \{c_j\} \qquad S = S^A \cup S^C$$
$$F^A = \bigcup_{1 \le i \le n} F_i \qquad F^C = \bigcup_{1 \le j \le m} F_{\Omega_j^C} \qquad F = F^A \cup F^C$$
$$M = \bigcup_{1 \le j \le m} M_j \qquad O = \bigcup_{1 \le j \le m} O_j.$$

Example 4.1.24 *The following CO-OPN/2 specfication is a complete CO-OPN/2 specification with Class module* PackagingUnit *as the root of the two dependencies graphs:*

$$\begin{aligned} Spec = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Chocolate}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Capacity}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, \\ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Packaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ConveyorBelt}}, \\ & (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{PralineContainer}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{PackagingUnit}}\}. \end{aligned}$$

ADT module `Capacity` is used by ADT module `Packaging` and `PralineContainer`. It uses ADT module `Naturals`, which uses ADT module `Booleans`.

4.2 Semantics

This section presents the semantic aspects of the CO-OPN/2 formalism which are based on two notions, the order-sorted algebras, and the transition systems.

First of all, we concentrate on order-sorted algebras as models of a CO-OPN/2 specification, and we introduce an essential element of the CO-OPN/2 formalism, namely the order-sorted algebra of object identifiers, which is organised in a very specific way. Second, the management of object identifiers is presented, as well as the definition of state space. Afterwards we present how the notion of transition system is used in order to describe a system composed of objects dynamically created. Then, we provide all the inference rules which allow us to construct the transition system of a CO-OPN/2 specification. Such a transition system is considered as the semantics of the specification.

4.2.1 Algebraic Models of a CO-OPN/2 Specification

Here, we focus on the semantics of the algebraic dimension of a CO-OPN/2 specification.

Definition 4.1.6 presents the ADT signature induced by each Class module interface of the specification. Remember that such an ADT signature is composed of a type, of a sub-type 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 signature and of the formulae which determine the intended semantics of the operations.

The ADT signature mentioned above includes, for syntactic consistency, a constant for each static object defined in the class interface. At the semantics level, static objects are created at the very beginning of the transition system, and the role of those constants is just to abbreviate the object identifiers of the class instances statically created. Clearly, these abbreviations are not essential. Thus, without loss of generality and for the sake of simplicity, those constants are omitted in the following definition.

Definition 4.2.1 *ADT module induced by a class module.*
Let Spec be a well-formed CO-OPN/2 specification and $\leq$ *be its global sub-sort/sub-type relation. Let* $Md^{\mathsf{C}} = \langle \Omega^{\mathsf{C}}, P, I, V, \Psi \rangle$ *in which* $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, O \rangle$ *be a class module of Spec. The ADT module induced by* Md^{C} *is noted* $Md^{\mathsf{A}}_{\Omega^{\mathsf{C}}} = \langle \Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}}, V_{\Omega^{\mathsf{C}}}, \Phi_{\Omega^{\mathsf{C}}} \rangle$ *in which* $\Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}} = \langle \{c\}, \leq^{\mathsf{C}}, F_{\Omega^{\mathsf{C}}} \rangle$*, and where:*

- $F_{\Omega^{\mathsf{C}}} = \{\mathrm{init}_c : \rightarrow c,\ \mathrm{new}_c : c \rightarrow c\} \cup$
  $\{\mathrm{sub}_{c,c'} : c \rightarrow c',\ \mathrm{super}_{c,c''} : c \rightarrow c'' \mid c' \leq c, c \leq c''\}.$
- $V_{\Omega^{\mathsf{C}}} = \{o_c : c,\ o_{c'} : c' \mid c' \leq c\}$;
- $\Phi_{\Omega^{\mathsf{C}}} = \{\mathrm{sub}_{c,c'}\ \mathrm{init}_c = \mathrm{init}_{c'},\ \mathrm{sub}_{c,c'}\ (\mathrm{new}_c\ o_c) = \mathrm{new}_{c'}\ (\mathrm{sub}_{c,c'}\ o_c),$
  $\mathrm{super}_{c',c}\ \mathrm{init}_{c'} = \mathrm{init}_c,\ \mathrm{super}_{c',c}\ (\mathrm{new}_{c'}\ o_{c'}) = \mathrm{new}_c\ (\mathrm{super}_{c',c}\ o_{c'}),$
  $\boldsymbol{D}\ \mathrm{init}_c \mid c' \leq c\}$

The variables of $V_{\Omega^{\mathsf{C}}}$ *are chosen in a way such that they do not interfere with other identifiers of the module signature.* $\boldsymbol{D}\ \mathrm{init}_c$ *denotes the definedness of the term* init_c.

The formulae $\Phi_{\Omega^{\mathsf{C}}}$ formally define $\mathrm{sub}_{c,c'}$, and $\mathrm{super}_{c',c}$ functions wrt init_c, and new_c functions: $\mathrm{sub}_{c,c'}$ or $\mathrm{super}_{c',c}$ return an object identifier of sub-type or super-type c' of c respectively, which corresponds to the object identifier given as parameter. By correspond we mean that if o_c is the n^{th} object identifier of type c then $\mathrm{sub}_{c,c'}(o_c)$ is the n^{th} object identifier of type c'.

The presentation of a CO-OPN/2 specification consists in collapsing all the ADT modules of the specification and all the ADT modules which are induced by the class modules.

Definition 4.2.2 *Presentation of a CO-OPN/2 specification.*
Let us consider a well-formed CO-OPN/2 specification $Spec = \{Md^{\mathsf{A}}_i \mid 1 \leq i \leq n\} \cup \{Md^{\mathsf{C}}_j \mid 1 \leq j \leq m\}$ *such that* $Md^{\mathsf{A}}_i = \langle \Sigma^{\mathsf{A}}_i, X_i, \Phi_i \rangle$ *and* $Md^{\mathsf{C}}_j = \langle \Omega^{\mathsf{C}}_j, P_j, I_j, V_j, \Psi_j \rangle$*. Let* Σ *be its global signature and* $Md^{\mathsf{A}}_{\Omega^{\mathsf{C}}_j} = \langle \Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}_j}, V_{\Omega^{\mathsf{C}}_j}, \Phi_{\Omega^{\mathsf{C}}_j} \rangle$ *(*$1 \leq j \leq m$*) be the ADT modules*

induced by the class modules of Spec. The presentation *of a CO-OPN/2 specification is noted Pres(Spec) and defined as follows:*

$$Pres(Spec) = \left\langle \Sigma, \bigcup_{1 \leq i \leq n} X_i \cup \bigcup_{1 \leq j \leq m} V_j \cup \bigcup_{1 \leq j \leq m} V_{\Omega_j^{\mathsf{C}}}, \bigcup_{1 \leq i \leq n} \Phi_i \cup \bigcup_{1 \leq j \leq m} \Phi_{\Omega_j^{\mathsf{C}}} \right\rangle.$$

Renaming is necessary to avoid name clashes between the various modules.

Proposition 4.2.1 *Let Spec be a well-formed CO-OPN/2 specification. Pres(Spec) is an order-sorted presentation with the structure:*

$$Pres(Spec) = \langle \Sigma, X, \Phi \rangle, \quad \textit{in which } \Sigma = \langle S^A \cup S^C, \leq^{\mathsf{A}} \cup \leq^{\mathsf{C}}, F \rangle$$

such that the following properties hold:

$$i)\ S^A \cap S^C = \varnothing, \quad ii)\ \leq^{\mathsf{A}} \subseteq S^A \times S^A, \quad iii)\ \leq^{\mathsf{C}} \subseteq S^C \times S^C, \quad iv)\ \leq^{\mathsf{A}} \cap \leq^{\mathsf{C}} = \varnothing.$$

In order to define the semantics of the presentation *Pres(Spec)* we need to define: a Σ-algebra; the least sort of a term; the interpretation of terms; the satisfaction of formulae; and the validity of a presentation.

Definition 4.2.3 *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 a S-sorted set $A = (A_s)_{s \in S}$ and a family of partial functions $F^A = (f^A_{s_1 \cdots s_n, s})_{f : s_1 \cdots s_n \to s \in F}$ where $f^A_{s_1 \cdots s_n, s}$ is a function from $A_{s_1} \times \cdots \times A_{s_n}$ into A_s such that*

i) $s \leq s' \in S$ *implies* $A_s \subseteq A'_s$

ii) $f \in F_{s_1 \cdots s_n, s} \cap F_{s'_1 \cdots s'_n, s'}$ *with* $(s_1 \cdots s_n, s) \leq (s'_1 \cdots s'_n, s')$ *implies*

$$f^A_{s_1 \cdots s_n, s} = f^A_{s'_1 \cdots s'_n, s'} \, |_{A_{s_1} \times \cdots \times A_{s_n}}$$

i.e. $f^A_{s_1 \cdots s_n, s}(a_1, \ldots, a_n) = f^A_{s'_1 \cdots s'_n, s'}(a_1, \ldots, a_n)$ *for all* $a_i \in A_{s_i}$ $i = 1, \ldots, n$, *or both are undefined for all* $a_i \in A_{s_i}, i = 1, \ldots, n$.

The equality in condition ii) is usually called *strong* equality, which requires that both sides are defined and equal, or both are undefined. We usually omit the family F^A and write A for a partial order-sorted Σ-algebra (A, F^A). Moreover, we denote the set of all order-sorted Σ-algebras by $Alg(\Sigma)$.

Proposition 4.2.2 *Let $\Sigma = \langle S, \leq, F \rangle$ be a regular signature. For every term $t \in T_{\Sigma,X}$, there exists a* least sort *$s \in S$, noted $LS(t)$, such that $t \in (T_{\Sigma,X})_s$.*

Definition 4.2.4 *Assignment, interpretation.*
Let $\Sigma = \langle S, \leq, F \rangle$ be a regular signature, X be a S-sorted set of variables and A in $Alg(\Sigma)$. An assignment *from X into A is a S-sorted function*[6] $\sigma : X \to A$. *An* interpretation *of terms of $T_{\Sigma,X}$ in A is a S-sorted partial function*[7] $\mu^\sigma : T_{\Sigma,X} \to A$ *defined as follows*

i) if $x \in X_s$ and $s \leq s'$ then $\mu^\sigma_{s'}(x) \stackrel{def}{=} \sigma_s(x)$,

ii) if $f : \to s \in F$ and $s \leq s'$ then $\mu^\sigma_{s'}(f) \stackrel{def}{=} f^A_s$,

iii) if $f : s_1, \ldots, s_n \to s \in F$ and $s \leq s'$ then

$$\mu^\sigma_{s'}(f(t_1, \ldots, t_n)) \stackrel{def}{=} \begin{cases} f^A_{s_1 \cdots s_n, s}(\mu^\sigma_{s_1}(t_1), \ldots, \mu^\sigma_{s_n}(t_n)) & \text{if all } \mu^\sigma_{s_i}(t_i) \text{ are defined,} \\ \text{undefined otherwise.} \end{cases}$$

Definition 4.2.5 *Formula satisfaction and validity.*
Let Σ be a regular signature and A be in $Alg(\Sigma)$.

$$A, \sigma \models \boldsymbol{D}\, t \iff \mu^\sigma_{LS(t)}(t) \text{ is defined,}$$

$$A, \sigma \models t = t' \iff \mu^\sigma_{LS(t)}(t) \text{ and } \mu^\sigma_{LS(t')}(t') \text{ are both undefined, or both are defined and } \mu^\sigma_{LS(t)}(t) = \mu^\sigma_{LS(t')}(t'),$$

$$A, \sigma \models (\bigwedge_{1 \leq i \leq n} \phi_i) \Rightarrow \phi' \iff A, \sigma \models \phi_i \text{ for all } i\ (1 \leq i \leq n) \text{ implies } A, \sigma \models \phi'.$$

We say that a Σ-formula ϕ is valid *in a Σ-algebra A iff $A, \sigma \models \phi$ for any assignment σ. We note this $A \models \phi$.*

Definition 4.2.6 *Validity of a Presentation.*
Let $Pres = \langle \Sigma, X, \Phi \rangle$ be a presentation in which $\Sigma = \langle S, \leq, F \rangle$. We say that Pres is valid *in a Σ-algebra A when every Σ-formula is valid in A. $Alg(Pres)$ denotes the sub-class of all Σ-algebras in which Pres is valid.*

The class of model $Alg(Pres)$ represents all the models that validate presentation $Pres$. Amongst all these models, there is a unique (up to isomorphism) model which is initial in $Alg(Pres)$. The "initial approach" consists in considering the initial model[8] as the semantics of the presentation.

Definition 4.2.7 *Semantics of a Presentation.*
Let Spec be a well-formed CO-OPN/2 specification, and let Pres(Spec) be the presentation of Spec. The semantics of Pres(Spec), noted Sem(Pres(Spec)), is the initial model ofAlg(Pres).

[6] a S-sorted function $\sigma : X \to A$ is a family of functions indexed by S written $\sigma = (\sigma_s : X_s \to A_s)_s \in S$.
[7] a S-sorted partial function is a family of partial functions indexed by S.
[8] the initial model is given by the algebra of ground terms.

The semantics of such a presentation is composed of two distinct parts. The first one consists of all the carrier sets[9] 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 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 carriers set defined by the ADT modules of the specification are usually noted $\ddot{A}$, while the object identifier algebra defined by the ADT modules induced by the class modules of the specification is $\widehat{A}$. Both $\ddot{A}$ and $\widehat{A}$ are disjoint as will be established by the next proposition.

Proposition 4.2.3 *Let Spec be a well-formed CO-OPN/2 specification and Pres(Spec) its presentation with the structure as above. Let $A = Sem(Pres(Spec))$ then*

$$A = \ddot{A} \cup \widehat{A}$$

such that:

i) $\ddot{A} = (\ddot{A}_s)_{s \in S^A}$ *is an S^A-sorted set (the model of the ADT modules of Spec);*

ii) $\widehat{A} = (\widehat{A}_c)_{c \in S^C}$ *is an S^C-sorted set (object identifier algebra);*

iii) $\ddot{A} \cap \widehat{A} = \varnothing$.

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 provides 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 sub-type relation over these types. It means that two sets of identifiers are related by inclusion if their respective types are related by sub-typing.

Indeed, each class module defines a type and a sub-type relation which are present in the ADT module induced by each class module (see Definition 4.2.1). 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 sub-type relation imposes a specific structure over the carrier sets (two carrier set are related by inclusion if they are related by sub-typing). Moreover, four operations are defined in each ADT modules 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;

[9] $Sem(Pres(Spec))$ is a S-sorted set $A = (A_s)_{s \in S}$, A_s are called carrier sets.

2. the generator new_c returns a new object identifier of type c;
3. the operation $\mathrm{sub}_{c,c'}$ maps the object identifiers of types c onto ones of type c', when $c' \leq c$;
4. the operation $\mathrm{super}_{c',c}$ maps the object identifiers of types c' into the ones of type c, when $c' \leq c$;
5. as indicated by their names, $\mathrm{super}_{c',c}$ is the inverse operation of $\mathrm{sub}_{c,c'}$ (cf the next theorem).

Theorem 4.2.1 *Let Spec be a well-formed CO-OPN/2 specification and $\leq$ be its global relation. For any types c, c' such that $c' \leq c$ the following properties hold:*

i) $\mathrm{super}_{c',c}\ (\mathrm{sub}_{c,c'}\ o_c) \ = \ o_c$, *where* $o_c : c$*;*

ii) $\mathrm{sub}_{c,c'}\ (\mathrm{super}_{c',c}\ o_{c'}) \ = \ o_{c'}$, *where* $o_{c'} : c'$*.*

4.2.2 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 partial 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*, $A = Sem(Pres(Spec))$, and the set of all types of the specification S^C.

The partial function which returns, for each class, the last object identifier used is a member of the set of partial functions[10]:

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

in which $\widetilde{A}_c = \widehat{A}_c \setminus \bigcup_{c' \leq^C_\Omega c} \widehat{A}_{c'}$ represents the proper object identifiers of the class type c (excluding the ones of any sub-type 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.

[10]The name *Loid* refers to functions that return the last object identifier used.

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
$\{newloid_c : Loid_{Spec,A} \rightarrow Loid_{Spec,A} \mid c \in S^C\}$ (new last object identifier) defined as:

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

The second function retained by the system throughout the evolution of the system returns the set of the alive objects of a given class. It belongs to the set of partial functions[11]:

$$Aoid_{Spec,A} = \{a : S^C \rightarrow C \mid C \subseteq \mathcal{P}(\widehat{A}),\ a(c) \in \mathcal{P}(\widetilde{A}_c)\}.$$

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 $\{newaoid_c : Aoid_{Spec,A} \times \widehat{A} \rightarrow Aoid_{Spec,A} \mid c \in S^C\}$ (new alive object identifiers) defined as:

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

Both families of functions $newloid_c$ and $newaoid_c$ are used in the inference rules concerning the creation of new instances, see Definition 4.2.16 below.

The set of functions $\{remaoid_c : Aoid_{Spec,A} \times \widehat{A} \rightarrow Aoid_{Spec,A} \mid c \in S^C\}$ 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:

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

This family of functions is necessary when the destruction of class instances is considered, see Definition 4.2.16 below.

Here are 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 4.2.18; they have been developped in order to allow simultaneous creation and destruction of objects. 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. These operators will be explained in more details later.

[11]The name *Aoid* refers to functions that return the alive (or active) object identifiers. The notation $\mathcal{P}(A)$ represents the *power set* of a set A.

Definition 4.2.8 *Operators.*

$\vartriangle$: $Loid_{Spec,A} \times Loid_{Spec,A} \times Loid_{Spec,A} \rightarrow Loid_{Spec,A}$ *such that*

$$(\forall c \in S^C)\ (l' \vartriangle_l l'')(c) = \begin{cases} l'(c) & \textit{if } l'(c) \neq l(c) \wedge l''(c) = l(c), \\ l''(c) & \textit{if } l'(c) = l(c) \wedge l''(c) \neq l(c), \\ l(c) & \textit{otherwise.} \end{cases}$$

$\triangleq$: $Loid_{Spec,A} \times Loid_{Spec,A} \times Loid_{Spec,A}$ *such that*

$$(\forall c \in S^C)\ (l' \triangleq_l l'')(c) = ((l(c) = l'(c) = l''(c)) \vee (l'(c) \neq l''(c)))$$

$\cup$: $Aoid_{Spec,A} \times Aoid_{Spec,A} \rightarrow Aoid_{Spec,A}$ *such that*

$$(\forall c \in S^C)\ (a \cup a')(c) = a(c) \cup a'(c)$$

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

$P(a_1, a'_1, a_2, a'_2) \iff$

$$\begin{aligned} (\forall c \in S^C)\ \ & (((a_1(c) \cap ((a_2(c) \setminus a'_2(c)) \cup (a'_2(c) \setminus a_2(c)))) = \varnothing) \wedge \\ & ((a'_1(c) \cap ((a_2(c) \setminus a'_2(c)) \cup (a'_2(c) \setminus a_2(c)))) = \varnothing) \wedge \\ & ((a_2(c) \cap ((a_1(c) \setminus a'_1(c)) \cup (a'_1(c) \setminus a_1(c)))) = \varnothing) \wedge \\ & ((a'_2(c) \cap ((a_1(c) \setminus a'_1(c)) \cup (a'_1(c) \setminus a_1(c)))) = \varnothing)) \end{aligned}$$

4.2.3 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-sets of algebraic values. However, this current notion of marking is not suitable in the CO-OPN/2 context. Remember that CO-OPN/2 is a structured formalism which allows the description of a system by means of a collection of entities. Moreover, this collection can dynamically increase or decrease in terms of number of entities. This implies that the system has to retain two additional informations as explained above. In that case, the state of a system consists of three elements. The first two ones manage the object identifiers, i.e., a partial function to memorise the last oids used, and a second function to memorise which oids are created and alive. The third element consists in 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. This is a more sophisticated notion of marking than the one presented in the section related to the algebraic nets. This new notion of marking is necessary in the CO-OPN/2 context because, here, a net does not describe a single instance but a class of objects which can be dynamically created.

Definition 4.2.9 *Marking, definition domain, state.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. Let S be the set of sorts and types

of Spec, and let P be the S-sorted set of all places of Spec. A marking *is a partial function $m : \widehat{A} \times P \rightarrow [A]$[12] 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*

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

Notation 4.2.10 *Initial marking, State space.*
A marking m is noted $\perp$ when $Dom_{Spec,A}(m) = \varnothing$. The state *of a system over Spec and A is a triple $(l,a,m) \in Loid_{Spec,A} \times Aoid_{Spec,A} \times Mark_{Spec,A}$. We denote the* state space, *i.e. the set of all states, by $State_{Spec,A}$.*

4.2.4 Transition System

The notion of transition system is an essential element of the semantics of a CO-OPN/2 specification. In the context of algebraic nets, a transition system is defined as a graph in which the arcs are labelled by a multi-set of transition names, in order to allow the simultaneous firing of transitions. Although CO-OPN/2 is also based on a step semantics, the events of a system described by a CO-OPN/2 specification are not restricted to transition names, but are much more sophisticated. The introduction of the distinction between invisible and observable events, the synchronisations between the objects and then the parameterised transitions (methods), as well as the three operators '//', '..', and '$\oplus$', led us to adopt a different notion of transition system. With this new notion of transition system the state space is defined as above, and each transition is labelled by an element of $\mathbf{E}_{A,M,\widehat{A},S^C}$ (see Definition 4.1.17).

Definition 4.2.11 *Transition system.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. Let S^C and M be respectively the set of types and the set of methods of Spec. A transition system *over Spec and A is a set of triples*

$$TS_{Spec,A} \subseteq State_{Spec,A} \times \mathbf{E}_{A,M,\widehat{A},S^C} \times State_{Spec,A}.$$

Notation 4.2.12 *Set of all transition systems.*
The set of all transitions systems over Spec and A is noted $\mathbf{TS}_{Spec,A}$. A triple $\langle st, e, st' \rangle$ is called a transition, *and is commonly written either $st \xrightarrow{e} st'$ or $st \xRightarrow{e} st'$.*

[12]the semantic multi-set extension of model A is noted $[A]$; it consists of adding to A, for all sorts $[s]$ such that $s \in S$, the free monoid induced by A_s, namely $[A_s]$, and the 3 multi-set operations.

4.2.5 Inference Rules

In order to construct the semantics of a CO-OPN/2 specification which consists mainly of a transition system, we provide here a set of inference rules expressed as *Structural Operational Semantics* [53], a well-known formalism used for describing the computational meaning of systems.

The idea behind the construction of the semantics of a specification composed of several class modules, is to build the semantics of each individual class modules first, and compose them subsequently by means of synchronisations. This semantics of an individual class module is called a *partial semantics* in the sense that it is not yet composed with other partial semantics (with synchronisations), and it still contains some invisible events.

The distinction between the observable events (in relation with the methods) and the ones that are invisible (in relation with the internal transitions τ) implies a *stabilisation process*. This process is necessary so that the observable events are performed only when all invisible events have occurred. A system in which no more invisible event can occur is said to be in a *stable* state.

Another operation called the *closure operation* is necessary to take into account the three operators (sequence, simultaneity, alternative) as well as the synchronisation requests. Such a closure operation determines all the sequential, concurrent, and non-deterministic behaviours of a given semantics and composes the different parts of the semantics by means of synchronisations.

The successive composition of both the stabilisation process and the closure operation on all the class modules of the specification will finally provide a transition system in which:

- all the sequential, concurrent, and non-deterministic behaviours will have been inferred;
- all the synchronisation requests will have been solved;
- all the invisible or spontaneous events will have been eliminated; in other words every state of the transition system is stable.

Such a transition system will be considered as the semantics of a CO-OPN/2 specification.

As we will see, the inference rules introduced further for the construction of the semantics of a specification, generate two kinds of transitions. The transitions which involve both invisible and observable events are noted by a single arrow $st \xrightarrow{e} st'$, while the ones which involve only observable events are noted by a double arrow $st \stackrel{e}{\Longrightarrow} st'$. A transition system can then include two kinds of transitions which must be distinguished during the construction of the semantics. Thus, in order to identify these two kinds of transitions, any transition system is $\{\rightarrow, \Rightarrow\}$-disjointly-sorted. This means that any transition system is divided into two disjoint sub-transition systems: the sub-transition system which contains only $\rightarrow$-transitions and the one which is composed of $\Rightarrow$-transitions.

The inference rules are arranged into three categories and realize the following tasks:

- the rules Class and Mono build, for a given class, its partial transition system according to its methods, places, and behavioural formulae; Create and Destroy take charge of the dynamic creation and destruction of class instances;
- Seq, Sim, Alt-1, and Alt-2 generate all deductible sequential, concurrent, and non-deterministic behaviours; Sync composes the various partial semantics by means of the synchronisation requests between the transition systems;
- Stab-1 and Stab-2, involved in the stabilisation process, "eliminates" all invisible or spontaneous events which correspond to internal transitions of the classes.

Before introducing the set of inference rules designed for the construction of the transition system associated to a given CO-OPN/2 specification, we first define some basic operators for markings and for the management of object identifiers. These operators are intensively used in those inference rules.

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 on their common places. As for the *fusion* of markings '$m_1 \trianglelefteq m_2$', it returns a marking whose values are those of m_1 and those of m_2 which do not appear in m_1.

Definition 4.2.13 *Sum of markings, common markings, fusion of markings.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. Let S and P be respectively the set of sorts and types and the set of places of Spec.

– *The* sum *of two markings is* $+ : Mark_{Spec,A} \times Mark_{Spec,A} \to Mark_{Spec,A}$

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

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

– *The* common markings *predicate is* $\bowtie : Mark_{Spec,A} \times Mark_{Spec,A}$

$$m_1 \bowtie m_2 \iff \forall (o,p) \in \widehat{A} \times P$$
$$(o,p) \in Dom(m_1) \cap Dom(m_2) \implies m_1(o,p) = m_2(o,p)\ ;$$

– *The* fusion *of two markings is* $\trianglelefteq : Mark_{Spec,A} \times Mark_{Spec,A} \rightarrow Mark_{Spec,A}$

$$m_1 \trianglelefteq m_2 = m_3 \text{ such that } \forall (o,p) \in \widehat{A} \times P$$

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

Partial Semantics of a Class

We now develop the partial semantics of a given class module of a specification. First of all, we give some auxiliary definitions used in the subsequent construction of the partial semantics.

Definition 4.2.14 *Evaluation of terms in places.*
Let Spec be a well-formed CO-OPN/2 specification, $A = Sem(Pres(Spec))$*, and a class module* $Md^{\mathsf{C}} = \langle \Omega^{\mathsf{C}}, P, I, X, \Psi \rangle$ *of type c. Let* S^A*,* S^C*,* M *be respectively the set of sorts, types and methods of Spec, and let* Σ *be the global signature of Spec.*

The evaluation of terms of $T_{[\Sigma],X}$ *indexed*[13] *by* P*, for a given assignment of the variables* $\sigma : X \rightarrow A$*, and a given class instance* o*, into the set of markings* $Mark_{Spec,A}$ *is noted* $[\![(t_p)_{p\in P}]\!]^{\sigma}_{o}$*, and defined in the following way:*

$$[\![(t_p)_{p\in P}]\!]^{\sigma}_{o} = m \text{ such that } (\forall p \in P)(\forall o' \in \widehat{A})$$

$$m(o',p) = \begin{cases} [\![t_p]\!]^{\sigma} \text{ if } o' = o \text{ and } p \in P, \\ \text{undefined otherwise.} \end{cases}$$

$[\![.]\!]^{\sigma} : T_{[\Sigma],X} \rightarrow [A]$ *is the usual interpretation of terms of* $T_{[\Sigma],X}$*, given an assignment* σ *of the variables.*

Such terms form, for example, a pre/post condition of a behavioural formula or an initial marking. This kind of evaluation is used in the inference rules, as shown in the next definition.

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

Definition 4.2.15 *Event evaluation.*
Let Spec be a well-formed CO-OPN/2 specification, Σ *be the global signature of Spec,* X

[13] remember that a term indexed by a place $p \in P_s$ is of type $[s]$.

be the set of variables of Spec, $A = Sem(Pres(Spec)))$, $\sigma : X \to A$ *be an assignment of the variables,* $\mu^\sigma : T_{\Sigma,X} \to A$ *be the interpretation of terms, and* $s, c \in S^C$.

The event evaluation $[\![\ ]\!]^\sigma : \mathbf{E}_{(T_{\Sigma,X}),M,(T_{\Sigma,X})_s,\{c\}} \to \mathbf{E}_{A,M,\widehat{A},\{c\}}$ *with* $s \in S^C$ *naturally follows from Definition 4.1.17 and is inductively defined as:*

$$
\begin{aligned}
[\![t.\tau]\!]^\sigma &= \mu(t)^\sigma.\tau \\
[\![t.m(a_1,\ldots,a_n)]\!]^\sigma &= \mu(t)^\sigma.m(\mu(a_1)^\sigma,\ldots,\mu(a_n)^\sigma) \\
[\![t.\mathrm{create}]\!]^\sigma &= \mu(t)^\sigma.\mathrm{create} \\
[\![t.\mathrm{destroy}]\!]^\sigma &= \mu(t)^\sigma.\mathrm{destroy} \\
[\![Event' \text{ with } Event'']\!]^\sigma &= [\![Event']\!]^\sigma \text{ with } [\![Event'']\!]^\sigma \\
[\![Event' \ op \ Event'']\!]^\sigma &= [\![Event']\!]^\sigma \ op \ [\![Event'']\!]^\sigma
\end{aligned}
$$

for all $Event, Event', Event'' \in \mathbf{E}_{(T_{\Sigma,X}),M,(T_{\Sigma,X})_s,\{c\}}$ *with* $s \in S^C$, *for all* $t \in (T_{\Sigma,X})_s$ *and for all methods* $m \in M_{s,s_1,\ldots,s_n}$ *with* $s \in S^C$ *and* $s_i \in S$, *and for all synchronisation operators* $op \in \{\,//\,,\ .\ .\ ,\oplus\}$.

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

Finally, the satisfaction of a condition of a behavioural formula is defined as:

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

Definition 4.2.16 *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* $\Omega^{\mathsf{C}} = \langle \{c\}, \leq^{\mathsf{C}}, M, O \rangle$. *The* partial semantics *of* Md^{C} *is the transition system noted* $PSem_{Spec,A}(Md^{\mathsf{C}})$ *which is the least fixed point resulting from the application of the inference rules:* Class, Mono, Create, *and* Destroy *given in Table 4.1.*

The inference rules introduced in Table 4.1 can be informally formulated as follows:

- The Class rule generates the basic observable – as well as invisible – transitions that follow from the behavioural formulae 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:
  1. there is a behavioural formula $Event :: Cond \Rightarrow Pre \to Post$ in the class;
  2. there exists an assignment $\sigma : X \to A$;

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

$$\text{CREATE}\ \frac{\begin{array}{l} \exists \sigma : X \rightarrow A, \\ l' = \mathit{newloid}_c(l),\ a' = \mathit{newaoid}_c(a, o),\ o = l'(c),\ o \notin a(c) \end{array}}{\langle l, a, \perp \rangle \xrightarrow{o.\text{create}} \langle l', a', [\![I]\!]_o^\sigma \rangle}$$

$$\text{DESTROY}\ \frac{o \in a(c),\ a' = \mathit{remaoid}_c(a, o)}{\langle l, a, \perp \rangle \xrightarrow{o.\text{destroy}} \langle l, a', \perp \rangle}$$

$$\text{MONO}\ \frac{\langle l, a, m \rangle \xrightarrow{e} \langle l', a', m' \rangle}{\langle l, a, m + m'' \rangle \xrightarrow{e} \langle l', a', m' + m'' \rangle}$$

for all l, l' in $\mathit{Loid}_{\mathit{Spec},A}$, for all a, a' in $\mathit{Aoid}_{\mathit{Spec},A}$, for all m, m', m'' in $\mathit{Mark}_{\mathit{Spec},A}$, for all o in $\widetilde{A}_c$, and for all e in $\mathbf{E}_{A,M,\widehat{A},\{c\}}$.

Table 4.1: Inference Rules for the Partial Semantics Construction.

3. all the equations of the global condition are satisfied ($A, \sigma \models Cond$);
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 behavioural formula.

- The CREATE rule generates the transitions aimed at the dynamic creation of new objects provided:

  1. for any last object identifier function l and any alive object identifier function a;
  2. a new last object identifier function is computed ($l' = newloid_c(l)$);
  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' and of an updated function a' in which the new object identifier has been added to the set of created objects of the class.

- 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 the object is alive.

- The MONO rule (for monotonicity) generates all the firable transitions from the transitions already generated.

Proposition 4.2.4 *Well-definedness of the partial semantics.*
Let Spec be specification and $A = Sem(Pres(Spec))$. The partial semantics of a class module $PSem_{Spec,A}(Md^{\mathsf{C}})$ is well-defined.

The construction of the whole semantics of a CO-OPN/2 specification composed of several class modules consists in considering each partial semantics and combine them by means of the successive composition of a stabilisation process and a closure operation. This cannot be done in random order because observable events (methods) can be performed only when invisible events have occurred.

In order to build the whole semantics of a specification *Spec*, we introduce a total order over the class modules of *Spec* which depends on the partial order induced by the clientship relation D^{C}_{Spec}. This total order is used to construct the semantics; it is noted $\sqsubset$ and defined such that $D^{\mathsf{C}}_{Spec} \subseteq \sqsubset$.

Given Md^{C}_0 the least module of the total order and the fact that $Md^{\mathsf{C}}_i \sqsubset Md^{\mathsf{C}}_{i+1}$ $(0 \leq i < n)$, we introduce the partial semantics of all the modules Md^{C}_i $(0 \leq i \leq n)$ of a specification from the bottom to the top.

Stabilisation Process

The purpose of the stabilisation process is to provide a transition system in which all the invisible events (internal transitions) have been taken into account. More precisely, the stabilisation process consists in merging all the observable events and the invisible ones into one step.

Thus, the stabilisation process proceeds in two stages. The first stage is the application of two inference rules on a given transition system to produce the merged transitions. This step is called the *pre-stabilisation*. The second step produces the intended transition system which contains only the relevant transitions, i.e. all the transitions except the transitions which do not lead to a stable state.

We observe that the STAB-1 and STAB-2 involve a new kind of transitions noted with a double arrow ($\Rightarrow$-transitions). This kind of transitions is introduced in order to distinguish between a transition system composed of stable states and another in which some invisible events have to be taken into account.

Definition 4.2.17 *Stabilisation process.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. The stabilisation process *consists of the function Stab* : $\mathbf{TS}_{Spec,A} \to \mathbf{TS}_{Spec,A}$ *defined as follows:*

$$\begin{aligned} Stab(TS) = &\{m \xrightarrow{e} m' \in TS\} \cup \\ &\{m \xRightarrow{e} m' \in PreStab(TS) \mid \nexists\, m' \xrightarrow{\tau} m'' \in PreStab(TS)\} \end{aligned}$$

in which PreStab : $\mathbf{TS}_{Spec,A} \to \mathbf{TS}_{Spec,A}$ *is a function such that $PreStab(TS)$ is the least fixed point which results from the application on TS of the inference rules*[14] STAB-*1 and* STAB-*2 given in Table 4.2.*

The inference rules introduced in Table 4.2 can be informally formulated as follows:

- Rule STAB-1 generates all the observable events which can be merged with invisible events if they lead to an unstable state; note that neither the pure internal transitions nor the internal transitions asking to be synchronised with some partners are considered by this rule;
- Rule STAB-2 merges an event leading to a non-stable state and the invisible event which can occur "in sequence". This rule is very similar the SEQ introduced later when the closure operation is presented. Thus, the same comments regarding its functioning and the meaning of the operators involved in the rule hold.

It is worthwhile to note that:

[14]The result of the application of the inference rules on TS obviously includes TS itself.

$$\text{STAB-1}\ \frac{e \neq o.\tau,\ e \neq o.\tau \text{ with } e',\ \langle l,a,m\rangle \xrightarrow{e} \langle l',a',m'\rangle}{\langle l,a,m\rangle \stackrel{e}{\Longrightarrow} \langle l',a',m'\rangle}$$

$$\text{STAB-2}\ \frac{m'_1 \bowtie m_2,\ \langle l,a,m_1\rangle \stackrel{e}{\Longrightarrow} \langle l',a',m'_1\rangle,\ \langle l',a',m_2\rangle \xrightarrow{o.\tau} \langle l'',a'',m'_2\rangle}{\langle l,a,m_1 \trianglelefteq m_2\rangle \stackrel{e}{\Longrightarrow} \langle l'',a'',m'_2 \trianglelefteq m'_1\rangle}$$

for all $m, m', m_1, m'_1, m_2, m'_2$ in $Mark_{Spec,A}$, for all l, l', l'' in $Loid_{Spec,A}$, for all a, a', a'' in $Aoid_{Spec,A}$, for all o in $\widehat{A}$, and for all e, e' in $\mathbf{E}_{A,M,\widehat{A},S^C}$.

Table 4.2: Inference Rules of the Stabilisation Process.

1. Generally, the states, in particular the marking domains, are not identical, and both the operators '$\bowtie$' and '$\trianglelefteq$' play an important role as commented and illustrated below when the SEQ inference rule involved in the closure operation is presented.

2. When infinite sequences of transitions are encountered, the stabilisation process does not retain any collapsed transition. From an operational point of view, such infinite sequence of internal transitions can be considered as a program that loops. However, in a distributed software setting, when an object (or a group of objects) loops, it does not mean that the whole system loops; it simply means that such an object is not able to give any more services and, therefore, it can be ignored.

3. The stabilisation process has to retain the $\rightarrow$-transitions for the inductive construction of the whole semantics presented further.

Closure Operation

The closure operation consists of adding to a given transition system all the sequential, simultaneous, alternative behaviours, and to perform the synchronisation requests. A set of inference rules are provided for these aims.

Definition 4.2.18 *Closure operation.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. The closure operation *$Closure$: $\mathbf{TS}_{Spec,A} \rightarrow \mathbf{TS}_{Spec,A}$ is such that $Closure(TS)$ is the least fixed point which results from the application on TS of the inference rules* SEQ, SIM, ALT-*1*, ALT-*2, and* SYNC *given in Table 4.3.*

The inference rules of Table 4.3 can be informally formulated as follows:

$$\text{SEQ}\ \frac{m'_1 \bowtie m_2,\ \langle l,a_1,m_1\rangle \overset{e_1}{\Longrightarrow} \langle l',a'_1,m'_1\rangle,\ \langle l',a'_2,m_2\rangle \xrightarrow{e_2} \langle l'',a'_2,m'_2\rangle}{\langle l,a,m_1 \trianglelefteq m_2\rangle \xrightarrow{e_1\,..\,e_2} \langle l'',a'_2,m'_2 \trianglelefteq m'_1\rangle}$$

$$\text{SIM}\ \frac{\begin{array}{l} l' \triangleq_l l'',\ \triangle(a_1,a'_1,a_2,a'_2), \\ \langle l,a_1,m_1\rangle \xrightarrow{e_1} \langle l',a'_1,m'_1\rangle,\ \langle l,a_2,m_2\rangle \xrightarrow{e_2} \langle l'',a'_2,m'_2\rangle \end{array}}{\langle l,a_1\cup a_2,m_1+m_2\rangle \xrightarrow{e_1\,//\,e_2} \langle l' \vartriangle_l l'',a'_1\cup a'_2,m'_1+m'_2\rangle}$$

$$\text{ALT-1}\ \frac{\langle l,a,m\rangle \xrightarrow{e_1} \langle l',a',m'\rangle}{\langle l,a,m\rangle \xrightarrow{e_1 \oplus e_2} \langle l',a',m'\rangle} \qquad \text{ALT-2}\ \frac{\langle l,a,m\rangle \xrightarrow{e_1} \langle l',a',m'\rangle}{\langle l,a,m\rangle \xrightarrow{e_2 \oplus e_1} \langle l',a',m'\rangle}$$

$$\text{SYNC}\ \frac{\begin{array}{l} l' \triangleq_l l'',\ \triangle(a_1,a'_1,a_2,a'_2), \\ \langle l,a_1,m_1\rangle \xrightarrow{e_3 \text{ with } e_2} \langle l',a'_1,m'_1\rangle,\ \langle l,a_2,m_2\rangle \overset{e_2}{\Longrightarrow} \langle l'',a'_2,m'_2\rangle \end{array}}{\langle l,a_1\cup a_2,m_1+m_2\rangle \xrightarrow{e_3} \langle l' \vartriangle_l l'',a'_1\cup a'_2,m'_1+m'_2\rangle}$$

for all m_1, m'_1, m_2, m'_2 in $Mark_{Spec,A}$, for all l, l', l'' in $Loid_{Spec,A}$, and for all $a, a', a_1, a'_1, a_2, a'_2$ in $Aoid_{Spec,A}$, for all e_1, e_2 in $\mathbf{E}_{A,M,\widehat{A},S^C}$ which are not equal to $o.\tau$ or to $o.\tau$ with e' and for all e_3 in $\mathbf{E}_{A,M,\widehat{A},S^C}$.

Table 4.3: Inference Rules of the Closure Operation.

- Rule SEQ infers the sequence of two transitions provided the markings shared between m'_1 and m_2 are equal. Note that the creation of object requires that the usual l and a functions are different for each transition. The double arrow under the e_1 event forces that e_1 leads to a stable state. This guarantees that all the invisible events are taken into account before inferring the sequential behaviours.

- Rule SIM infers the simultaneity of two transitions, provided some constraints on functions l and a are satisfied. The purposes of these constraints are:

  1. to prevent an event from using an object being created by the other event (i.e. which does not already exist);
  2. to prevent an event from using an object being destroyed by the other event (i.e. which does not exit any more).

  The operators of Definition 4.2.8 are used to:

  1. $a \triangleq a''$ eliminates (for e_1) the objects which are created by e_2; their use in the upper left derivation tree is therefore not allowed;

2. $a \triangleq a'$ eliminates (for e_2) the objects which are created by e_1; their use in the upper right derivation tree is therefore not allowed;
3. $a' \cup a''$ makes simply the union of the a' and a'' for each type;
4. predicate $\triangle(a_1, a'_1, a_2, a'_2)$ 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 $a'_1(c)$.

- Rules ALT-1 and ALT-2 provide all the alternative behaviours. Two rules are necessary for the commutativity of the alternative operator $\oplus$.

- Rule SYNC "solves" the synchronisation requests. It generates the event which behaves in the same way as the event 'e_1 with e_2' asking to be synchronised with the event e_2. The double arrow under the event e_2 guarantees that the synchronisations are performed with events leading to stable states. Note that e_1 can be an invisible event because internal transitions may ask for a synchronisation; and that event e_1 can occur only if event e_2 can occur simultaneously.

  The similarities between the SIM and SYNC are not surprising because of the synchronous nature of CO-OPN/2.

The following results ensure that several intuitive but important intended events can never occur in a system which is built by means of such formal system.

Proposition 4.2.5 *The following events can never occur:*

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 synchronisation of the use of an object with the creation (or destruction) of this object;*

Corollary 4.2.2 *The following events can never occur:*

1. *the multiple creation of the same object;*
2. *the multiple destruction of the same object;*

3. the destruction followed by the creation of the same object;

Before defining how the stabilisation process and the closure operation are combined in order to obtain the whole semantics of a CO-OPN/2 specification, we provide here a proposition which states that both these operations are well-defined.

Proposition 4.2.6 *Stab and Closure are well-defined.*
Let Spec be specification and $A = Sem(Pres(Spec))$. *Stab and Closure are well-defined functions for any transition system* $TS \in \mathbf{TS}_{Spec,A}$.

4.2.6 Semantics of a CO-OPN/2 Specification

The whole semantics, expressed by the following definition, is calculated starting from the partial semantics of the least object (for a given total order), and repeatedly adding the partial semantics of a new object. For each new object added to the system, we observe that the stabilisation process is obviously performed before the closure operation. Moreover, let us note that the limit tending towards infinity is required to cover the special case of recursive synchronisations.

Definition 4.2.19 *Semantics of a specification for a given total order.*
Let Spec be a specification composed of a set of class modules $\{Md_j^{\mathsf{C}} \mid 0 \leq j \leq m\}$ *and* $A = Sem(Pres(Spec))$. *Let* $\sqsubset$ *be a total order over the class modules such that* $D_{Spec}^{\mathsf{C}} \subseteq \sqsubset$. *The* semantics of *Spec* for $\sqsubset$ *is noted* $Sem_A^{\sqsubset}(Spec)$ *and inductively defined as:*

$$Sem_A^{\sqsubset}(\emptyset) = \emptyset$$

$$Sem_A^{\sqsubset}(\{Md_0^{\mathsf{C}}\}) = \lim_{n\to\infty} (Closure \circ Stab)^n (PSem(Md_0^{\mathsf{C}}))$$

$$Sem_A^{\sqsubset}(\cup_{0\leq j\leq k}\{Md_j^{\mathsf{C}}\}) = \lim_{n\to\infty} (Closure \circ Stab)^n (Sem_A^{\sqsubset}(\cup_{0\leq j\leq k-1}\{Md_j^{\mathsf{C}}\}) \cup PSem_A(Md_k^{\mathsf{C}}))$$

for $1 \leq k \leq m$.

The above definition of the semantics is not independent of the total order. Thus, we define the semantics of a CO-OPN/2 specification when it does not depend of such a total order.

Definition 4.2.20 *Semantics of a specification.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. *The* semantics of *Spec noted* $Sem_A(Spec)$ *is defined as the* $Sem_A(Spec) = Sem_A^{\sqsubset}(Spec)$ *such that it is independent of the total order* $\sqsubset$ *over the class modules of Spec.*

Finally, we define the *step semantics* of a CO-OPN/2 specification from the above semantics in which we only retain the $\Rightarrow$-transitions whose events are atomic or simultaneous. Moreover, we only consider the transitions from states which are reachable from the initial state.

Definition 4.2.21 *Step Semantics of a specification.*
Let Spec be a specification and $A = Sem(Pres(Spec))$. *The* step semantics of *Spec, noted* $SSem_A(Spec)$*, is defined as the greatest set in* $\mathbf{TS}_{Spec,A}$ *such that* $SSem_A(Spec) \subseteq Sem_A(Spec)$ *and for any transition* $st \stackrel{e}{\Longrightarrow} st'$ *in* $SSem_A(Spec)$ *the following properties holds*[15]*:*

i) $e = e_1 \mathbin{//} e_2 \mathbin{//} \cdots \mathbin{//} e_n$*, where* $e_i = o_i.m_i(a_{1i}, \ldots, a_{ki}) \quad (1 \leq i \leq n)$ *;*

ii) $\langle \bot, \varnothing, \bot \rangle \Vdash^* st$ *;*

where $e, e_i \in \mathbf{E}_{A,M,\widehat{A},S^C}$ $(1 \leq i \leq n)$.

For a given CO-OPN/2 specification *Spec*, the transition system defined by the step semantics *is* the semantics of *Spec*.

Example 4.2.22 *Let Spec be the CO-OPN/2 specification of Example 4.1.24, a total order for the Class modules of Spec is the following:*

$$\text{PackagingUnit} \sqsubset \text{PralineContainer} \sqsubset \text{ConveyorBelt} \sqsubset \text{Packaging}\,.$$

The semantics of Spec is defined since any other order with PackagingUnit *at the root produces the same transition system. Indeed, Class module* PackagingUnit *is the unique Class module of Spec which requires synchronisations with other Class modules.*

Transitions of the step semantics of Spec contain events made with the various method names appearing in the Class modules of Spec. It is worth mentioning that, due to the stabilisation process, transitions filling *and* store *of Class module* PackagingUnit *must be fired as many times as necessary in order to reach a stable state. Therefore, once method* take *has been fired (once, twice, or more times), the stored box(es) are filled with chocolates (stabilisation of transition* filling*) and stored (stabilisation of transition* store*) before method* take *is newly firable.*

[15]The symbol $\Vdash^*$ corresponds to the reflexive transitive closure of the reachability relation defined for the $\Rightarrow$-transitions. The initial state is noted $\langle \bot, \varnothing, \bot \rangle$.

Chapter 5

CO-OPN/2 Refinement

Chapter 3 defines a general theory of refinement of model-oriented formal specifications that is based on the preservation of essential properties collected in a contract. The scope of the current chapter is to apply this theory to the CO-OPN/2 formal specifications language presented in Chapter 4.

The refinement theory can be applied to a model-oriented formal specifications language, in so far as a logic is provided for expressing formulae on specifications, as well as a satisfaction relation on models of specifications and formulae. The logic used to express formulae on CO-OPN/2 specifications is the Hennessy-Milner temporal logic (HML). This logic is particularly well-suited for CO-OPN/2 since, first, it enables to distinguish models of CO-OPN/2 specifications as finely as the bisimulation equivalence; and second, it facilitates the practical verification of refinement steps.

This chapter first defines HML formulae on CO-OPN/2 specifications, as well as the satisfaction relation on CO-OPN/2 models and HML formulae. Second, it defines contractual CO-OPN/2 specifications, a refine relation, a formula refinement, and a refinement relation on contractual CO-OPN/2 specifications. Finally, it presents some compositional results on contractual CO-OPN/2 specifications and their refinement.

5.1 Hennessy-Milner Logic

In the framework of the CO-OPN/2 language, the Hennessy-Milner logic [41] (HML) is currently used in the formal testing activity. Since this thesis aims at defining a refinement and an implementation of CO-OPN/2 specifications based on contracts, it is natural to use HML for expressing formulae of contracts. Thus, the implementation phase and the test phase are linked by the use of HML formulae. In addition, the same languages, i.e., CO-OPN/2 and HML, are used during the development phase, the implementation phase and the test phase. A supplementary argument in favour of HML is provided by its power of discriminating CO-OPN/2 specifications as finely as the bisimulation equivalence - as shown by Hennessy and Milner in [41].

A HML formula is a sequence of observable events. An observable event is either the firing of a method of a CO-OPN/2 object, or the parallel firing of several methods of CO-OPN/2 objects. We call these events *observable*, because their evaluation corresponds to an event of the step semantics of the specification. Indeed, the step semantics provides all the events that a user of the specification may observe; events that are not in the step semantics cannot be observed.

A HML formula is satisfied by the step semantics of a CO-OPN/2 specification, if every event constituting the formula can be evaluated as an event of the step semantics, and if the sequence of the evaluated events corresponds to the beginning of an execution path (a sequence of events) of the step semantics.

Throughout this chapter, we use the following notation:

Notation 5.1.1 *Let* $Spec = \{(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_i \mid 1 \le i \le n\} \cup \{(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_j \mid 1 \le j \le m\}$ *be a well-formed CO-OPN/2 specification, and*

$$\Sigma = \left\langle \bigcup_{1 \le i \le n} S^{\mathsf{A}}_i \cup \bigcup_{1 \le j \le m} \{c_j\},\ \le,\ \bigcup_{1 \le i \le n} F_i \cup \bigcup_{1 \le j \le m} F_{\Omega^{\mathsf{C}}_j} \right\rangle.$$

be the global signature of Spec, and

$$\Omega = \left\langle \bigcup_{1 \le j \le m} \{c_j\},\ (\bigcup_{1 \le j \le m} \le^{\mathsf{C}}_j)^*,\ \bigcup_{1 \le j \le m} M_j,\ \bigcup_{1 \le j \le m} O_j \right\rangle.$$

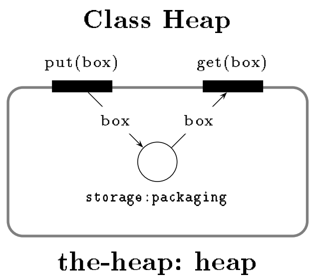
be the global interface of Spec.
We denote:

$$S^A = \bigcup_{1 \le i \le n} S^{\mathsf{A}}_i \qquad S^C = \bigcup_{1 \le j \le m} \{c_j\} \qquad S = S^A \cup S^C$$
$$F^A = \bigcup_{1 \le i \le n} F_i \qquad F^C = \bigcup_{1 \le j \le m} F_{\Omega^{\mathsf{C}}_j} \qquad F = F^A \cup F^C$$
$$M = \bigcup_{1 \le j \le m} M_j \qquad O = \bigcup_{1 \le j \le m} O_j.$$

This section defines a running example, the syntax of HML formulae, and their semantics.

5.1.1 Running Example

Examples of this section use the CO-OPN/2 Class module of Figure 5.1.




```
Class Heap;
Interface
  Use   Packaging;
  Type  heap;
  Object the-heap;
  Methods put _, get _ : packaging;
Body
  Place storage _ : packaging;
  Axioms
    put box :: -> storage box;
    get box :: storage box -> ;
  Where
    box : packaging;
End Heap;
```

Figure 5.1: CO-OPN/2 `Heap` Class Module

The right of part of Figure 5.1 shows the textual representation of the CO-OPN/2 Class module `Heap`. Its graphical representation is on the left part of the figure. This Class module defines a type `heap`, and a static object `the-heap`. Every instance of this type stores boxes of type `packaging`, and removes boxes when requested to do so. Boxes are not necessarily removed in the order of their storage. Method `put(box)` stores `box` into place `storage`, method `get(box)` removes `box` from that place. Class module `Packaging` defines type `packaging`, i.e., chocolate boxes, and a method `fill` for filling the box with pralines.

Example 5.1.2 below will be used as a running example throughout this section. It defines the minimal well-formed CO-OPN/2 specification that enables to define CO-OPN/2 Class module `Heap`. According to the examples of Chapter 4, the minimal CO-OPN/2 specification that enables to define the `Heap` class is made of the following modules: ADT modules `Chocolate`, `Capacity`, `Booleans`, `Naturals`; and Class modules `Packaging`, and `Heap`. Given ADT and Class modules textual representations, their respective abstract modules are easily retrieved following Definitions 4.1.15 and 4.1.20.

Example 5.1.2 *Running Example.*
We define the following CO-OPN/2 specification:

$$\begin{aligned} Spec_0 = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Chocolate}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Capacity}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, \\ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Packaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Heap}}\}. \end{aligned}$$

Appendix A gives the complete textual CO-OPN/2 specifications of $Spec_0$ as well as its CO-OPN/2 abstract specification, global signature and global interface (see Definition 4.1.8).

5.1.2 HML Formulae

HML formulae are made of sequences of observable events. Observable events are syntactical terms corresponding to: the creation of a new object, the destruction of an object, the firing of a method (with of without parameters) of a given object, the parallel firing of one or more events.

Definition 5.1.3 *Observable Events with Variables.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, $T_{\Sigma,X}$ be the set of terms built over Σ and X. The set of observable events of Spec with variables in X, noted $Event_{Spec,X}$, is the least set recursively defined as follows:

$$
\begin{array}{rl}
t.m \in Event_{Spec,X} & \textit{iff}\; t \in (T_{\Sigma,X})_c\,,\ m_c \in M \\
t.m(t_1, \dots, t_k) \in Event_{Spec,X} & \textit{iff}\; t \in (T_{\Sigma,X})_c\,,\ m_c : s_1, \dots, s_k \in M\,, \\
& \quad t_i \in (T_{\Sigma,X})_{s_i}\ (1 \leq i \leq k) \\
t.\text{create} \in Event_{Spec,X} & \textit{iff}\; t \in (T_{\Sigma,X})_c\,,\ c \in S^C \\
t.\text{destroy} \in Event_{Spec,X} & \textit{iff}\; t \in (T_{\Sigma,X})_c\,,\ c \in S^C \\
e_1 \mathbin{//} \dots \mathbin{//} e_n \in Event_{Spec,X} & \textit{iff}\; e_i \in Event_{Spec,X}.
\end{array}
$$

Remark 5.1.4 *The set $Event_{Spec,X}$ of observable events of Spec with variables in X is actually a subset of $\cup_{c \in S^C} \mathbf{E}_{(T_{\Sigma,X}),M,(T_{\Sigma,X})_c,S^C}$ (see Definition 4.1.17).*

Due to the CO-OPN/2 semantics, static objects are implicitly created at the beginning of the transition system of *Spec*, using new_c and init_c. Thus, if a class c defines a unique static object, o, then the term o_c and the term init_c refers to the same object, i.e., the interpretation function - which maps terms to values in the semantics - affects the same value to o_c and to new_c. More generally, if a class c defines n static objects, the n terms: init_c, $\text{new}_c(\text{init}_c)$, ..., $\text{new}_c(\text{new}_c(\dots(\text{init}_c)))$ ($n-1$ times new_c) refers to the n static objects. In order to simplify the notation of static object identifiers in observable events and because they are non-deterministically created, the use of o_c names is allowed in observable events.

The creation of dynamic objects occurs either in an observable way, if the dynamic object is created by the user of the specification (context); or in an unobservable way, if the dynamic object is created as part of a synchronous request. Thus, it is impossible for the specifier to know exactly how many objects have been created, and thus which term to use to refer to an existing object, or to create a new object. For this reason, we allow the use of variables for the object identifiers and parameter terms, these variables are *not* variables defined in the specification, they are extra variables used exclusively to build observable events. Therefore, the set of variables X is meant to be *different* from the set of variables of the specification.

Some observable events of the CO-OPN/2 specification $Spec_0$ are given by the following example:

Example 5.1.5 *Observable Events of* $Spec_0$.
Let $Spec_0$ *be the CO-OPN/2 specification of Example 5.1.2, and*

$$X_0 = (\{pack_1, pack_2\})_{\text{packaging}}$$

be a set of variables. The following events are observable events of $Spec_0$ *with variables in* X_0*, i.e., events of* $Event_{Spec_0, X_0}$.

- $\text{the-heap}\,.\text{create}, \text{the-heap}\,.\,\text{put}(pack_1), \text{the-heap}\,.\,\text{get}(pack_1)$
- $\text{the-heap}\,.\,\text{get}(\text{new}(pack_1)),\ \text{new}(\text{the-heap}).\,\text{put}(pack_1)$
- $\text{the-heap}\,.\,\text{put}(pack_1)\ //\ pack_1.\,\text{fill}(\text{P})$
- $pack_1.\text{create}, pack_2.\text{create}, pack_1.\,\text{fill}(\text{P})$.

A HML formula can be the *true* formula, **T**; a sequence of observable events, embedded in the <.> (*next*) operator, ending with **T**; the conjunction $\wedge$ of two HML formulae, or the negation $\neg$ of a HML formula. The **T** formula is an empty formula used as a terminator for every HML formula.

Definition 5.1.6 *HML Formulae.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ *be a* S*-disjointly-sorted set of variables,* $Event_{Spec,X}$ *be the set of observable events of Spec with variables in* X*. The set of HML formulae that can be expressed on Spec and* X*, noted* $\text{PROP}_{Spec,X}$*, is the least set such that:*

$$\begin{aligned} \mathbf{T} &\in \text{PROP}_{Spec,X} & \\ \neg\phi &\in \text{PROP}_{Spec,X} & \text{if } \phi \in \text{PROP}_{Spec,X} \\ \phi \wedge \psi &\in \text{PROP}_{Spec,X} & \text{if } \phi, \psi \in \text{PROP}_{Spec,X} \\ <e>\ \phi &\in \text{PROP}_{Spec,X} & \text{if } \phi \in \text{PROP}_{Spec,X}, e \in Event_{Spec,X}. \end{aligned}$$

Remark 5.1.7 *The choice of HML as the logic for expressing formulae on CO-OPN/2 specifications enables to express formulae on services that the CO-OPN/2 specification is able to furnish, however it is not possible to express properties about the internal behaviour or the state of a CO-OPN/2 specification.*

Remark 5.1.8 *Variables appearing in the formulae are not quantified; they are implicitly existentially quantified, as we will see later in the semantics of HML formulae.*

Notation 5.1.9 *We denote by* SPEC *the set of all CO-OPN/2 specifications, and* **X** *the class of all sets of variables.*

We denote by PROP *the set of all HML formulae that can be expressed on CO-OPN/2 specifications and sets of variables:* $\text{PROP} = \bigcup_{Spec \in \text{SPEC}, X \in \mathbf{X}} \text{PROP}_{Spec,X}$.

Example below gives some HML formulae on $Spec_0$ and X_0. We will see in the sequel in which cases some of these formulae are actually satisfied by the transition system of $Spec_0$, and which of them can be part of a contract.

Example 5.1.10 *HML Formulae of* PROP_{Spec_0,X_0}.
Let $Spec_0$ *be the CO-OPN/2 specification of Example 5.1.2, and* X_0 *be the set of variables of Example 5.1.5. The following formulae are HML formulae on* $Spec_0$ *and* X_0.

$$\begin{aligned}
\phi_1 = \ & <pack_1.\text{create}> \\
& <\text{the-heap . put}(pack_1)><\text{the-heap . get}(pack_1)> \mathbf{T} \\
\phi_2 = & \neg(<pack_1.\text{create}> \\
& <\text{the-heap . get}(pack_1)> \mathbf{T}) \\
\phi_3 = \ & <pack_1.\text{create}><pack_1.\text{ fill(P)}> \mathbf{T} \\
\phi_4 = \ & <pack_1.\text{create}><pack_2.\text{create}> \\
& <\text{the-heap . put}(pack_1)><\text{the-heap . put}(pack_2)> \\
& (<\text{the-heap . get}(pack_1)><\text{the-heap . get}(pack_2)> \wedge \\
& <\text{the-heap . get}(pack_2)><\text{the-heap . get}(pack_1)>)\mathbf{T} \\
\phi_5 = \ & <\text{the-heap .create}><pack_1.\text{create}><pack_1.\text{ fill(P)}> \mathbf{T} \\
\phi_6 = \ & <pack_2.\text{create}> \\
& <\text{the-heap . put}(pack_2) \ // \ pack_2.\text{ fill(P)}> \mathbf{T}.
\end{aligned}$$

Formula ϕ_1 means that a chocolate packaging can be created, and that it can first be inserted into the heap and then removed. Formula ϕ_2 states that it is not possible to remove a packaging from the heap, if it has not been previously inserted into the heap. Formula ϕ_3 states that after having created a packaging, it is possible to fill it with a praline. Formula ϕ_4 gives the essential feature of a heap: two packagings can be removed from the heap in the same order as they have been inserted, but also in the reverse order. Formula ϕ_5 is the same as ϕ_3 except that it requires to observe the creation of the static object `the-heap`. Formula ϕ_6 states that a packaging can be created and that it is possible to simultaneously insert the packaging into the heap, and fill the packaging with a praline.

Remark 5.1.11 *A formula like* $<\text{the-heap .create}><pack_1.\text{ fill(P)}> \mathbf{T}$ *could be a HML formula, satisfied by the transition system of a CO-OPN/2 specification, even though the event* $<pack_1.\text{ fill(P)}>$ *is observed without the event* $<pack_1.\text{ create}>$ *is previously observed. Indeed, due to the CO-OPN/2 semantics, it is possible (1) to create instances in an unobserved way, i.e., their creation is not visible in the transition system, and (2) to call methods of these instances in an observed way.*

The set of events of a HML formula is simply given by the set of all observable events appearing in the formula.

Definition 5.1.12 *Events of a HML Formula.*
Let $\phi \in$ PROP be a HML formula. The set of events of ϕ, noted $Event_\phi$ is the least set recursively defined as follows:

$$\begin{aligned}
\phi = \mathbf{T} &\Rightarrow Event_\phi = \varnothing \\
\phi = \neg\psi &\Rightarrow Event_\phi = Event_\psi \\
\phi = \phi_1 \wedge \phi_2 &\Rightarrow Event_\phi = Event_{\phi_1} \cup Event_{\phi_2} \\
\phi = \lt e \gt\ \psi &\Rightarrow Event_\phi = \{e\} \cup Event_\psi .
\end{aligned}$$

The following example shows the events of HML formulae of Example 5.1.10.

Example 5.1.13 *The sets of events of ϕ_i $(1 \leq i \leq 6)$ of Example 5.1.10 are the following:*

$$\begin{aligned}
Event_{\phi_1} =& \{pack_1.\text{create}, \text{the-heap} \,.\, \text{put}(pack_1), \text{the-heap} \,.\, \text{get}(pack_1)\} \\
Event_{\phi_2} =& \{pack_1.\text{create}, \text{the-heap} \,.\, \text{get}(pack_1)\} \\
Event_{\phi_3} =& \{pack_1.\text{create}, pack_1 .\, \text{fill(P)}\} \\
Event_{\phi_4} =& \{pack_1.\text{create}, pack_2.\text{create}, \\
& \text{the-heap} \,.\, \text{put}(pack_1), \text{the-heap} \,.\, \text{put}(pack_2), \text{the-heap} \,.\, \text{get}(pack_1), \\
& \text{the-heap} \,.\, \text{get}(pack_2), \text{the-heap} \,.\, \text{get}(pack_2), \text{the-heap} \,.\, \text{get}(pack_1)\} \\
Event_{\phi_5} =& \{\text{ the-heap .create}, pack_1.\text{create}, pack_1 .\, \text{fill(P)}\} \\
Event_{\phi_6} =& \{pack_2.\text{create}, \text{the-heap} \,.\, \text{put}(pack_2) \ // \ pack_2 .\, \text{fill(P)}\}.
\end{aligned}$$

5.1.3 Satisfaction Relation

HML formulae are built with observable events of a given CO-OPN/2 specification, which are made of *syntactical* terms. In order to be able to state if a model satisfies or not a HML formula, it is necessary to evaluate the observable events, i.e., to map every observable event to an event that appears in the model.

As observable events contain terms with variables, it is necessary to first give an assignment that maps every variable to a value in the algebra $A = Sem(Pres(Spec))$ (see Proposition 4.2.3). Then, every term can be interpreted and finally, the observable events themselves can be evaluated as semantical events.

Remark 5.1.14 *Assignment,Interpretation of Terms.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted

set of variables, and $A = Sem(Pres(Spec))$ be the semantics of the presentation of Spec[1]*. An assignment from X to A, noted σ, is a S-sorted function $\sigma : X \to A$.*

Given σ an assignment from X to A, the terms of $T_{\Sigma,X}$ can be interpreted by the S-sorted function $\mu^\sigma : T_{\Sigma,X} \to A$ according to Definition 4.2.4.

Remark 5.1.15 *An assignment is not necessarily injective: two different variables (of the same sort) may be mapped to the same value.*

Notation 5.1.16 *We denote by* ASSIGN *the set of all assignments.*

Example 5.1.17 below gives an assignment for the variables X_0 of example 5.1.5.

Example 5.1.17 *Assignment for $Spec_0$.*
Let $Spec_0$ be the CO-OPN/2 specification of Example 5.1.2, and X_0 be the set of variables of Example 5.1.5. Let $A_0 = Sem(Pres(Spec_0))$ be the semantics of the presentation of $Spec_0$. The following assignment $\sigma_0 : X_0 \to A_0$ is an assignment from X_0 to A_0:

$$\begin{aligned}\sigma_0(pack_1) &= \text{init}^{A_0}_{\text{packaging}}\\ \sigma_0(pack_2) &= \text{new}^{A_0}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}}).\end{aligned}$$

In the case of our running example, the example below gives the interpretation of some of its terms.

Example 5.1.18 *Interpretation of Terms of $Spec_0$ and X_0.*
Let σ_0 be the assignment of variables of Example 5.1.17, some terms of $Spec_0$ with variables in X_0 are interpreted in the following way:

$$\begin{aligned}\mu^{\sigma_0}(\text{init}_{\text{packaging}}) &= \text{init}^{A_0}_{\text{packaging}}\\ \mu^{\sigma_0}(pack_1) &= \text{init}^{A_0}_{\text{packaging}}\\ \mu^{\sigma_0}(pack_2) &= \text{new}^{A_0}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}})\\ \mu^{\sigma_0}(\text{new}_{\text{packaging}}(pack_1)) &= \text{new}^{A_0}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}})\\ \mu^{\sigma_0}(\text{init}_{\text{heap}}) &= \text{init}^{A_0}_{\text{heap}}\\ \mu^{\sigma_0}(\text{the-heap}_{\text{heap}}) &= \text{init}^{A_0}_{\text{heap}}.\end{aligned}$$

It is worth noting that the interpretation of $pack_2$ and $\text{new}_{\text{packaging}}(pack_1)$ are the same, and that the interpretation of $\text{init}_{\text{heap}}$ and $\text{the-heap}_{\text{heap}}$ are the same. In the sequel we will note indifferently $\text{init}^{A_0}_{\text{heap}}$ or $\text{the-heap}^{A_0}_{\text{heap}}$.

The evaluation of an observable event of $Spec$ is an event of the CO-OPN/2 step semantics $SSem_A(Spec)$. Given σ an assignment from X to A, the evaluation of observable events $Event_{Spec,X}$ follows from Definition 4.2.15.

[1] A is the initial model, see Definition 4.2.7

Definition 5.1.19 *Evaluation of Events.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, $A = Sem(Pres(Spec))$ be the semantics of the presentation of Spec, $Event_{Spec,X}$ be the set of observable events of Spec with variables in X, σ be an assignment from X to A, and μ^σ be the interpretation of $T_{\Sigma,X}$ in A according to σ. The evaluation of $Event_{Spec,X}$ according to σ is a function, noted $[[.]]^\sigma : Event_{Spec,X} \to \mathbf{E}_{A,M,\widehat{A},S^C}$, defined as follows:

$$\begin{aligned}
t.m \in Event_{Spec,X} &\Rightarrow [[t.m]]^\sigma = \mu^\sigma(t).m \\
t.m(t_1, \ldots, t_k) \in Event_{Spec,X} &\Rightarrow [[t.m(t_1, \ldots, t_k)]]^\sigma = \mu^\sigma(t).m(\mu^\sigma(t_1), \ldots, \mu^\sigma(t_k)) \\
t.\text{create} \in Event_{Spec,X} &\Rightarrow [[t.\text{create}]]^\sigma = \mu^\sigma(t).\text{create} \\
t.\text{destroy} \in Event_{Spec,X} &\Rightarrow [[t.\text{destroy}]]^\sigma = \mu^\sigma(t).\text{destroy} \\
e_1 \mathbin{//} \ldots \mathbin{//} e_n \in Event_{Spec,X} &\Rightarrow [[e_1 \mathbin{//} \ldots \mathbin{//} e_n]]^\sigma = [[e_1]]^\sigma \mathbin{//} \ldots \mathbin{//} [[e_n]]^\sigma.
\end{aligned}$$

Remark 5.1.20 *The set $[[Event_{Spec,X}]]^\sigma$ is actually a strict subset of $\mathbf{E}_{A,M,\widehat{A},S^C}$, since $[[Event_{Spec,X}]]^\sigma$ contains only events that appear in the transition system $SSem_A(Spec)$ given by the step semantics of Spec.*

Example 5.1.21 below gives the evolution of some observable events of $Spec_0$.

Example 5.1.21 *Evaluation of Events of $Spec_0$ and X_0.*
Let σ_0 be the assignment of variables of Example 5.1.17, the events of Example 5.1.5 are evaluated in the following way:

$$\begin{aligned}
[[\text{the-heap}\,.\text{create}]]^{\sigma_0} &= \text{the-heap}^{A_0}_{\text{heap}}\,.\text{create} \\
[[\text{the-heap}\,.\,\text{put}(pack_1)]]^{\sigma_0} &= \text{the-heap}^{A_0}_{\text{heap}}\,.\,\text{put}(\text{init}^{A_0}_{\text{packaging}}) \\
[[\text{the-heap}\,.\,\text{get}(pack_1)]]^{\sigma_0} &= \text{the-heap}^{A_0}_{\text{heap}}\,.\,\text{get}(\text{init}^{A_0}_{\text{packaging}}) \\
[[\text{the-heap}\,.\,\text{get}(\text{new}(pack_1))]]^{\sigma_0} &= \text{the-heap}^{A_0}_{\text{heap}}\,.\,\text{get}(\text{new}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}})) \\
[[\text{new}(\text{the-heap}).\,\text{put}(pack_1)]]^{\sigma_0} &= \text{new}^{A_0}_{\text{heap}}(\text{the-heap}^{A_0}_{\text{heap}}).\,\text{put}(\text{init}^{A_0}_{\text{packaging}}) \\
[[\text{the-heap}\,.\,\text{put}(pack_1) \mathbin{//} pack_1.\,\text{fill}(\text{P})]]^{\sigma_0} &= \text{the-heap}^{A_0}_{\text{heap}}\,.\,\text{put}(\text{init}^{A_0}_{\text{packaging}}) \mathbin{//} \\
&\quad\ \text{init}^{A_0}_{\text{packaging}}\,.\,\text{fill}(\text{P}^{\text{A}_0}) \\
[[pack_1.\text{create}]]^{\sigma_0} &= \text{init}^{A_0}_{\text{packaging}}\,.\text{create} \\
[[pack_2.\text{create}]]^{\sigma_0} &= \text{new}^{A_0}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}}).\text{create} \\
[[pack_1.\,\text{fill}(\text{P})]]^{\sigma_0} &= \text{init}^{A_0}_{\text{packaging}}\,.\,\text{fill}(\text{P}^{\text{A}_0})\,.
\end{aligned}$$

Notation 5.1.22 *We denote by $\mathbf{TS}$ the set of all transition systems of CO-OPN/2 specifications obtained by the step semantics: $\mathbf{TS} = \bigcup_{Spec \in \text{SPEC}} SSem_A(Spec)$.*

We denote by $\mathbf{St}$ the set of all states of transition systems of CO-OPN/2 specifications: $\mathbf{St} = \bigcup_{Spec \in \text{SPEC}} State_{Spec,A}$.

$SSem_A(Spec)$ is given by Definition 4.2.21, and $State_{Spec,A}$ by Definition 4.2.9.

The following definition states in which cases a HML formula built on $Spec$, a CO-OPN/2 specification, and X a set of variables, is satisfied by a state st of $SSem_A(Spec)$, the step semantics of $Spec$.

Definition 5.1.23 *HML satisfaction relation of HML formulae on Spec and X.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, $\text{PROP}_{Spec,X}$ be the set of HML formulae that can be expressed on Spec and X, $A = Sem(Pres(Spec))$ be the semantics of the presentation of Spec, and σ be an assignment from X to A. Let $SSem_A(Spec)$ be the transition system of Spec according to the step semantics, $st \in State_{Spec,A}$ be a reachable state of $SSem_A(Spec)$, and $\phi, \psi \in \text{PROP}_{Spec,X}$ be HML formulae on Spec and X. The HML satisfaction relation of HML formulae on Spec and X given the assignment σ, noted $\models^{\sigma}_{HML,Spec,X} \subseteq \mathsf{TS} \times \mathsf{St} \times \text{PROP}$, is the least set such that:

(1) $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \mathbf{T}$

(2) $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \neg\phi$ *iff* $SSem_A(Spec), st \not\models^{\sigma}_{HML,Spec,X} \phi$

(3) $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \phi \wedge \psi$ *iff* $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \phi$ *and* $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \psi$

(4) $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} <e>\phi$ *iff* $\exists\,(st, [[e]]^{\sigma}, st') \in SSem_A(Spec)$ *and* $SSem_A(Spec), st' \models^{\sigma}_{HML,Spec,X} \phi$.

Given a reachable state st, i.e., a state such that there exists a sequence of transitions from state $\langle \bot, \varnothing, \bot \rangle$ to state st, the HML satisfaction relation is such that: (1) the HML formula $\mathbf{T}$ is a formula true for every reachable state st of $SSem_A(Spec)$; (2) the negation of a formula is true in a state st, if there is no path, starting from st in $SSem_A(Spec)$, where the formula is true; (3) the conjunction of two HML formulae $\phi \wedge \psi$ is true in a state st, if there is a path starting from st where ϕ is true, and there is a path (the same or another path) starting from st where ψ is true; (4) if a formula begins with an event $<e>$, the formula is true in state st if among all the paths starting from st there is one path starting with the event $[[e]]^{\sigma}$, and such that the new state reached, st', is a state where the end of the HML formula is true.

It is worth noting that:

- a HML formula is satisfied by the step semantics of $Spec$, provided its variables are *existentially* quantified;
- if $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} <e_1><e_2>\phi$ then there exists *a* path, starting from st, that corresponds exactly to ϕ; i.e., $[[e_1]]^{\sigma}$ is observed and is followed immediately by $[[e_2]]^{\sigma}$, which is observed too).

  However, there may be *non observable* events occurring between $[[e_1]]^{\sigma}$ and $[[e_2]]^{\sigma}$;

- even though $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} <e_1><e_2>\ \phi$ holds, there may be *other* paths, starting from st such that e_2 does not follow e_1 (e.g., $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} <e_1><e_2><e_2>\ \phi$ can hold too).

Remark 5.1.24 *We denote* $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \phi$ *instead of* $(SSem_A(Spec), st, \phi) \in \models^{\sigma}_{HML,Spec,X}$.

The definition of $\models^{\sigma}_{HML,Spec,X}$ *is given generally for any transition system, however it is actually* $\models^{\sigma}_{HML,Spec,X} \subseteq \{SSem_A(Spec)\} \times State_{Spec,A} \times \text{PROP}_{Spec,X}$.

Inference rules 4.2.5 allow to compute all valid transitions that the system can perform. Vachon in [59] gives inference rules for computing all invalid transitions.

We extend $\models^{\sigma}_{HML,Spec,X}$ to sets of formulae:

Notation 5.1.25 *Let* $\Phi \subseteq \text{PROP}_{Spec,X}$ *a set of HML formulae on Spec and X, and* $\sigma : X \to A$ *an assignment of variables X. We denote* $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \Phi$ *if* $SSem_A(Spec), st \models^{\sigma}_{HML,Spec,X} \phi$*, for all* $\phi \in \Phi$.

Example 5.1.26 below applies the above definition to our running example.

Example 5.1.26 *Satisfaction of HML formulae on* $Spec_0$ *and* X_0.
Let σ_0 *be the assignment of variables of Example 5.1.17, the HML formulae of Example 5.1.10 are satisfied in the following way by* $SSem_{A_0}(Spec_0)$ *in the initial state and state* st_1 *of Figure 5.2:*

$$SSem_{A_0}(Spec_0), \langle \bot, \varnothing, \bot \rangle \models^{\sigma_0}_{HML,Spec_0,X_0} \{\phi_2, \phi_5\}$$
$$SSem_{A_0}(Spec_0), \langle \bot, \varnothing, \bot \rangle \not\models^{\sigma_0}_{HML,Spec_0,X_0} \{\phi_1, \phi_3, \phi_4, \phi_6\}$$
$$SSem_{A_0}(Spec_0), st_1 \models^{\sigma_0}_{HML,Spec_0,X_0} \{\phi_1, \phi_2, \phi_3, \phi_4\}$$
$$SSem_{A_0}(Spec_0), st_1 \not\models^{\sigma_0}_{HML,Spec_0,X_0} \{\phi_5, \phi_6\}.$$

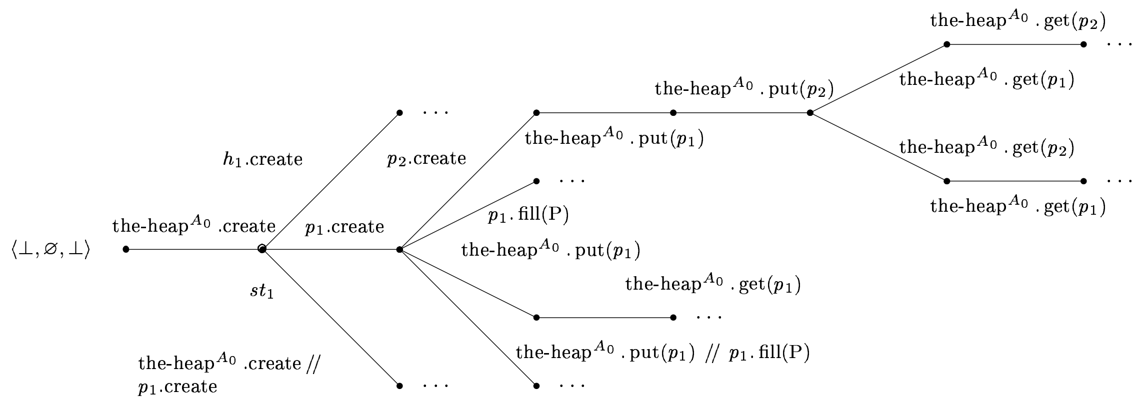
Indeed, according to Figure 5.2 below, which depicts a small view of the sequence of events of the transition system $SSem_{A_0}(Spec_0)$*, the following holds:*

- *Formulae* ϕ_1, ϕ_3, ϕ_4 *and* ϕ_6 *cannot be satisfied in the initial state, since in that state the static object* `the-heap` *is created.*

  However, ϕ_1, ϕ_3, ϕ_4 *are satisfied in the state* st_1*, since there is for each of these formulae a path starting from state* st_1 *and whose beginning is made of events corresponding to the events of the formula evaluated using* σ_0.

  Formula ϕ_6 *cannot be satisfied in state* st_1*. Indeed, formula* ϕ_6 *begins with event* $pack_2.\text{create}$*, and* σ_0 *assigns the value* $\text{new}^{A}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}})$ *to* $pack_2$*. In state* st_1*, it is only possible to create* $\text{init}^{A_0}_{\text{packaging}}$;

- *Formula ϕ_2 is satisfied in both the initial state and state st_1. Indeed, in the initial state it is only possible to create static objects; in state st_1, the static object has been created, but it is not possible to remove a packaging from the heap if it has not been previously inserted;*
- *Formula ϕ_5 can be satisfied only in the initial state since it requires the creation of the static object* `the-heap`*, and this creation is performed only once at the beginning of the transition system.*



Where $h_1 = \text{new}^{A_0}_{\text{heap}}(\text{init}^{A_0}_{\text{heap}})$, $p_1 = \text{init}^{A_0}_{\text{packaging}}$, $p_2 = \text{new}^{A_0}_{\text{packaging}}(\text{init}^{A_0}_{\text{packaging}})$.

Figure 5.2: Sequence of Events of $SSem_{A_0}(Spec_0)$

The HML satisfaction relation is given by the union of all the HML satisfaction relations of HML formulae on $Spec$ and X.

Definition 5.1.27 *HML Satisfaction Relation.*
The HML satisfaction relation, noted $\models_{HML} \subseteq \mathbf{TS} \times \mathbf{St} \times \textsc{Prop}$, is such that:

$$\models_{HML} = \bigcup_{Spec \in \textsc{Spec}, X \in \mathbf{X}} \left(\bigcup_{\sigma : X \to Sem(Pres(Spec)) \in \textsc{Assign}} \models^{\sigma}_{HML,Spec,X} \right).$$

Remark 5.1.28 *According to this definition, a transition system $TS \in \mathbf{TS}$ and a state $s \in \mathbf{St}$ satisfy a HML formula ϕ, $TS, st \models_{HML} \phi$, iff there is a CO-OPN/2 specification Spec, a set of variables X, and an assignment σ of the variables X to $A = Sem(Pres(Spec))$, such that: (1) ϕ is a HML formula on Spec and X, i.e., $\phi \in Event_{Spec,X}$; (2) $TS = SSem_A(Spec)$; (3) s is a reachable state of $SSem_A(Spec)$; and (4) $TS, s \models^{\sigma}_{HML,Spec,X} \phi$.*

Notation 5.1.29 *Models.*
Let Spec be a well-formed CO-OPN/2 specification, according to Definition 4.2.21, it has

only one model: the transition system $SSem_A(Spec)$ *(where* $A = Sem(Pres(Spec))$*). We denote by* $\text{MOD}_{Spec} = \{SSem_A(Spec)\}$ *the set made of this model.*

We denote MOD *the set of all models of CO-OPN/2 specifications:*
$\text{MOD} = \bigcup_{Spec \in \text{SPEC}} \text{MOD}_{Spec}$.

Let Spec be a well-formed CO-OPN/2 specification, we denote Init_{Spec} *the first state of* $SSem_A(Spec)$ *where all the static objects of Spec have been created.*

It is worth noting that $\text{Init}_{Spec} = \langle \bot, \varnothing, \bot \rangle$ when $Spec$ defines no static object.

The satisfaction relation is a relation on models of CO-OPN/2 specifications and HML formulae. A model satisfies a HML formula, if the model and the state Init_{Spec} satisfy the formula, i.e., if there is a path starting from Init_{Spec}, and an assignment of the variables such that the formula can be seen as the beginning of the path.

Definition 5.1.30 *Satisfaction Relation.*
Let $Mod \in \text{MOD}$ *be a model of a CO-OPN/2 specification Spec,* $\phi \in \text{PROP}$ *be a HML formula. The satisfaction relation, noted* $\models \subseteq \text{MOD} \times \text{PROP}$*, is such that:*

$$Mod \models \phi \;\Leftrightarrow\; Mod, \text{Init}_{Spec} \models_{HML} \phi.$$

Due to the definition of $\models_{HML}$, a formula is satisfied by a model, provided there *exists* an assignment of the variables that let the formula be satisfied in the state Init_{Spec} of the model.

Example 5.1.26 shows that some HML formulae are not satisfied for the assignment σ_0 of example 5.1.17. Example 5.1.31 below shows how the HML formulae of example 5.1.10 are satisfied by $SSem_A(Spec_0)$.

Example 5.1.31 *Satisfaction of HML Formulae of* $Spec_0$.
HML formulae of Example 5.1.10 are satisfied by $SSem_A(Spec_0)$ *in the following way:*

$$\begin{array}{ll} SSem_{A_0}(Spec_0) \models \phi_1 & SSem_{A_0}(Spec_0) \models \phi_4 \\ SSem_{A_0}(Spec_0) \models \phi_2 & SSem_{A_0}(Spec_0) \not\models \phi_5 \\ SSem_{A_0}(Spec_0) \models \phi_3 & SSem_{A_0}(Spec_0) \models \phi_6. \end{array}$$

Example 5.1.26 shows that formulae ϕ_1 *to* ϕ_4 *are satisfied by* $SSem_{A_0}(Spec_0)$ *and state* st_1 *(which is exactly* Init_{Spec_0}*), using assignment* σ_0 *of Example 5.1.17. Formula* ϕ_5 *can be satisfied on the initial state only, thus it cannot be satisfied on state* Init_{Spec_0}*. Formula* ϕ_6 *cannot be satisfied using assignment* σ_0*, however it can be satisfied using an assignment* σ_0' *such that* $\sigma_0'(pack_2) = \text{init}^{A_0}_{\text{packaging}}$.

5.2 CO-OPN/2 Refinement

The refinement of CO-OPN/2 specifications is based on contracts as defined in Chapter 3. Given a CO-OPN/2 specifications, a contract is a set of HML formulae, that are satisfied by the transition system of the specification for the same assignment of the variables. A contractual specification is simply a pair given by a specification and a contract. The refine relation is an injective, partial function that is total on elements of the contract, i.e., it is essentially a renaming that maintains the part of the structure of the high-level specification concerned by the contract. The formula refinement is a simple rewriting of the formulae based on the renaming given by the refine relation as well. Finally, two contractual CO-OPN/2 specifications are in a refinement relation if the translated high-level contract is part of the lower-level contract.

This section defines contractual CO-OPN/2 specifications, the refine relation on elements of contractual CO-OPN/2 specifications, the formula refinement univocally defined from the refine relation, and finally the refinement relation on CO-OPN/2 specifications.

5.2.1 Contractual CO-OPN/2 Specifications

A contractual CO-OPN/2 specification is a pair made of a CO-OPN/2 specification and a contract, that is a set of HML properties, i.e., HML formulae satisfied by the model of the specification for the same assignment of the variables. We define first HML properties, then contracts, and finally contractual CO-OPN/2 specifications.

A HML property of a CO-OPN/2 specification $Spec$ is a HML formula, on $Spec$ and a set X of variables, satisfied by the state Init_{Spec} of the step semantics of $Spec$, and for some assignment of the variables.

Definition 5.2.1 *HML Properties.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, $\textsc{Prop}_{Spec,X}$ be the set of HML formulae that can be expressed on Spec and X. A HML property ϕ on Spec with variables in X is a HML formula on Spec and X satisfied by the model of Spec, i.e.,

$$\textsc{Mod}_{Spec} \models \phi.$$

The set of all HML properties of Spec with variables in X, noted $\Phi_{Spec,X}$, is such that:

$$\Phi_{Spec,X} = \{\phi \in \textsc{Prop}_{Spec,X} \mid \textsc{Mod}_{Spec} \models \phi\}.$$

Remark 5.2.2 *Since a well-formed CO-OPN/2 specification Spec has only one model, $SSem_A(Spec)$, a HML formula ϕ on Spec is a HML property of Spec iff*

$$SSem_A(Spec), \mathrm{Init}_{Spec} \models_{HML} \phi.$$

A contract is a set of properties such that the *same* assignment σ is used for the satisfaction relation $\models_{HML}$.

Definition 5.2.3 *Contract of a CO-OPN/2 specification.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s\in S}$ be a S-disjointly-sorted set of variables, and $A = Sem(Pres(Spec))$. A contract on Spec and X, noted Φ, is a set of properties of Spec with variables in X:

$$\Phi \subseteq \Phi_{Spec,X} \ ,$$

such that there is $\sigma : X \to A$, an assignment of the variables, and

$$SSem_A(Spec), \mathrm{Init}_{Spec} \models^{\sigma}_{HML,Spec,X} \Phi.$$

Remark 5.2.4 *Variables of the contract are existentially quantified, but the same assignment of the variables is used for every property of the contract.*

Due to this definition and to the semantics of HML formulae, the set of HML formulae constituting a contract could be replaced by a single HML formula made of the conjunction of all the HML formulae of the contract, without the semantics of the contract being altered. We prefer to keep a set of HML formulae in the contract, in order to stick with the notation of Chapter 3, i.e., a concrete specification refines correctly a more abstract specification if all the translated properties of the abstract contract are part of the concrete contract.

A contract Φ is not necessarily the biggest set of properties satisfied by the initial state of the step semantics of Spec and for the same assignment of variables σ.

A contractual CO-OPN/2 specification is a pair made of a CO-OPN/2 specification and a contract on the specification.

Definition 5.2.5 *Contractual CO-OPN/2 Specifications.*
Let Spec be a well-formed CO-OPN/2 specification, $X = (X_s)_{s\in S}$ be a S-disjointly-sorted set of variables, and $\Phi \subseteq \Phi_{Spec,X}$ be a contract on Spec. A contractual CO-OPN/2 specification, noted CSpec, is a pair:

$$CSpec = \langle Spec, \Phi \rangle.$$

The models of $\langle Spec, \Phi\rangle$ are simply given by the models of *Spec*.

Definition 5.2.6 *Models of a Contractual CO-OPN/2 Specification.*
Let $CSpec = \langle Spec, \Phi\rangle$ be a contractual CO-OPN/2 specification, and MOD_{Spec} be the models of Spec. The set of models of CSpec, noted MOD_{CSpec}, is given by:

$$\mathrm{MOD}_{CSpec} = \mathrm{MOD}_{Spec} (= \{SSem_A(Spec)\}).$$

Notation 5.2.7 *Contractual CO-OPN/2 Specifications.*
We denote CSPEC *the set of all contractual CO-OPN/2 specifications.*

Example 5.2.8 *A Contract for* $Spec_0$*.*
Given $Spec_0$ *of example 5.1.2, formulae* ϕ_1*,* ϕ_2 *and* ϕ_3 *below form a contract* $\Phi_0 = \{\phi_1, \phi_2, \phi_3\}$ *:*

$$\phi_1 = \ <pack_1.\text{create}><\text{the-heap}\,.\,\text{put}(pack_1)><\text{the-heap}\,.\,\text{get}(pack_1)>\ \mathbf{T}$$
$$\phi_2 = \neg(<pack_1.\text{create}><\text{the-heap}\,.\,\text{get}(pack_1)>\ \mathbf{T})$$
$$\phi_3 = \ <pack_1.\text{create}><pack_1.\,\text{fill}(\text{P})>\ \mathbf{T}.$$

As shown in Example 5.1.26, these formulae are actually properties of $Spec_0$ *for the same assignment,* σ_0*, of variables:*

$$SSem_A(Spec_0), \text{Init}_{Spec} \models^{\sigma_0}_{HML, Spec_0, X_0} \Phi_0.$$

Thus, we define the following contractual CO-OPN/2 specification:

$$CSpec_0 = \langle Spec_0, \Phi_0 \rangle.$$

5.2.2 Refine Relation

There are several ways of defining a refine relation on CO-OPN/2, all related of them related to the preservation or not of the structure: (1) ADT and Class modules of a higher-level specification are maintained in their entirety, and the lower-level specification may add some ADT and Class modules; (2) ADT and Class modules of a higher-level specification are partially maintained, i.e., the lower-level specification may add new functions, methods and static objects to existing ADT and Class modules, and may remove existing elements. In addition, new ADT and Class modules can be added. In this case the structure is partially maintained; (3) the ADT and Class modules of a higher-level specification are not maintained, the lower-level specification may split a high-level ADT or Class module over several lower-level ADT of Class modules respectively, provided the functions, methods and static objects of the higher-level specification are related to some function, method or static object of the lower-level specification. In this last case the structure is no longer preserved.

In the framework of CO-OPN/2, we have chosen the second case, i.e., with the help of a renaming, the following holds:

- high-level ADT sorts and Class types whose elements appear in the contract are maintained;
- ADT and Class module interfaces whose elements appear in the contract are partially maintained, i.e., operators and methods appearing in the contract are preserved with the same arity as well as static objects needed in the contract, while operators, methods and static objects that do not appear in the contract may be removed;

- the sub-typing and sub-sorting relations of the higher-level CO-OPN/2 contractual specification are maintained on types and sorts that are maintained;
- the lower-level CO-OPN/2 contractual specification can add new functions to an ADT module, and new methods and static objects to a Class module;
- the lower-level CO-OPN/2 contractual specification can add new ADT and Class modules.

This solution offers a simple translation of the high-level formulae into lower-level ones, since no ambiguity is authorised. In addition, from a theoretical point of view, if the specifier needs to split or fusion ADT and Class modules, this means that the higher-level contractual specification is not correct, since he should have already foreseen this case from the higher-level contractual specification. In addition, this solution does not allow a method to be refined by two methods in parallel (or in sequence, as a non-deterministic choice between two methods or a combination of these cases). The *internal* behaviour of the more concrete method will specify that particular case. However, this solution offers some disadvantages as well, since from a practical point of view, the specifier does not always want to redesign a high-level contractual specification, or, if he uses pre-defined modules, he has not all the necessary modules at his disposal.

Since the purpose of the refine relation is to map syntactical elements of an abstract contractual specification to those of a more concrete contractual specification, we will first define elements of a CO-OPN/2 specification and then give the refine relation on these elements.

An element of a contractual CO-OPN/2 specification is a variable name, an element of the global signature, or an element of the global interface of the CO-OPN/2 specification.

Definition 5.2.9 *Elements of a Contractual CO-OPN/2 Specification.*
Let $CSpec = \langle Spec, \Phi \rangle$ be a contractual CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, $\Phi \subseteq \Phi_{Spec,X}$ a contract on Spec and X. The set of elements of CSpec, noted ELEM_{CSpec}, is such that

$$\text{ELEM}_{CSpec} = S^A \cup S^C \cup F^A \cup F^C \cup M \cup O \cup X.$$

An element of ELEM_{CSpec} is an element of the contract if it is a variable, a function name, a method name or a static object name that appears in a property of the contract.

Definition 5.2.10 *Elements of a Contract.*
Let $CSpec = \langle Spec, \Phi \rangle$ be a contractual CO-OPN/2 specification, and $l \in \text{ELEM}_{CSpec}$, an element of CSpec. The element l belongs to the contract Φ, noted $l \in \Phi$, if $\exists \phi \in \Phi$ and an event $e \in Event_\phi$ such that l belongs to e. An element l belongs to an event e, noted $l \in e$, if one of the following holds:

- $e = t.m$ *and* $l \in t$
- $e = t.m$ *and* $l = m$
- $e = t.m(t_1, \dots, t_k)$ *and* $l \in t$
- $e = t.m(t_1, \dots, t_k)$ *and* $l \in t_i$ *for some* $i \in \{1, \dots, k\}$
- $e = t.m(t_1, \dots, t_k)$ *and* $l = m$
- $e = t.\text{create}$ *and* $l \in t$
- $e = t.\text{destroy}$ *and* $l \in t$
- $e = e_1 \ // \ \dots \ // \ e_n$ *and* $l \in e_i$ *for some* $i \in \{1, \dots, n\}$.

An element l belongs to a term t if it appears in that term, i.e., $l \in t$ if $t = l$, or $t = f(t_1, \dots, t_n)$ and $l = f$, or $l \in t_i$ for some $i \in \{1, \dots, n\}$.

Example 5.2.11 *Elements of $CSpec_0$.*
The elements of the contractual CO-OPN/2 specification $CSpec_0$ of Example 5.2.8 are given by:

$$\begin{aligned}\textsc{Elem}_{CSpec_0} = &\{\text{chocolate}, \text{praline}, \text{truffle}, \text{boolean}, \text{natural}\} \cup \\ &\{\text{heap}, \text{packaging}\} \cup \\ &\{\text{P}, \text{T}, \text{praline-capacity}, \text{truffle-capacity}, \\ &\ \textit{Operations of ADT Naturals}, \textit{Operations of ADT Booleans}\} \cup \\ &\{\text{init}_{\text{heap}}, \text{new}_{\text{heap}}, \text{init}_{\text{packaging}}, \text{new}_{\text{packaging}}\} \cup \\ &\{\text{put}_{\text{heap,packaging}}, \text{get}_{\text{heap,packaging}}, \\ &\ \text{fill}_{\text{packaging,chocolate}}, \text{full-praline}_{\text{packaging}}\} \cup \\ &\{\text{the-heap}_{\text{heap}}\} \cup \\ &\{b, n, pack_1, pack_2\}.\end{aligned}$$

The elements belonging to the contract are:

$$\begin{aligned}&\{\text{P}\} \cup \{\text{put}_{\text{heap,packaging}}, \text{get}_{\text{heap,packaging}}, \text{fill}_{\text{packaging,chocolate}}\} \cup \\ &\{\text{the-heap}_{\text{heap}}\} \cup \{pack_1, pack_2\}.\end{aligned}$$

Indeed, only these elements appear in the contract Φ_0 of Example 5.2.8.

The following definition presents the refine relation on elements of CO-OPN/2 contractual specifications. It is an injective, partial function that maintains the part of the structure of the high-level contractual specification that takes part in the contract.

Definition 5.2.12 *CO-OPN/2 Refine Relation.*
Let $CSpec = \langle Spec, \Phi\rangle$, $CSpec' = \langle Spec', \Phi'\rangle$ *be two contractual CO-OPN/2 specifications. A CO-OPN/2 refine relation on* $CSpec$ *and* $CSpec'$*, noted* λ*, is a relation on elements of* $CSpec$ *and elements of* $CSpec'$*:*

$$\lambda \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec'} \ ,$$

such that : $\lambda = \lambda_{S^A} \cup \lambda_{S^C} \cup \lambda_{F^A} \cup \lambda_{F^C} \cup \lambda_M \cup \lambda_O \cup \lambda_X$*, where:*

$$\begin{array}{ll} \lambda_{S^A} \subseteq S^A \times S^{A'} & \lambda_M \subseteq M \times M' \\ \lambda_{S^C} \subseteq S^C \times S^{C'} & \lambda_O \subseteq O \times O' \\ \lambda_{F^A} \subseteq F^A \times F^{A'} & \lambda_X \subseteq X \times X' \ , \\ \lambda_{F^C} \subseteq F^C \times F^{C'} & \end{array}$$

and

$$\begin{array}{rcl}
(f,f') \in \lambda_{F^A} & \Rightarrow & (f : s_1, \ldots, s_n \to s, f' : s'_1, \ldots, s'_n \to s' \ or \\
& & f : \to s, f' : \to s') \ and \\
& & (s,s'), (s_i, s'_i) \in \lambda_{S^A} \cup \lambda_{S^C} \ (1 \leq i \leq n) \\
(f,f') \in \lambda_{F^C} & \Rightarrow & (f = \mathrm{init}_c, f' = \mathrm{init}_{c'} \ or \\
& & f = \mathrm{new}_c, f' = \mathrm{new}_{c'} \ or \\
& & f = \mathrm{sub}_{c,c_1}, f' = \mathrm{sub}_{c',c'_1} \ or \\
& & f = \mathrm{super}_{c,c_1}, f' = \mathrm{super}_{c',c'_1}) \ and \\
& & (c,c'), (c_1, c'_1) \in \lambda_{S^C} \\
(m,m') \in \lambda_M & \Rightarrow & m_c : s_1, \ldots, s_k, \ m'_{c'} : s'_1, \ldots, s'_k \ and \\
& & (c,c') \in \lambda_{S^C}, (s_i, s'_i) \in \lambda_{S^A} \cup \lambda_{S^C} \ (1 \leq i \leq k) \\
(o_c, o'_{c'}) \in \lambda_O & \Rightarrow & o : c, \ o' : c' \ and \ (c,c') \in \lambda_{S^C} \\
(x,x') \in \lambda_X & \Rightarrow & x \in X_s, x' \in X'_{s'} \ and \ (s,s') \in \lambda_{S^A} \cup \lambda_{S^C} \\
(s,s'), (s_1, s'_1) \in \lambda_{S^A} \cup \lambda_{S^C} \ \wedge \ s \leq s_1 & \Rightarrow & s' \leq' s'_1 \\
(l,l'), (l,l'') \in \lambda & \Rightarrow & l' = l'' \\
(l,l'), (l'',l') \in \lambda & \Rightarrow & l = l'' \\
l \in \Phi & \Rightarrow & \exists l' \in \textsc{Elem}_{CSpec'} \ s.t \ (l,l') \in \lambda.
\end{array}$$

The CO-OPN/2 refine relation relates sorts, types, functions, methods, static objects, and variables of $CSpec$ and sorts, types, functions, methods, static objects and variables of $CSpec'$ respectively. A type (in S^C) cannot be related to a sort (in S^A) and vice-versa a sort cannot be related to a type; a function cannot be related to a method and vice-versa.

The refine relation respects the types and sorts of the methods and functions, i.e., a function f or a method m of $CSpec$ is related to a function f or a method m' of $CSpec'$

respectively, such that the size of the arity of f or m is the same as that of f' or m' respectively, and each type or sort of the arity of f or m is related to the corresponding type or sort of the arity of f' or m' respectively. The refine relation imposes that functions of F^C are related to corresponding functions of $F^{C'}$. For instance, it is not allowed to relate an init_c function with a $\text{new}_{c'}$ function, it can only be related to a $\text{init}_{c'}$ function.

A static object o is related to a static object o' provided the type of o is related to the type of o'. Similarly for the variables, a variable x of type or sort s is related to a variable x' of type or sort s' provided s is related to s'.

The subtyping and the sub-sorting relations of $CSpec$ are preserved, i.e., two sorts of $CSpec$, that are in a sub-sorting or subtyping relationship, are related to two sorts of $CSpec'$, that are also in a sub-sorting relationship.

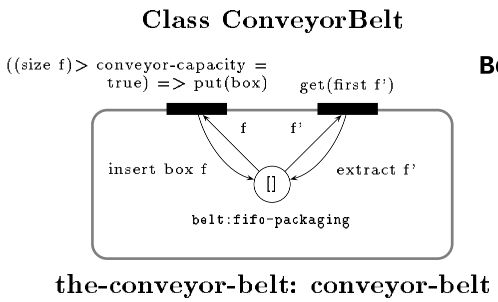
The refine relation is functional, i.e., an element l of $CSpec$ cannot be related to two different elements of $CSpec'$; and it is injective, i.e., two different elements of $CSpec$ cannot be related to the same element of $CSpec'$.

Finally, the refine relation may be partial, but must be total on elements belonging to the contract. If an element of $CSpec$ appears in the contract Φ, then this element must be related to some element of $CSpec'$.

Remark 5.2.13 *CO-OPN/2 Refine Relation is a Refine Relation.*
A CO-OPN/2 refine relation, λ, given in Definition 5.2.12 is actually a refine relation as stated by Definition 3.1.8, since λ is total on elements of the contract.

5.2.3 Running Example

The contractual CO-OPN/2 specification $CSpec_0$, defined in Example 5.2.8, is refined by the contractual CO-OPN/2 specification $CSpec_1 = \langle Spec_1, \Phi_1 \rangle$ defined in Example 5.2.14 below. $Spec_1$ is based on the CO-OPN/2 Class module of Figure 5.3:




```
Class ConveyorBelt;
Interface
  Use    Packaging;
  Type   conveyor-belt;
  Object the-conveyor-belt;
  Methods put _, get _ : packaging;
Body
  Use    FifoPackaging;
  Place  belt _ : fifo-packaging;
  Initial belt [];
  Axioms
    put box ::
      (size f)>conveyor-capacity = true =>
      belt f -> belt (insert box f);
    get (first f') ::
      belt f' -> belt (extract f');
    Where
      f   : fifo-packaging;
      f'  : ne-fifo-packaging;
      box : packaging;
End ConveyorBelt;
```

Figure 5.3: CO-OPN/2 `ConveyorBelt` Class Module

The CO-OPN/2 `ConveyorBelt` Class module is very similar to the CO-OPN/2 `Heap` Class module. They both store and remove `packaging` boxes. The major difference between them is that the $\texttt{get}_{\text{conveyor-belt,packaging}}$ method extracts boxes, from the `belt` place, in the same order as their order of insertion into the place, while method $\texttt{get}_{\text{heap,packaging}}$ has no policy to extract boxes from the `storage` place. The second difference comes from the fact that the `ConveyorBelt` Class module limits the number of the stored boxes to `conveyor-capacity`, while the `Heap` Class module does not limit this number.

$Spec_1$ is defined as the minimal complete CO-OPN/2 specification such that it allows Class module `ConveyorBelt` to be defined, and it allows boxes to be of type `packaging` and of type `deluxe-packaging`. This type is a subtype of `packaging`, defined in the `DeluxePackaging` ADT module. It allows boxes to contain square holes for storing pralines and round holes for storing truffles. Example 5.2.14 below defines $Spec_1$ and $CSpec_1$.

Example 5.2.14 *$Spec_1$, X_1, $CSpec_1$.*
We define the following CO-OPN/2 specification:

$$Spec_1 = \{(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Chocolate}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Capacity}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}},$$
$$(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Packaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{DeluxePackaging}},$$
$$(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{FifoPackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ConveyorBelt}}\}.$$

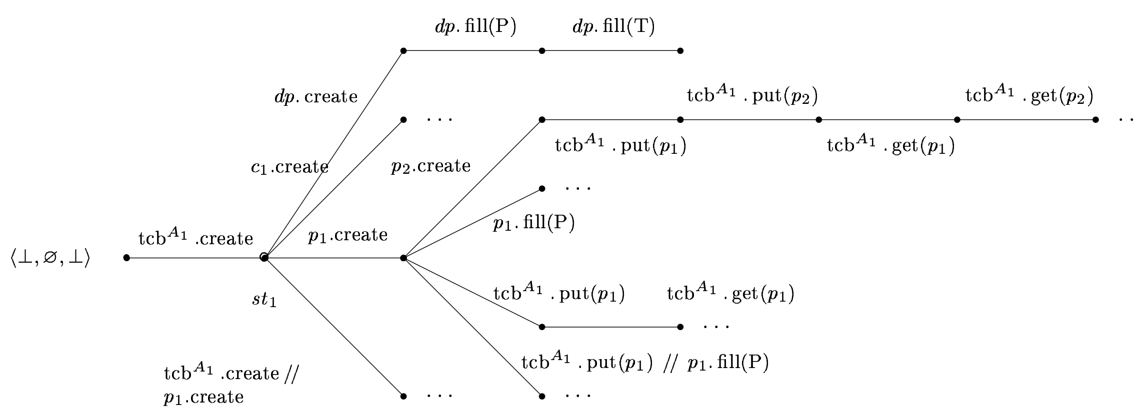
We define the following set of variables:

$$X_1 = \{pack_1, \ldots, pack_{51}\}_{\text{packaging}} \cup \{dpack\}_{\text{deluxe-packaging}},$$

the following contract $\Phi_1 = \{\phi_1^1, \phi_2^1, \phi_3^1, \phi_4^1, \phi_5^1, \phi_6^1\}$ *below:*

$$
\begin{aligned}
\phi_1^1 = \; & \langle pack_1.\text{create}\rangle \\
& \langle\text{the-conveyor-belt . put}(pack_1)\rangle\langle\text{the-conveyor-belt . get}(pack_1)\rangle\ \mathbf{T} \\
\phi_2^1 = \; & \neg(\langle pack_1.\text{create}\rangle\langle\text{the-conveyor-belt . get}(pack_1)\rangle\ \mathbf{T}) \\
\phi_3^1 = \; & \langle pack_1.\text{create}\rangle\langle pack_1.\ \text{fill}_{\text{packaging}}(\text{P})\rangle\ \mathbf{T} \\
\phi_4^1 = \; & \langle pack_1.\text{create}\rangle\langle pack_2.\text{create}\rangle \\
& \langle\text{the-conveyor-belt . put}(pack_1)\rangle\langle\text{the-conveyor-belt . put}(pack_2)\rangle \\
& (\langle\text{the-conveyor-belt . get}(pack_1)\rangle\langle\text{the-conveyor-belt . get}(pack_2)\rangle \wedge \\
& \neg(\langle\text{the-conveyor-belt . get}(pack_2)\rangle\langle\text{the-conveyor-belt . get}(pack_1)\rangle))\mathbf{T} \\
\phi_5^1 = \; & \langle pack_1.\text{create}\rangle \ldots \langle pack_{50}.\text{create}\rangle\langle pack_{51}.\text{create}\rangle \\
& \langle\text{the-conveyor-belt . put}(pack_1)\rangle \ldots \langle\text{the-conveyor-belt . put}(pack_{50})\rangle \\
& \neg(\langle\text{the-conveyor-belt . put}(pack_{51})\rangle)\mathbf{T} \\
\phi_6^1 = \; & \langle dpack.\text{create}\rangle\langle dpack.\ \text{fill}_{\text{deluxe-packaging}}(\text{T})\rangle\langle dpack.\ \text{fill}_{\text{deluxe-packaging}}(\text{P})\rangle\ \mathbf{T}.
\end{aligned}
$$

The contract Φ_1 *of* $CSpec_1$ *is actually a contract. Figure 5.4 below gives a restricted view of the sequence of events of the transition system* $SSem_{A_1}(Spec_1)$ $(A_1 = Sem(Pres(Spec_1)))$.



Where tcb = the-conveyor-belt^{A_1}, $c_1 = \text{new}^{A_1}_{\text{conveyor-belt}}(\text{init}^{A_1}_{\text{conveyor-belt}})$, $p_1 = \text{init}^{A_1}_{\text{packaging}}$, $p_2 = \text{new}^{A_1}_{\text{packaging}}(\text{init}^{A_1}_{\text{packaging}})$, $dp = \text{init}^{A_1}_{\text{deluxe-packaging}}$.

Figure 5.4: Sequence of Events of $SSem_{A_1}(Spec_1)$

Formulae ϕ_1^1, ϕ_2^1, ϕ_3^1 *are similar to formulae* ϕ_1, ϕ_2, *and* ϕ_3 *discussed for* $Spec_0$. *Formula* ϕ_4^1 *describes the essential feature of the* `conveyor-belt` *type: boxes are removed in the same order as their insertion order. It is not possible to remove first* $pack_2$ *and then* $pack_1$ *if* $pack_1$ *has been inserted before* $pack_2$. *Formulae* ϕ_5^1 *describes the second feature of the* `conveyor-belt` *type: the number of boxes that can be stored is limited to the* `conveyor-capactity`, *which is* 50. *Formula* ϕ_6^1 *is similar to* ϕ_3^1, *except that it requires that a praline* `P` *and a truffle* `T` *can be inserted in a* `deluxe-packaging` *box.*

Finally, we define $CSpec_1$ *as*

$$CSpec_1 = \langle Spec_1, \Phi_1 \rangle.$$

Appendix A gives the complete textual CO-OPN/2 specification of $Spec_1$ as well as its CO-OPN/2 abstract specification, global signature, and global interface.

Example 5.2.15 below gives a CO-OPN/2 refine relation on $CSpec_0$ and $CSpec_1$.

Example 5.2.15 *CO-OPN/2 Refine Relation.*
Given $CSpec_0$, $CSpec_1$ *of Examples 5.2.8 and 5.2.14 respectively, we define a CO-OPN/2 refine relation* $\lambda \subseteq \text{ELEM}_{CSpec_0} \times \text{ELEM}_{CSpec_1}$ *on* $CSpec_0$ *and* $CSpec_1$ *in the following way:*

$$\begin{aligned}
\lambda_{0_{SA}} =& \{(\text{chocolate}, \text{chocolate}), (\text{praline}, \text{praline})\} \\
\lambda_{0_{SC}} =& \{(\text{packaging}, \text{packaging}), (\text{heap}, \text{conveyor-belt})\} \\
\lambda_{0_{FA}} =& \{(\text{P}_{\text{praline}}, \text{P}_{\text{praline}})\} \\
\lambda_{0_{FC}} =& \{(\text{new}_{\text{heap}}, \text{new}_{\text{conveyor-belt}}), (\text{init}_{\text{heap}}, \text{init}_{\text{conveyor-belt}}), \\
& (\text{new}_{\text{packaging}}, \text{new}_{\text{packaging}}), (\text{init}_{\text{packaging}}, \text{init}_{\text{packaging}})\} \\
\lambda_{0_M} =& \{(\text{put}_{\text{heap,packaging}}, \text{put}_{\text{conveyor-belt,packaging}}), \\
& (\text{get}_{\text{heap,packaging}}, \text{get}_{\text{conveyor-belt,packaging}}), \\
& (\text{fill}_{\text{packaging,chocolate}}, \text{fill}_{\text{packaging,chocolate}})\} \\
\lambda_{0_O} =& \{(\text{the-heap}, \text{the-conveyor-belt})\} \\
\lambda_{0_X} =& \{(pack_1, pack_1)\}.
\end{aligned}$$

Since the `ConveyorBelt` *Class module is meant to replace the* `Heap` *Class module, the refine relation relates the* `heap` *type and the* `conveyor-belt` *type,* `put`, `get` *of* `heap` *to* `put`, `get` *of* `conveyor-belt` *respectively, and static object* `the-heap` *to static object* `the-conveyor-belt`. *It is the identity for the other elements.* λ_0 *given here is minimal, it is not defined for elements which are not in the contract, e.g., operator T or method* `full-praline`.

5.2.4 Formula Refinement

The refine relation enables to map elements of a high-level CO-OPN/2 contractual specification with elements of a lower-level one. Based on this mapping it is possible to transform every property of the high-level contract into a HML formula of the lower-level specification. In order to transform high-level HML formulae into lower-level HML formulae, it is necessary to transform first the high-level terms, constituting the observable events, into lower-level terms, second the high-level observable events into lower-level ones, and finally the HML formulae themselves.

The term refinement consists of replacing the term name by the corresponding term name given by λ, the refine relation.

Definition 5.2.16 *Term Refinement.*
Let $CSpec = \langle Spec, \Phi \rangle$ and $CSpec' = \langle Spec', \Phi' \rangle$ be two contractual CO-OPN/2 specifications. Let $T_{\Sigma,X}$ be the set of terms of Spec with variables in X, and $T_{\Sigma',X'}$ be the set of terms of Spec′ with variables in X'. Let $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ be a CO-OPN/2 refine relation on elements of CSpec and elements of CSpec′. The term refinement induced by λ, noted $\Lambda_T : T_{\Sigma,X} \to T_{\Sigma',X'}$, is a partial function, such that:

$$\Lambda_T(x) = \begin{cases} x' & \text{if } (x, x') \in \lambda, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_T(f) = \begin{cases} f' & \text{if } f : \to s \text{ and } (f, f') \in \lambda, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_T(f(t_1, \ldots, t_n)) = \begin{cases} f'(\Lambda_T(t_1), \ldots, \Lambda_T(t_n)), & \text{if } (f, f') \in \lambda, \text{ and} \\ & \Lambda_T(t_i) \text{ is defined } (1 \le i \le n), \\ \text{undefined otherwise.} \end{cases}$$

Remark 5.2.17 *Λ_T is defined on terms belonging to the contract Φ of Spec, since λ is total on elements of the contract, thus λ is total on terms of the contract.*

The following example illustrates the term refinement for our running example:

Example 5.2.18 *Refinement of Terms of $CSpec_0$.*
Let $CSpec_0$, $CSpec_1$ be the contractual CO-OPN/2 specifications of Examples 5.2.8 and 5.2.14 respectively. Let λ_0 be the CO-OPN/2 refine relation of Example 5.2.15. Some of the terms of Example 5.1.5 are refined in the following way:

$$\begin{aligned} \Lambda_T(\text{init}_{\text{packaging}}) &= \text{init}_{\text{packaging}} \\ \Lambda_T(\text{init}_{\text{heap}}) &= \text{init}_{\text{conveyor-belt}} \\ \Lambda_T(\text{the-heap}) &= \text{the-conveyor-belt} \\ \Lambda_T(pack_1) &= pack_1. \end{aligned}$$

The event refinement consists of replacing every term appearing in a high-level observable event by its refinement, and of replacing every high-level method appearing in the high-level event by the low-level method related to the high-level method through the CO-OPN/2 refine relation. Default constructor `create` and default destructor `destroy` are related to the default constructor and the default destructor respectively.

Definition 5.2.19 *Event Refinement.*
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$ be two contractual CO-OPN/2 specifications, $Event_{Spec,X}$ be the set of observable events of Spec and X, $Event_{Spec',X'}$ the set of observable events of Spec′ and X' respectively, and $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ a

CO-OPN/2 refine relation on $CSpec$ and $CSpec'$. The event refinement induced by λ, noted $\Lambda_{Event} : Event_{Spec,X} \to Event_{Spec',X'}$, is a partial function such that:

$$\Lambda_{Event}(t.m) = \begin{cases} \Lambda_T(t).m' & \text{if } \Lambda_T(t) \text{ is defined and } (m,m') \in \lambda, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_{Event}(t.m(t_1,\ldots,t_k)) = \begin{cases} \Lambda_T(t).m'(\Lambda_T(t_1),\ldots,\Lambda_T(t_k)) & \text{if } \Lambda_T(t), \Lambda_T(t_i)\ (1 \le i \le n) \text{ is} \\ & \text{defined and } (m,m') \in \lambda, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_{Event}(t.\mathrm{create}) = \begin{cases} \Lambda_T(t).\mathrm{create} & \text{if } \Lambda_T(t) \text{ is defined,} \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_{Event}(t.\mathrm{destroy}) = \begin{cases} \Lambda_T(t).\mathrm{destroy} & \text{if } \Lambda_T(t) \text{ is defined,} \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda_{Event}(e_1 \ /\!/ \ \ldots \ /\!/ \ e_n) = \begin{cases} \Lambda_{Event}(e_1) \ /\!/ \ \ldots \ /\!/ \ \Lambda_{Event}(e_n) & \text{if } \Lambda_{Event}(e_i) \text{ is defined} \\ & (1 \le i \le n), \\ \text{undefined otherwise.} \end{cases}$$

Remark 5.2.20 *Λ_{Event} is defined on events belonging to the contract Φ of $Spec$, since λ is total on elements belonging to the contract, thus on terms, and events.*

The following example illustrates the event refinement for our running example:

Example 5.2.21 *Refinement of Events of $CSpec_0$.*
Let $CSpec_0$, $CSpec_1$ be the contractual CO-OPN/2 specifications of Examples 5.2.8 and 5.2.14 respectively. Let λ_0 be the CO-OPN/2 refine relation of Example 5.2.15. Some of the events of Example 5.1.5 are refined in the following way:

$$\begin{aligned} \Lambda_{Event}(pack_1.\mathrm{create}) &= pack_1.\mathrm{create} \\ \Lambda_{Event}(\text{the-heap} . \mathrm{put}(pack_1)) &= \text{the-conveyor-belt} . \mathrm{put}(pack_1) \\ \Lambda_{Event}(\text{the-heap} . \mathrm{get}(pack_1)) &= \text{the-conveyor-belt} . \mathrm{get}(pack_1) \\ \Lambda_{Event}(pack_1 . \mathrm{fill(P)}) &= pack_1 . \mathrm{fill(P)} \,. \end{aligned}$$

The formula refinement is based on the event refinement: the refinement of a high-level HML formula consists of replacing every event appearing in the formula by its refinement.

Definition 5.2.22 *CO-OPN/2 Formula Refinement.*
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$ be two contractual CO-OPN/2 specifications, and $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ be a CO-OPN/2 refine relation on elements of

CSpec and elements of CSpec′. The CO-OPN/2 formula refinement induced by λ*, noted* $\Lambda : \textsc{Prop}_{Spec,X} \to \textsc{Prop}_{Spec',X'}$*, is a partial function such that:*

$$\Lambda(\mathbf{T}) = \mathbf{T}$$

$$\Lambda(\neg\phi) = \begin{cases} \neg\Lambda(\phi) & \textit{if } \Lambda(\phi) \textit{ is defined,} \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda(\phi \wedge \psi) = \begin{cases} \Lambda(\phi) \wedge \Lambda(\psi) & \textit{if } \Lambda(\phi) \textit{ and } \Lambda(\psi) \textit{ are defined,} \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda(<e>\ \phi) = \begin{cases} <\Lambda_{Event}(e)>\ \Lambda(\phi) & \textit{if } \Lambda_{Event}(e) \textit{ and } \Lambda(\phi) \textit{ are defined,} \\ \textit{undefined otherwise.} \end{cases}$$

Proposition 5.2.1 Λ *is a total function on formulae of the contract.*
Let $CSpec = \langle Spec, \Phi \rangle$*,* $CSpec' = \langle Spec', \Phi' \rangle$ *be two contractual CO-OPN/2 specifications, and* $\lambda \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CSpec'}$ *be a CO-OPN/2 refine relation on elements of CSpec and elements of CSpec′. The CO-OPN/2 formula refinement induced by* λ*,* $\Lambda : \textsc{Prop}_{Spec,X} \to \textsc{Prop}_{Spec',X'}$*, is a total function on the formulae of the contract* Φ *of CSpec.*

Proof.
The CO-OPN/2 refine relation λ is total on elements of the contract, thus Λ_T is total on terms of the contract, and consequently Λ_{Event} is total on $\cup_{\phi \in \Phi} Event_\phi$, the events of the properties of the contract of $CSpec$. This induces Λ to be total on the formulae of the contract. ■

Proposition 5.2.2 *CO-OPN/2 Formula Refinement is actually a Formula Refinement.* Λ *as given by Definition 5.2.22 is a formula refinement as stated in Definition 3.1.12.*

Proof.
We must show the three following points:

- Λ is total on formulae of the contract.
  Indeed, Proposition 5.2.1 above shows this fact;

- if $\lambda = Id_{\textsc{Elem}_{CSpec}}$, i.e., the refine relation is the identity, then Λ must be the identity on formulae.
  Indeed, if $\lambda = Id_{\textsc{Elem}_{CSpec}}$, then the term refinement Λ_T is the identity on terms, and the event refinement Λ_{Event} is the identity on events. Thus, Λ is the identity on formulae.

- if $\lambda'' = \lambda; \lambda'$ is a refine relation, then $\Lambda'' = \Lambda' \circ \Lambda$.
  Indeed, the term refinement and the event refinement are simply functional renamings, thus $\Lambda''_T = \Lambda'_T \circ \Lambda_T$, and $\Lambda''_{Event} = \Lambda'_{Event} \circ \Lambda_{Event}$, and consequently $\Lambda'' = \Lambda' \circ \Lambda$.

■

Notation 5.2.23 *We use the same notation as the one defined in Chapter 3,* $\Lambda(\Phi) = \{\Lambda(\phi) \mid \phi \in \Phi\}$.

Example 5.2.24 *Formula Refinement of the Contract of* $CSpec_0$.
Let $CSpec_0$, $CSpec_1$ *be the contractual CO-OPN/2 specifications of Examples 5.2.8 and 5.2.14 respectively. Let* λ_0 *be the CO-OPN/2 refine relation of Example 5.2.15. The contract* $\Phi_0 = \{\phi_1, \phi_2, \phi_3\}$ *is refined in the following way:*

$$\begin{aligned}\Lambda_0(\phi_1) = \ & <pack_1.\text{create}><\text{the-conveyor-belt}\,.\,\text{put}(pack_1)> \\ & <\text{the-conveyor-belt}\,.\,\text{get}(pack_1)>\ \mathbf{T} \\ \Lambda_0(\phi_2) = & \neg(<pack_1.\text{create}><\text{the-conveyor-belt}\,.\,\text{get}(pack_1)>\ \mathbf{T}) \\ \Lambda_0(\phi_3) = \ & <pack_1.\text{create}><pack_1.\,\text{fill(P)}>\ \mathbf{T}.\end{aligned}$$

5.2.5 Refinement Relation

A lower-level CO-OPN/2 contractual specification correctly refines a higher-level CO-OPN/2 contractual specification via a CO-OPN/2 refine relation λ, if the refinement of the high-level contract, obtained with the CO-OPN/2 formula refinement Λ induced by λ, is a subset of the lower-level contract.

Definition 5.2.25 *Refinement of Contractual CO-OPN/2 Specifications via* λ.
Let $CSpec = \langle Spec, \Phi\rangle$, $CSpec' = \langle Spec', \Phi'\rangle$ *be two contractual CO-OPN/2 specifications,* $\lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ *be a CO-OPN/2 refine relation on* $CSpec$ *and* $CSpec'$, *and* Λ *be the CO-OPN/2 formula refinement induced by* λ. $\langle Spec', \Phi'\rangle$ *is a refinement of* $\langle Spec, \Phi\rangle$ *via* λ, *noted* $\langle Spec, \Phi\rangle \sqsubseteq^{\lambda} \langle Spec', \Phi'\rangle$, *iff:*

$$\Lambda(\Phi) \subseteq \Phi'.$$

More generally, two contractual CO-OPN/2 specifications are in a refinement relation if there exists a CO-OPN/2 refine relation λ on them, such that one of them is correctly refined by the other via λ.

Definition 5.2.26 *Refinement Relation.*
The refinement relation, noted $\sqsubseteq$, *is a relation on contractual CO-OPN/2 specifications:*

$$\sqsubseteq\ \subseteq \text{CSPEC} \times \text{CSPEC}\ ,$$

such that for every $CSpec = \langle Spec, \Phi\rangle$, $CSpec' = \langle Spec', \Phi'\rangle \in \text{CSPEC}$, $\langle Spec, \Phi\rangle \sqsubseteq \langle Spec', \Phi'\rangle$ *iff*

$\exists \lambda \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CSpec'}$ *a CO-OPN/2 refine relation on* $CSpec$ *and* $CSpec'$, *s.t.* $\langle Spec, \Phi\rangle\ \sqsubseteq^{\lambda} \langle Spec', \Phi'\rangle$.

Proposition 5.2.3 *The refinement relation* $\sqsubseteq \subseteq \text{CSpec} \times \text{CSpec}$ *is a pre-order.*

Proof.
Follows from proposition 3.1.1. ■

Example 5.2.27 *$CSpec_1$ refines $CSpec_0$.*
Let $CSpec_0$, $CSpec_1$ be the contractual CO-OPN/2 specifications of Examples 5.2.8 and 5.2.14 respectively. Let λ_0 be the CO-OPN/2 refine relation of Example 5.2.15. The following holds:

$$\Lambda_0(\Phi_0) \subseteq \Phi_1.$$

Indeed, Example 5.2.24 shows that $\Lambda_0(\phi_1) = \phi_1^1$, $\Lambda_0(\phi_2) = \phi_2^1$, and $\Lambda_0(\phi_3) = \phi_3^1$. Formulae $\phi_4^1, \phi_5^1, \phi_6^1$ are additional formulae required by $CSpec_1$ for further refinement steps. In addition, these formulae have no equivalent in $Spec_0$, they are specific to $Spec_1$.

If we consider now another contract $\Phi_0' = \Phi_0 \cup \{\phi_4\}$ instead of Φ_0, we obtain a new contractual CO-OPN/2 specification, $CSpec_0' = \langle Spec_0, \Phi_0' \rangle$. Given this new contract, $CSpec_1$ above does *not* refine $CSpec_0'$, as shown in the following example.

Example 5.2.28 *$CSpec_1$ does not refine $CSpec_0'$.*
Let $\Phi_0' = \Phi_0 \cup \{\phi_4\}$, $CSpec_0' = \langle Spec_0, \Phi_0' \rangle$, and $CSpec_1$ be the contractual CO-OPN/2 specification of Example 5.2.14. Let λ_0 be the CO-OPN/2 refine relation of Example 5.2.15, $\lambda_0' = \lambda_0 \cup \{(pack_2, pack_2)\}$, and Λ_0' be the formula refinement univocally defined from λ_0'. The following holds:

$$\Lambda_0'(\Phi_0') \nsubseteq \Phi_1.$$

Indeed,

$$\begin{aligned}\Lambda_0'(\phi_4) = {} & \text{<}pack_1.\text{create>}\text{<}pack_2.\text{create>} \\ & \text{<the-conveyor-belt . put(}pack_1\text{)><the-conveyor-belt . put(}pack_2\text{)>} \\ & (\text{<the-conveyor-belt . get(}pack_1\text{)><the-conveyor-belt . get(}pack_2\text{)>} \wedge \\ & \text{<the-conveyor-belt . get(}pack_2\text{)><the-conveyor-belt . get(}pack_1\text{)>})\mathbf{T}.\end{aligned}$$

We can easily see that $\Lambda_0'(\phi_4) \neq \phi_4^1$, and consequently $\Lambda_0'(\phi_4) \notin \Phi_1$, thus $CSpec_1$ does not refine $CSpec_0$.

The particularity of the behaviour of every instance of the `conveyor-belt` type is that it acts as a FIFO buffer. For this reason, it is not able to extract $pack_2$ before $pack_1$, if $pack_1$ has been stored before $pack_2$. Thus $\Lambda_0'(\phi_4)$ is not a HML property of $Spec_1$ and cannot be part of any contract on $Spec_1$.

Remark 5.2.29 *Biberstein [14] shows that the `heap` type and the `conveyor-belt` type are not subtypes, since they are not bisimular. Formulae ϕ_4 and ϕ_4^1 show this fact.*

It is interesting to note that, although these types are not bisimular, their corresponding Class modules can refine each other; it all depends on the contracts.

5.3 Compositional CO-OPN/2 Refinement

As discussed in Section 3.4, there are two ways of defining compositional specifications: hierarchical specifications and parameterised specifications. The refinement of hierarchical specifications needs only the refinement of complete specifications[2] to be defined. The refinement of parameterised specifications needs as well the refinement of incomplete specifications to be defined. Since, the refinement of incomplete CO-OPN/2 specifications is not defined, and since CO-OPN/2 specifications are naturally hierarchic (no cycles), we define *hierarchical* compositional operators on contractual CO-OPN/2 specifications. The CO-OPN/2 compositional refinement is then defined as the replacement of every high-level component by a lower-level component that refines it.

This section defines compositional contractual CO-OPN/2 specifications, the refinement of compositional contractual CO-OPN/2 specifications, and shows that this refinement is actually compositional.

5.3.1 Compositional Contractual CO-OPN/2 Specifications

A hierarchical compositional operator adds to a set of complete specifications, some CO-OPN/2 ADT and Class modules. The added part considered by itself is an incomplete CO-OPN/2 specification; the set of complete specifications together with the added modules form a complete specification.

We define first incomplete contractual specifications, and second the CO-OPN/2 hierarchical operator.

An incomplete CO-OPN/2 specification is, like a CO-OPN/2 complete specification, a set of ADT modules and a set of Class modules. The only difference is that the ADT or Class modules forming the incomplete specification may use elements that are not defined in these modules.

Definition 5.3.1 *Incomplete CO-OPN/2 Specification.*
An incomplete CO-OPN/2 specification denoted, $\Delta Spec$, is a set of ADT modules and a set of Class modules, i.e.,

$$\Delta Spec = \big\{(Md^{\mathsf{A}})_i \mid 1 \le i \le n\big\} \cup \big\{(Md^{\mathsf{C}})_j \mid 1 \le j \le m\big\}.$$

Definition 4.1.8 (global signature, global interface) can be applied to complete as well as to incomplete CO-OPN/2 specifications. Thus, an incomplete CO-OPN/2 specification has a global signature and a global interface. It is worth noting that the global signature, and the global interface of an incomplete CO-OPN/2 specification, are incomplete too, i.e., they contain only elements of the incomplete CO-OPN/2 specification. Notation 5.1.1

[2]a specification is complete when it uses elements locally defined.

is extended to incomplete CO-OPN/2 specifications, as well as Definition 4.1.12 (terms), Definition 5.1.3 (observable events) and Definition 5.1.6 (HML formulae). Again, it is worth noting that a HML formula on an incomplete CO-OPN/2 specification contains only terms or events that are terms or events of the incomplete CO-OPN/2 specification.

An incomplete contractual CO-OPN/2 specification is a pair made of an incomplete CO-OPN/2 specification and a set of HML formulae.

Definition 5.3.2 *Incomplete Contractual CO-OPN/2 Specification.*
Let $\Delta Spec$ be an incomplete CO-OPN/2 specification, $X = (X_s)_{s \in S}$ be a S-disjointly-sorted set of variables, and $\Delta\Phi \subseteq \textrm{PROP}_{\Delta Spec, X}$ be a set of HML formulae on $\Delta Spec$. An incomplete contractual CO-OPN/2 specification, noted $\Delta CSpec$, is a pair:

$$\Delta CSpec = \langle \Delta Spec, \Delta\Phi \rangle.$$

The contracts of contractual CO-OPN/2 specifications are satisfied by the model of the specification part. It is different for incomplete contractual CO-OPN/2 specifications, the contract part is only a set of HML formulae and not a set of HML properties, since there is no model attached to an incomplete specification. In addition, these HML formulae are expressed exclusively on the incomplete specification.

A k-ary hierarchical compositional operator on contractual CO-OPN/2 specifications is a partial function that builds, from a set of *complete* contractual CO-OPN/2 specifications and an *incomplete* contractual CO-OPN/2 specification, a new *complete* contractual CO-OPN/2 specification. This new complete contractual CO-OPN/2 specification is obtained by the union of the complete and the incomplete contractual CO-OPN/2 specifications.

Definition 5.3.3 *CO-OPN/2 Hierarchical Operator.*
Let $\Delta CSpec = \langle \Delta Spec, \Delta\Phi \rangle$ be an incomplete contractual CO-OPN/2 specification. Let $CSpec_i = \langle Spec_i, \Phi_i \rangle$ $(1 \leq i \leq k)$ be k well-formed CO-OPN/2 contractual specifications. A k-ary CO-OPN/2 hierarchical operator based on $\Delta CSpec$ is a partial function, noted $f_{\Delta CSpec} : \textrm{CSPEC}^k \to \textrm{CSPEC}$, such that:

$$f_{\Delta CSpec}(CSpec_1, \ldots, CSpec_k) = \begin{cases} CSpec = \langle Spec, \Phi \rangle, \textit{ such that:} \\ \quad Spec = \bigcup_{i \in \{1,\ldots,k\}} Spec_i \bigcup \Delta Spec \quad \textit{and} \\ \quad \Phi = \bigcup_{i \in \{1,\ldots,k\}} \Phi_i \bigcup \Delta\Phi \quad \textit{and} \\ \quad \langle Spec, \Phi \rangle \textit{ is a complete contractual} \\ \quad \textit{CO-OPN/2 specification,} \\ \textit{undefined otherwise.} \end{cases}$$

There are several cases where $f_{\Delta CSpec}$ can be undefined:

- *Spec* is incomplete, i.e., the modules of $\Delta Spec$ need elements that are not defined in $\bigcup_{i \in \{1,\ldots,k\}} Spec_i$;

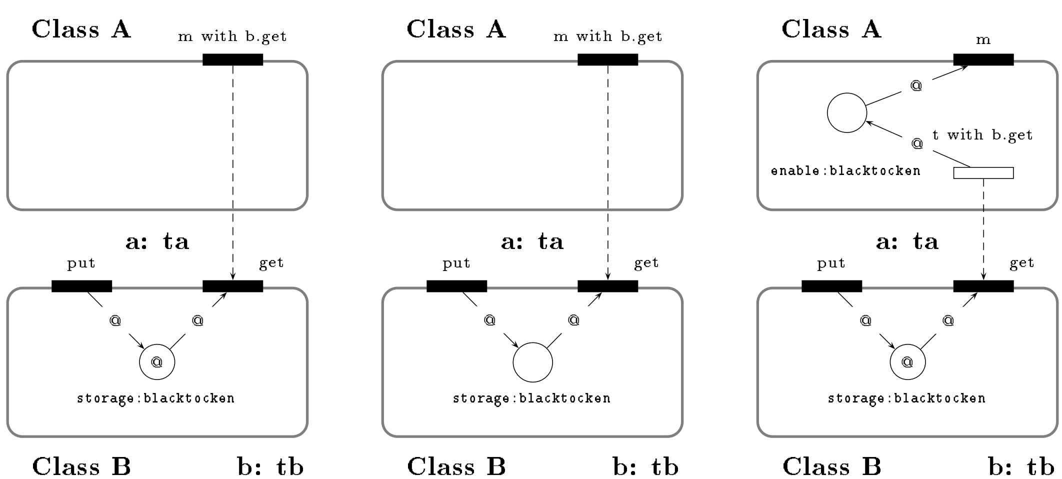
- *Spec* is complete but not well-formed, i.e., the modules of *Spec* have cycles;
- *Spec* is well-formed but the model of *Spec* does not satisfy Φ. Two cases occur: (1) the contract $\Delta\Phi$ on the incomplete contractual CO-OPN/2 specification is not satisfied by the model of the complete specification *Spec*; this is the case when one or more formulae of $\Delta\Phi$ depend, in an unobservable way, on the underlying $Spec_i$, that are such that they do not ensure $\Delta\Phi$; (2) there is some i, such that the contract Φ_i of the contractual CO-OPN/2 specification $CSpec_i$ that is satisfied by the model of $Spec_i$, is *not* satisfied by the model of *Spec*. This last case is due to the fact that instances of modules of $\Delta Spec$ make use of instances of modules of $Spec_i$ in a way that some properties of Φ_i are violated.

Example 5.3.4 below shows three cases of compositional contractual CO-OPN/2 specification. A first case where the compositional contractual CO-OPN/2 specification is defined, and two cases where it is not. These two cases correspond to (1) and (2) above.

Example 5.3.4 *Compositional Contractual CO-OPN/2 Specifications.*
We consider an incomplete contractual specification $\Delta CSpec = \langle\{(Md^{\mathsf{C}})_{\mathrm{A}}\}, \Delta\Phi\rangle$, *with* $\Delta\Phi = \{\text{<a . m>}\ \mathbf{T}\}$. *We consider as well a complete contractual CO-OPN/2 specification* $CSpec_1 = \langle\{(Md^{\mathsf{A}})_{\mathrm{BlackTockens}}, (Md^{\mathsf{C}})_{\mathrm{B}}\}, \Phi_1\rangle$, *where* $\Phi_1 = \{\text{<b . put><b . get>}\ \mathbf{T}\}$. *ADT module* `BlackTockens` *define the* `blacktocken` *type and generator @.*

Figure 5.5 shows three possible cases for Class `A`*, defining static object* `a` *and type* `ta`*, and Class* `B`*, defining static object* `b` *and type* `tb`*. In all these cases, if it is defined,* $f_{\Delta CSpec}(CSpec_1)$ *should be equal to* $\langle Spec, \Phi\rangle$ *where:*

$$Spec = \langle\{(Md^{\mathsf{A}})_{\mathrm{BlackTockens}}, (Md^{\mathsf{C}})_{\mathrm{A}}, (Md^{\mathsf{C}})_{\mathrm{B}}\}$$
$$\Phi = \{\ \text{<b . put><b . get>}\ \mathbf{T}, \text{<a . m>}\ \mathbf{T}\}.$$



(a) Defined (b) Undefined (c) Undefined

Figure 5.5: Compositional Contractual CO-OPN/2 Specifications

Spec is a well-formed CO-OPN/2 specification in the three cases, however $\langle Spec, \Phi\rangle$ *is a contractual CO-OPN/2 specification in the first case only, i.e.,* $f_{\Delta CSpec}(CSpec_1)$ *is defined in the first case only. Indeed:*

- *Case (a): the two HML formulae of* Φ *are actually satisfied by the model of Spec.*
- *Case (b): the HML formula* <a . m> **T** *is not satisfied by the model of Spec. Indeed, method* get *of static object* b *cannot fire without method* `put` *having fired previously (place* `storage` *being empty). Thus, method* `m` *cannot fire on state* Init_{Spec} *(i.e., immediately after static objects* `a` *and* `b` *have been created).*
- *Case (c): the HML formula* <b . put><b . get> **T** *is not a HML property of Spec. Indeed, transition* `t` *of static object* `a` *fires as soon as method* `get` *is firable. For this reason, the firing of method* `get` *always occurs in an unobservable way, and consequently the event* `b.get` *cannot be an event of the transition system of Spec.*

In the rest of this chapter, we use as synonyms the terms *complete CO-OPN/2 specification* and *CO-OPN/2 specification*, as well as the terms *complete contractual CO-OPN/2 specification* and *contractual CO-OPN/2 specification.*

5.3.2 Compositional Refinement

The CO-OPN/2 compositional refinement consists of replacing every complete component of a high-level compositional contractual CO-OPN/2 specification by a complete component that refines it; and by replacing the incomplete component by an incomplete component that syntactically refines it, i.e., the translated high-level incomplete contract is part of the lower-level incomplete contract.

First we define the syntactic refinement of incomplete CO-OPN/2 contractual specifications, and show then that replacing every (complete and incomplete) component of a high-level compositional contractual CO-OPN/2 specification, by a component that refines it, leads to a lower-level compositional contractual CO-OPN/2 specification that refines the high-level one.

We extend trivially Definition 5.2.9 (element of a contractual specification), Definition 5.2.12 (CO-OPN/2 refine relation), and Definition 5.2.22 (CO-OPN/2 formula refinement) to incomplete specifications. Thus, we can define the refinement of incomplete contractual CO-OPN/2 specifications in a similar way to that of complete contractual CO-OPN/2 specification.

Definition 5.3.5 *Syntactic Refinement of Incomplete Contractual CO-OPN/2 Specification.*
Let $\Delta CSpec = \langle \Delta Spec, \Delta\Phi\rangle$*, and* $\Delta CSpec' = \langle \Delta Spec', \Delta\Phi'\rangle$ *be two incomplete contractual CO-OPN/2 specifications. Let* λ^{Δ} *be a refine relation on elements of* $\Delta CSpec$ *and*

$\Delta CSpec'$, and Λ^Δ the corresponding formula refinement. $\Delta CSpec'$ syntactically refines $\Delta CSpec$, noted $\Delta CSpec \sqsubseteq^\Delta \Delta CSpec'$ iff:

$$\Lambda^\Delta(\Delta\Phi) \subseteq \Delta\Phi'.$$

Remark 5.3.6 *It is important to note that even though we note in a similar way the refinement of complete contractual CO-OPN/2 specifications and the refinement of incomplete contractual CO-OPN/2 specifications, the former is semantically correct, while the latter is only a syntactical verification, but does not infer anything about the satisfaction / non-satisfaction of the formulae of the contract.*

Theorem 5.3.1 *CO-OPN/2 Compositional Refinement.*
Let $\Delta CSpec = \langle \Delta Spec, \Delta\Phi\rangle$, and $\Delta CSpec' = \langle \Delta Spec', \Delta\Phi'\rangle$ be two incomplete contractual CO-OPN/2 specifications. Let $f_{\Delta CSpec}$, $f_{\Delta CSpec'}$ be k-ary CO-OPN/2 hierarchical operators based on $\Delta CSpec$ and $\Delta CSpec'$ respectively. Let $CSpec_i = \langle Spec_i, \Phi_i\rangle$, $(1 \leq i \leq k)$ be k disjoint contractual CO-OPN/2 specifications and $CSpec'_i = \langle Spec'_i, \Phi'_i\rangle$, $(1 \leq i \leq k)$ be k disjoint contractual CO-OPN/2 specifications such that:
$CSpec = \langle Spec, \Phi\rangle = f_{\Delta CSpec}(\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_k, \Phi_k\rangle)$ and
$CSpec' = \langle Spec', \Phi'\rangle = f_{\Delta CSpec'}(\langle Spec'_1, \Phi'_1\rangle, \ldots, \langle Spec'_k, \Phi'_k\rangle)$ are defined. The following holds:

$$\begin{aligned}&\Delta CSpec \sqsubseteq^\Delta \Delta CSpec' \text{ and } \langle Spec_i, \Phi_i\rangle \sqsubseteq \langle Spec'_i, \Phi'_i\rangle, (1 \leq i \leq k) \quad \Rightarrow \\ &f_{\Delta CSpec}(\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_k, \Phi_k\rangle) \sqsubseteq f_{\Delta CSpec'}(\langle Spec'_1, \Phi'_1\rangle, \ldots, \langle Spec'_k, \Phi'_k\rangle).\end{aligned}$$

Proof.
We must prove that there exists $\lambda : \textsc{Elem}_{CSpec} \to \textsc{Elem}_{CSpec'}$, a refine relation, such that $\Lambda(\Phi) \subseteq \Phi'$.

We have that:

$$\textsc{Elem}_{CSpec} = \bigcup_{i\in\{1,\ldots,k\}} \textsc{Elem}_{CSpec_i} \bigcup \textsc{Elem}_{\Delta CSpec} \text{ and}$$
$$\textsc{Elem}_{CSpec'} = \bigcup_{i\in\{1,\ldots,k\}} \textsc{Elem}_{CSpec'_i} \bigcup \textsc{Elem}_{\Delta CSpec'}.$$

In addition, we have that:

$$\begin{aligned}\Delta CSpec \sqsubseteq^\Delta \Delta CSpec' &\Rightarrow \exists\lambda^\Delta : \textsc{Elem}_{\Delta CSpec} \to \textsc{Elem}_{\Delta CSpec'} \text{ s.t. } \Lambda^\Delta(\Delta\Phi) \subseteq \Delta\Phi' \\ \langle Spec_i, \Phi_i\rangle \sqsubseteq \langle Spec'_i, \Phi'_i\rangle &\Rightarrow \exists\lambda_i \text{ s.t. } \Lambda_i(\Phi_i) \subseteq \Phi'_i,\ (1 \leq i \leq k).\end{aligned}$$

Thus, we construct the CO-OPN/2 refine relation $\lambda : \textsc{Elem}_{CSpec} \to \textsc{Elem}_{CSpec'}$ in the following way:

$$\lambda(e) = \begin{cases} \lambda_i(e), & \text{if } e \in \textsc{Elem}_{CSpec_i}, \\ \lambda^\Delta(e), & \text{if } e \in \textsc{Elem}_{\Delta CSpec}, \\ \text{undefined otherwise.} \end{cases}$$

λ is actually a CO-OPN/2 refine relation. Indeed, first, λ^{Δ}, λ_i $(1 \leq i \leq k)$ are CO-OPN/2 refine relations, thus λ is total on the contract; second, $CSpec_i$ $(1 \leq i \leq k)$ are all disjoint, and $CSpec'_i$ $(1 \leq i \leq k)$ are all disjoint, thus λ is functional and injective.

The formula refinement is given by:

$$\Lambda(\phi) = \begin{cases} \Lambda_i(\phi), & \text{if } \phi \in \Phi_i, \\ \Lambda^{\Delta}(\phi), & \text{if } \phi \in \Delta\Phi, \\ \text{undefined otherwise.} \end{cases}$$

Thus, $\Lambda(\Phi_i) \subseteq \Phi'_i$, $(1 \leq i \leq k)$, and $\Lambda(\Delta\Phi) \subseteq \Delta\Phi'$. Finally, we have trivially $\Lambda(\Phi) \subseteq \Phi'$.
■

Remark 5.3.7 *The condition "$f_{\Delta CSpec'}(\langle Spec'_1, \Phi'_1\rangle, \ldots, \langle Spec'_k, \Phi'_k\rangle)$ is defined" is essential in the Theorem above. Indeed, replacing every $CSpec_i$ by any $CSpec'_i$, such that $CSpec_i \sqsubseteq CSpec'_i$ is not sufficient to ensure $f_{\Delta CSpec}(\langle Spec_1, \Phi_1\rangle, \ldots, \langle Spec_k, \Phi_k\rangle) \sqsubseteq f_{\Delta CSpec'}(\langle Spec'_1, \Phi'_1\rangle, \ldots, \langle Spec'_k, \Phi'_k\rangle)$, because it is not sufficient to ensure that $f_{\Delta CSpec'}(\langle Spec'_1, \Phi'_1\rangle, \ldots, \langle Spec'_k, \Phi'_k\rangle)$ is defined. As shown in Example 5.3.4, it may happen that HML formulae of $\Delta\Phi'$ are not satisfied by $CSpec'$, because the underlying $Spec'_i$ are such that $\Delta\Phi'$ cannot be satisfied. Similarly, HML formulae of Φ'_i may be not satisfied by $CSpec'$ because of $\Delta Spec'$. Thus, even though the contract $\Delta\Phi$ is syntactically preserved and the contracts Φ_i $(1 \leq i \leq n)$ are semantically preserved when we consider the separate refinements $CSpec_i \sqsubseteq CSpec'_i$, it may happen that these contracts are no longer preserved when we consider the whole composition.*

The following example illustrates the case, where, even though every complete contractual CO-OPN/2 specification $CSpec_i$ is replaced by a complete contractual CO-OPN/2 specification $CSpec'_i$ that correctly refines it, and an incomplete contractual CO-OPN/2 specification $\Delta CSpec$ is replaced by an incomplete contractual CO-OPN/2 specification that syntactically preserves its contract, the compositional refinement is incorrect.

Example 5.3.8 *Incorrect Compositional CO-OPN/2 Refinements.*
We consider example 5.3.4 and Figure 5.5. We note the incomplete contractual specification of each case: $\Delta CSpec^{\alpha} = \langle\{(Md^{\mathsf{C}})^{\alpha}_{\mathrm{A}}\}, \Delta\Phi\rangle$, with $\Delta\Phi = \{\text{<a . m>}\ \mathbf{T}\}$ $(\alpha \in \{a, b, c\})$. As well we note the complete underlying specification $CSpec^{\alpha}_1 = \langle\{(Md^{\mathsf{A}})^{\alpha}_{\mathrm{BlackTocken}}, (Md^{\mathsf{C}})^{\alpha}_{\mathrm{B}}\}, \Phi_1\rangle$, where $\Phi_1 = \{\text{<b . put><b . get>}\ \mathbf{T}\}$. Finally, we note $CSpec^{\alpha} = f_{\Delta CSpec^{\alpha}}(CSpec^{\alpha}_1)$ $(\alpha \in \{a, b, c\})$.

The following holds:

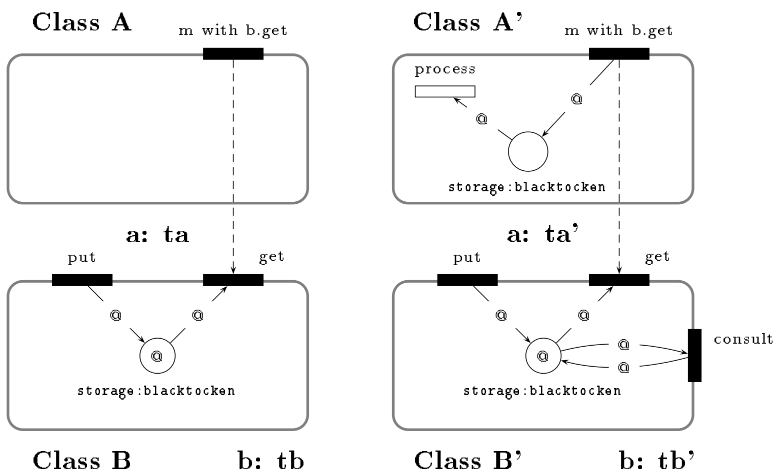
- *$CSpec^a_1 \sqsubseteq CSpec^b_1$ and $\Delta CSpec^a \sqsubseteq^{\Delta} \Delta CSpec^b$ but $CSpec^a \not\sqsubseteq CSpec^b$.*
  The refine relation is the identity. HML formula <b . put><b . get> $\mathbf{T}$ is satisfied by the model of $CSpec^a_1$ and that of $CSpec^b_1$. In addition, HML formula <a . m> $\mathbf{T}$ is a HML formula on $\Delta CSpec^b$. However, this last formula is not satisfied by the model of $CSpec^b$.

- $CSpec_1^a \sqsubseteq CSpec_1^c$ *and* $\Delta CSpec^a \sqsubseteq^{\Delta} \Delta CSpec^c$ *but* $CSpec^a \not\sqsubseteq CSpec^c$. *Formula* <b . put><b . get> **T** *is not satisfied by the model* $CSpec^c$.

Example 5.3.9 shows a case where the compositional refinement is correct.

Example 5.3.9 *Correct Compositional CO-OPN/2 Refinement.*
We consider two incomplete contractual specifications $\Delta CSpec = \langle \{(Md^{\mathsf{C}})_{\mathrm{A}}\}, \Delta\Phi \rangle$, *and* $\Delta CSpec' = \langle \{(Md^{\mathsf{C}})_{\mathrm{A'}}\}, \Delta\Phi \rangle$, *with* $\Delta\Phi = \{$<a . m> $\mathbf{T}\}$; *and two complete contractual CO-OPN/2 specifications* $CSpec_1 = \langle \{(Md^{\mathsf{A}})_{\mathrm{BlackTockens}}, (Md^{\mathsf{C}})_{\mathrm{B}}\}, \Phi_1 \rangle$, *and* $CSpec_1' = \langle \{(Md^{\mathsf{A}})_{\mathrm{BlackTockens}}, (Md^{\mathsf{C}})_{\mathrm{B'}}\}, \Phi_1 \rangle$, *with* $\Phi_1 = \{$<b . put><b . get> $\mathbf{T}\}$.

Left part of Figure 5.6 shows $CSpec = f_{\Delta CSpec}(CSpec_1)$. *The right part shows* $CSpec' = f_{\Delta CSpec'}(CSpec_1')$.



(a) Abstract (b) Concrete

Figure 5.6: Correct Compositional Refinement of CO-OPN/2 Specifications

We have $\Delta CSpec \sqsubseteq^{\Delta} \Delta CSpec'$, *and* $CSpec_1 \sqsubseteq CSpec_1'$, *and since* $CSpec'$ *is defined (formulae of contract* $\Phi_1 \cup \Delta\Phi$ *are satisfied by* $CSpec'$*), thus we have* $CSpec \sqsubseteq CSpec'$.

Chapter 6

CO-OPN/2 Implementation

Chapter 5 applies the theory of refinement, defined in Chapter 3, to the CO-OPN/2 formal specifications language. In a similar way, the current chapter applies the theory of implementation, defined in Chapter 3, to the CO-OPN/2 language and to object-oriented programming languages.

A program is abstractly defined with ADT and Class modules of program, that are very similar to ADT and Class modules of CO-OPN/2 specifications. The HML logic is used for expressing formulae on programs; and the implementation relation differs only slightly from the refinement relation.

First this chapter defines contractual programs. Second, an implement relation, a formula implementation, and an implementation relation on contractual CO-OPN/2 specifications and contractual programs. Third, it presents some compositional results on the implementation of contractual CO-OPN/2 specifications. Examples of this chapter are all related to Java, since implementations using this programming language have been more particularly studied.

6.1 Contractual Programs

Even though non object-oriented programming languages can be used to implement CO-OPN/2 specifications, we present the implementation of CO-OPN/2 specifications by object-oriented programs.

An object-oriented program can be viewed as a CO-OPN/2 specification, except for the body part of Class modules, which is not given by Petri nets elements but by program instructions. Therefore, most definitions related to CO-OPN/2 specifications can be extended to object-oriented programs. Among others, observable events of programs are similar to observable events of CO-OPN/2 specifications. Consequently, HML formulae on programs are defined like HML formulae on CO-OPN/2 specifications, i.e., they are sequences of observable events of programs. A contract on a program is a set of HML formulae on the program, that is satisfied by the execution of the program.

This section defines a running example, i.e., a Java program intended to implement running example of Chapter 5; programs; HML formulae on programs; contracts; and contractual programs.

6.1.1 Running Example

Examples of this chapter use Java classes of Figures 6.1 and 6.2.

```
class JavaHeap extends Vector{
  // Public Static Variables
  public static JavaHeap theheap = new JavaHeap();

  // Inserts a Packaging box at the end of theheap
  public static void insertElement(JavaPackaging box){
    theheap.insertElementAt(box,theheap.size());
  }

  // Removes a Packaging box at a Random Position
  public static JavaPackaging removeElement(){
    JavaPackaging elem;
    int i;
    i = (int) (Math.random() * theheap.size()) % theheap.size();
    elem = (JavaPackaging) theheap.elementAt(i);
    theheap.removeElementAt(i);
    return elem;
  }
}
```

```
class JavaPackaging extends Object {
  // Simulates the Insertion of a Praline into a Packaging box
  public void fill(boolean P){
    if (P == true) {
      System.out.println("One more Praline");}
  }
}
```

Figure 6.1: Java Classes for $CProg_0$

Figure 6.1 shows two Java classes: `JavaHeap` and `JavaPackaging`. The `JavaHeap` class defines a static object called `theheap`. It is used to store and remove objects of type `JavaPackaging` into and from the static object `theheap`. Elements are removed in a random order. Class `JavaHeap` is a sub-class of Class `Vector` which enables to store objects in an ordered structure. It is worth noting that in Java every Class is a sub-class of Class `Object`.

```
class JavaConveyorBelt extends Vector{
  // Public Static Variables
  public static JavaConveyorBelt theconveyorbelt = new JavaConveyorBelt();

  // Inserts Packaging box at the end of theconveyorbelt
  public static void insertElement(JavaPackaging box){
    // Limited size
    if (theconveyorbelt.size() < 51) {
      theconveyorbelt.insertElementAt(box,theconveyorbelt.size());}
  }

  // Removes Packaging box at the beginning of theconveyorbelt
  public static JavaPackaging removeElement(){
    JavaPackaging elem;
    elem = (JavaPackaging) theconveyorbelt.elementAt(0);
    theconveyorbelt.removeElementAt(0);
    return elem;
  }
}
```

```
class JavaDeluxePackaging extends JavaPackaging {
  // Simulates the insertion of a Praline and a Truffle
  // into DeluxePackaging box
  public void fill(boolean P){
    if (P == true) { // Praline
      super.fill(P);}
    else // Truffle
      System.out.println("One more Truffle");
  }
}
```

Figure 6.2: Java Classes for $CProg_1$

Figure 6.2 shows two Java classes: `JavaConveyorBelt` and `JavaDeluxePackaging`. The former is similar to the `JavaHeap` class, except that the static object is called `theconveyorbelt` and that objects of type `JavaPackaging` are removed in a FIFO manner. Since the class `JavaDeluxePackaging` is also defined, objects of type `JavaPackaging` but also of type `JavaDeluxePackaging` can be stored and removed into and from `theconveyorbelt`.

On the basis of these classes, we will show the following:

- the `JavaHeap` and the `JavaPackaging` classes can be used to form a contractual program $CProg_0$ that implements contractual CO-OPN/2 specification $CSpec_0$ of Example 5.2.8. They *cannot* be used to implement $CSpec_1$ of Example 5.2.14;
- the `JavaConveyorBelt`, the `JavaPackaging`, and the `JavaDeluxePackaging` classes can be used to form a contractual program $CProg_1$ that implements both $CSpec_0$ and $CSpec_1$.

Appendix A.3 shows a Java Class `ChocFactory` defining a `main` method using Java classes defined above; and Appendix A.5 shows an example of execution of $CProg_0$ and $CProg_1$.

6.1.2 Programs

Usually, object-oriented programming languages enable to define classes and sub-classes. Instances of sub-classes can be used instead of instances of super-classes. However, sub-classes are not sub-types of the type of their super-class as defined in the framework of CO-OPN/2. Indeed, object-oriented programming languages allow methods defined in a super-class to be newly defined in sub-classes. Thus, the behaviour of instances of the sub-classes can be completely different from that of instances of the super-class, and consequently types defined by sub-classes cannot be sub-types of the type of the super-class.

Object-oriented programming languages allow to define classes, static objects, and public methods, and usually have primitive types. Classes correspond to CO-OPN/2 Class modules, and primitive types correspond to CO-OPN/2 ADT modules. A program is described by a set of classes and a set of primitive types. The exported part of the classes and of the primitive types is very similar to the exported parts of CO-OPN/2 Class modules and CO-OPN/2 ADT modules respectively, and thus can be abstractly described in a similar way.

Moreover, object-oriented programming languages allow instances of classes to be created dynamically. Even though it is hidden for the programmer, a mechanism similar to the one defined in CO-OPN/2 for defining object identifiers (with init_c, $\text{new}_c(\text{init}_c)$, etc.), must be used in order to correctly identify instances dynamically created.

Thus, without loss of generality, we assume the following:

- we have an object-oriented programming language without sub-typing (with sub-classing only).
- every program is complete, i.e., every class or primitive type necessary for the program is defined in the program;
- the name of a class type is the same as the name of the class; this is different from CO-OPN/2 class types which have usually a different name than the Class module where they are defined;
- primitive types are defined with ADT modules defined in a similar way as CO-OPN/2 ADT modules (with an empty sub-sorting relation);
- class interfaces of the program are described with interfaces defined in a similar way as CO-OPN/2 class interfaces;

- Class modules of the programs are *different* from CO-OPN/2 Class modules, however they contain the class interface;
- a program is a set of ADT modules (for the primitive types) and Class modules of programs (different from CO-OPN/2 Class modules);
- every program has a global signature and a global interface defined in a similar way as global signatures and interfaces of CO-OPN/2 specifications (with the sub-typing relationship used for representing the sub-classing relationship).

Given the assumptions above, a program is very similar to a CO-OPN/2 specification, except for the body part of the Class modules, i.e., the Class module without the class interface, which are defined differently from the body part of CO-OPN/2 Class modules.

Notation 6.1.1 *Class Body of Program.*
We denote $Body^{\mathsf{C}}_{Prog}$ *the body part of a Class of program Prog.*

ADT modules of programs are defined as ADT modules of CO-OPN/2 specifications, see Definition 4.1.15.

Notation 6.1.2 *ADT module of Program.*
We denote Md^{A}_{Prog} *an ADT module of a program Prog.*

A Class module of a program is made of two parts: a class interface (see Definition 4.1.5), and a class body.

Definition 6.1.3 *Class module of Program.*
A Class module of a program, noted Md^{C}_{Prog}*, is a pair*

$$Md^{\mathsf{C}}_{Prog} = (\Omega^{\mathsf{C}}_{Prog}, Body^{\mathsf{C}}_{Prog}) \ ,$$

where $\Omega^{\mathsf{C}}_{Prog} = \langle \{c\}, \leq^{\mathsf{C}}, M \rangle$ *is a class interface, and* $Body^{\mathsf{C}}_{Prog}$ *is the body part of the class.*

A program is a set of ADT modules of program and a set of Class modules of program such that the program is complete, i.e., every element used in the program is defined in a ADT or Class module of the program.

Definition 6.1.4 *Program.*
A program, noted Prog, is a set of ADT modules and Class modules of program, i.e.,

$$Prog = \left\{ (Md^{\mathsf{A}}_{Prog})_i \mid 1 \leq i \leq n \right\} \cup \left\{ (Md^{\mathsf{C}}_{Prog})_j \mid 1 \leq j \leq m \right\},$$

such that Prog is complete.

Definitions 4.2.1 (ADT module induced by a Class module), 4.1.8 (global signature and global interface) are extended to programs.

We use the following notations:

Notation 6.1.5 *Programs, Signature, Interface.*
We denote PROG *the set of all programs.*

Let $Prog = \{(Md^{\mathsf{A}}_{Prog})_i \mid 1 \leq i \leq n\} \cup \{(Md^{\mathsf{C}}_{Prog})_j \mid 1 \leq j \leq m\}$ *be a program, and*

$$\Sigma_{Prog} = \left\langle \bigcup_{1 \leq i \leq n} S^{\mathsf{A}}_i \cup \bigcup_{1 \leq j \leq m} \{c_j\},\ \leq,\ \bigcup_{1 \leq i \leq n} F_i \cup \bigcup_{1 \leq j \leq m} F_{\Omega^{\mathsf{C}}_j} \right\rangle.$$

be the global signature of $Prog$*, and*

$$\Omega_{Prog} = \left\langle \bigcup_{1 \leq j \leq m} \{c_j\},\ (\bigcup_{1 \leq j \leq m} \leq^{\mathsf{C}}_j)^*,\ \bigcup_{1 \leq j \leq m} M_j,\ \bigcup_{1 \leq j \leq m} O_j \right\rangle.$$

be the global interface of $Prog$.
We denote:

$$S^A_{Prog} = \bigcup_{1 \leq i \leq n} S^{\mathsf{A}}_i \qquad S^C_{Prog} = \bigcup_{1 \leq j \leq m} \{c_j\} \qquad S_{Prog} = S^A_{Prog} \cup S^C_{Prog}$$

$$F^A_{Prog} = \bigcup_{1 \leq i \leq n} F_i \qquad F^C_{Prog} = \bigcup_{1 \leq j \leq m} F_{\Omega^{\mathsf{C}}_j} \qquad F_{Prog} = F^A_{Prog} \cup F^C$$

$$M_{Prog} = \bigcup_{1 \leq j \leq m} M_j \qquad O_{Prog} = \bigcup_{1 \leq j \leq m} O_j.$$

From the global signature of the program and its modules, it is possible to define the presentation of the program $Pres(Prog)$ in a way similar to the presentation of CO-OPN/2 specifications.

Definition 6.1.6 *Presentation of a Program.*
Let us consider a program $Prog = \{(Md^{\mathsf{A}}_{Prog})_i \mid 1 \leq i \leq n\} \cup \{(Md^{\mathsf{C}}_{Prog})_j \mid 1 \leq j \leq m\}$ *such that* $(Md^{\mathsf{A}}_{Prog})_i = \langle \Sigma^{\mathsf{A}}_i, X_i, \Phi_i \rangle$ *and* $(Md^{\mathsf{C}}_{Prog})_j = \langle \Omega^{\mathsf{C}}_j, (Body^{\mathsf{C}}_{Prog})_j \rangle$*. Let* Σ_{Prog} *be its global signature and* $Md^{\mathsf{A}}_{\Omega^{\mathsf{C}}_j} = \langle \Sigma^{\mathsf{A}}_{\Omega^{\mathsf{C}}_j}, V_{\Omega^{\mathsf{C}}_j}, \Phi_{\Omega^{\mathsf{C}}_j} \rangle$ $(1 \leq j \leq m)$ *be the ADT modules induced by the Class modules of* $Prog$*. The presentation of* $Prog$*, noted* $Pres(Prog)$*, is defined as follows:*

$$Pres(Prog) = \left\langle \Sigma_{Prog},\ \bigcup_{1 \leq i \leq n} X_i \cup \bigcup_{1 \leq j \leq m} V_{\Omega^{\mathsf{C}}_j},\ \bigcup_{1 \leq i \leq n} \Phi_i \cup \bigcup_{1 \leq j \leq m} \Phi_{\Omega^{\mathsf{C}}_j} \right\rangle.$$

Given the presentation, the semantics of $Pres(Prog)$ is given by an algebra B which depends on the target machine where the program is executed. Thus, B may be *different* from the initial semantics of $Pres(Prog)$. This is different from CO-OPN/2 specifications, where the semantics of a the presentation of $Spec$, noted $Sem(Pres(Spec))$, *is* the initial semantics of $Pres(Spec)$.

The transitions of the transition system of $Prog$ are made of states and events. States are built on B, a semantics of the presentation of $Prog$. States depend on the program and the machine where the program is executed. They have a different structure than states of a CO-OPN/2 specification. Events are method calls constructed over the algebra B, and the methods of the global interface of $Prog$. Thus, we can assume that the set of events of the transition system is a subset of $\mathbf{E}_{B,M_{Prog},\hat{B},S^{\mathsf{C}}_{Prog}}$ (see Definition 4.1.17) made of the method calls without the synchronisations.

Notation 6.1.7 *States and Transition System of a Program.*
We denote $State_{Prog,B}$ the set of possible states of the execution of the program Prog with algebra B as the semantics of the presentation of Prog.

We denote $TS_{Prog,B} \subseteq State_{Prog,B} \times \mathbf{E}_{B,M_{Prog},\hat{B},S^{\mathsf{C}}_{Prog}} \times State_{Prog,B}$ the transition system of Prog with algebra B as the semantics of the presentation of Prog.

Example 6.1.8 *Running Example: $Prog_0$ and $Prog_1$.*
We define the following Java programs:

$$\begin{aligned} Prog_0 = \{&(Md^{\mathsf{A}}_{Prog})_{\text{boolean}}, (Md^{\mathsf{A}}_{Prog})_{\text{int}}, (Md^{\mathsf{C}}_{Prog})_{\text{Object}}, (Md^{\mathsf{C}}_{Prog})_{\text{Vector}},\\ &(Md^{\mathsf{C}}_{Prog})_{\text{Random}}, (Md^{\mathsf{C}}_{Prog})_{\text{JavaPackaging}}, (Md^{\mathsf{C}}_{Prog})_{\text{JavaHeap}}\}\\ Prog_1 = \{&(Md^{\mathsf{A}}_{Prog})_{\text{boolean}}, (Md^{\mathsf{A}}_{Prog})_{\text{int}}, (Md^{\mathsf{C}}_{Prog})_{\text{Object}}, (Md^{\mathsf{C}}_{Prog})_{\text{Vector}},\\ &(Md^{\mathsf{C}}_{Prog})_{\text{JavaPackaging}}, (Md^{\mathsf{C}}_{Prog})_{\text{JavaDeluxePackaging}}, (Md^{\mathsf{C}}_{Prog})_{\text{JavaConveyorBelt}}\}. \end{aligned}$$

In order to be complete, a program using Classes `JavaPackaging`, and `JavaHeap`, or `JavaDeluxePackaging` and `JavaConveyorBelt`, must as well use Classes `Object` and `Vector`. Indeed, every Java Class is a sub-class of Class `Object`, and Classes `JavaHeap` and `JavaConveyorBelt` are sub-classes of Class `Vector`. In addition, $Prog_0$ has to use Class `Math` since it needs some of its methods.

Appendix A.3 gives the complete Java sources together with an extra class `ChocFactory` using them. Appendix A.4 gives the global signature and the global interface of $Prog_0$ and $Prog_1$.

6.1.3 HML Formulae on Programs

HML formulae on CO-OPN/2 specifications are defined on the basis of the global interface, the global signature of CO-OPN/2 specifications, and a set of variables. HML formulae on programs are defined as well on the basis of the global interface, the global signature of programs, and a set of variables. Thus, HML formulae on programs are very similar to HML formulae on CO-OPN/2 specifications. The differences between HML formulae on programs and those on CO-OPN/2 specifications are the following:

- since the global signature of CO-OPN/2 specifications define sub-sorting and sub-typing relationships, terms of object identifiers of the form $\mathrm{sub}_{c,c'}$ or $\mathrm{super}_{c,c'}$ are allowed to appear in HML formulae on CO-OPN/2 specifications. Object-oriented programming languages do not define sub-sorting and sub-typing relationships. Therefore, HML formulae on programs do not contain terms built with $\mathrm{sub}_{c,c'}$ or $\mathrm{super}_{c,c'}$ functions;
- every CO-OPN/2 Class module has a default constructor, called create, and a default destructor, called destroy. Programming languages usually have default constructors and destructors for every class, however the default constructor is not called create. We assume that the programming language defines for every class a default constructor with no parameters, whose name is the name of the class, and a default destructor called destroy. In the case of CO-OPN/2 specifications, create and destroy are not part of M_{Prog}. Similarly, for programs, we assume that the default constructor and the destroy method are not part of M_{Prog}.

Terms are defined with the global signature and a set of variables only, Definition 4.1.12 is extended trivially to terms of *Prog* with variables.

Notation 6.1.9 *Terms of Program with Variables.*
Let Prog be a program, Σ_{Prog} be the global signature of Prog and $Y = (Y_s)_{s \in S_{Prog}}$ a S_{Prog}-disjointly-sorted set of variables, we denote $T_{\Sigma_{Prog},Y} = ((T_{\Sigma_{Prog},Y})_s)_{s \in S_{Prog}}$ the set of terms of Prog with variables in Y.

Observable events of programs differ slightly from observable events of CO-OPN/2 specifications since create method is not available by default in a program, a method with the name of the class is available instead.

Definition 6.1.10 *Observable Events of Program with Variables.*
Let Prog be a program, $Y = (Y_s)_{s \in S_{Prog}}$ be a S_{Prog}-disjointly-sorted set of variables, $T_{\Sigma_{Prog},Y}$ be the set of terms built over Σ_{Prog} and Y. The set of observable events of Prog with variables in Y, noted $Event_{Prog,Y}$, is the least set recursively defined as follows:

$$
\begin{array}{rl}
t.m \in Event_{Prog,Y} & \textit{iff } t \in (T_{\Sigma_{Prog},Y})_c \ , \ m_c \in M \\
t.m(t_1,\ldots,t_k) \in Event_{Prog,Y} & \textit{iff } t \in (T_{\Sigma_{Prog},Y})_c \ , \ m_c : s_1,\ldots,s_k \in M \ , \\
& \quad t_i \in (T_{\Sigma_{Prog},Y})_{s_i} \ (1 \le i \le k) \\
t.\,\mathrm{c}() \in Event_{Prog,Y} & \textit{iff } t \in (T_{\Sigma,X})_c \ , \ c \in S^C \\
t.\mathrm{destroy} \in Event_{Prog,Y} & \textit{iff } t \in (T_{\Sigma,X})_c \ , \ c \in S^C \\
e_1 \ // \ \ldots \ // \ e_n \in Event_{Prog,Y} & \textit{iff } e_i \in Event_{Prog,Y}.
\end{array}
$$

HML formulae on programs are defined exactly as HML formulae on CO-OPN/2 specifications except that they are based on observable events of programs, instead of observable events of CO-OPN/2 specifications.

Definition 6.1.11 *HML Formulae of Programs.*
Let Prog be a program, $Y = (Y_s)_{s \in S_{Prog}}$ be a S_{Prog}-disjointly-sorted set of variables, $Event_{Prog,Y}$ be the set of observable events of Prog with variables in Y. The set of HML formulae that can be expressed on Prog and Y, noted $\text{PROP}_{Prog,Y}$, is the least set such that:

$$\begin{aligned}
\mathbf{T} &\in \text{PROP}_{Prog,Y} & \\
\neg\phi &\in \text{PROP}_{Prog,Y} & \text{if } \phi \in \text{PROP}_{Prog,Y} \\
\phi \wedge \psi &\in \text{PROP}_{Prog,Y} & \text{if } \phi, \psi \in \text{PROP}_{Prog,Y} \\
<e>\ \phi &\in \text{PROP}_{Prog,Y} & \text{if } \phi \in \text{PROP}_{Prog,Y}, e \in Event_{Prog,Y}.
\end{aligned}$$

Given $\sigma : Y \to B$ an assignment of the variables to B, a semantics of the presentation of *Prog*, the interpretation of terms of the program, μ^σ, is given by Definition 4.2.4.

The evaluation of observable events of a program is the same as that of observable events of a CO-OPN/2 specification, except for the default constructor method.

Definition 6.1.12 *Evaluation of Events*
Let Prog be a well-formed CO-OPN/2 specification, $Y = (Y_s)_{s \in S_{Prog}}$ be a S_{Prog}-disjointly-sorted set of variables, B be a semantics of the presentation of Prog, $Event_{Prog,Y}$ be the set of observable events of Prog with variables in Y, σ be an assignment from Y to B, and μ^σ be the interpretation of $T_{\Sigma_{Prog},Y}$ in B according to σ. The evaluation of $Event_{Prog,Y}$ according to σ is a function, noted $[[.]]^\sigma : Event_{Prog,Y} \to \mathbf{E}_{B,M_{Prog},\hat{B},S^C_{Prog}}$, defined as follows:

$$\begin{aligned}
t.m \in Event_{Prog,Y} &\Rightarrow [[t.m]]^\sigma = \mu^\sigma(t).m \\
t.m(t_1, \ldots, t_k) \in Event_{Prog,Y} &\Rightarrow [[t.m(t_1, \ldots, t_k)]]^\sigma = \mu^\sigma(t).m(\mu^\sigma(t_1), \ldots, \mu^\sigma(t_k)) \\
t.\,\text{c}() \in Event_{Prog,Y} &\Rightarrow [[t.\,\text{c}()]]^\sigma = \mu^\sigma(t).\,\text{c}() \\
t.\text{destroy} \in Event_{Prog,Y} &\Rightarrow [[t.\text{destroy}]]^\sigma = \mu^\sigma(t).\text{destroy} \\
e_1\ //\ \ldots\ //\ e_n \in Event_{Prog,Y} &\Rightarrow [[e_1\ //\ \ldots\ //\ e_n]]^\sigma = [[e_1]]^\sigma\ //\ \ldots\ //\ [[e_n]]^\sigma.
\end{aligned}$$

We extend below Notations 5.1.9 (HML formulae), 5.1.22 (transition systems, states), and 5.1.29 (models, Init state), in order to let them take programs into account.

Notation 6.1.13 *We denote* PROP *the set of all HML formulae that can be expressed on CO-OPN/2 specifications and sets of variables, and on programs and sets of variables:* $\text{PROP} = \bigcup_{Spec \in \text{SPEC}, X \in \mathbf{X}} \text{PROP}_{Spec,X} \bigcup_{Prog \in \text{PROG}, Y \in \mathbf{X}} \text{PROP}_{Prog,Y}$.

We denote $\mathbf{TS}$ *the set of all transition systems of CO-OPN/2 specifications and of programs:* $\mathbf{TS} = \bigcup_{Spec \in \text{SPEC}} SSem_A(Spec) \bigcup_{Prog \in \text{PROG}} TS_{Prog,B}$.

We denote Mod *the set of all models of CO-OPN/2 specifications and programs:* $\text{MOD} = \bigcup_{Spec \in \text{SPEC}} \text{MOD}_{Spec} \bigcup_{Prog \in \text{PROG}} TS_{Prog,B}$.

We denote $\mathbf{St}$ *the set of all states of transition systems of CO-OPN/2 specifications and programs:* $\mathbf{St} = \bigcup_{Spec \in \text{SPEC}} State_{Spec,A} \bigcup_{Prog \in \text{PROG}} State_{Prog,B}$.

Let Prog be a program, we denote Init_{Prog} *the first state of* $TS_{Prog,B}$ *where all the static objects of Prog have been created.*

Given the evaluation of events of Definition 6.1.12, the satisfaction of HML formulae on programs is similar to that of HML formulae on CO-OPN/2 specifications: a HML formula is satisfied in a given state st, provided there is path in the transition system of the program such that the formula is the beginning of this path.

Definition 6.1.14 *HML satisfaction relation of HML formulae on Prog and Y.*
Let Prog be a program, $Y = (Y_s)_{s \in S_{Prog}}$ *be a* S_{Prog}*-disjointly-sorted set of variables,* $\text{PROP}_{Prog,Y}$ *be the set of HML formulae that can be expressed on Prog and Y, B be a semantics of the presentation of Prog, and* σ *be an assignment from Y to B. Let* $TS_{Prog,B}$ *be the transition system of Prog,* $st \in State_{Prog,B}$ *be a reachable state of* $TS_{Prog,B}$*, and* $\phi, \psi \in \text{PROP}_{Prog,Y}$ *be HML formulae on Prog and Y. The HML satisfaction relation of HML formulae on Prog and Y given the assignment* σ*, noted* $\models^{\sigma}_{HML,Prog,Y} \subseteq \mathbf{TS} \times \mathbf{St} \times \text{PROP}$*, is the least set such that:*

$$
\begin{array}{lll}
TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} \mathbf{T} & & \\
TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} \neg\phi & \textit{iff} & TS_{Prog,B}, st \not\models^{\sigma}_{HML,Prog,Y} \phi \\
TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} \phi \wedge \psi & \textit{iff} & TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} \phi \quad \textit{and} \\
& & TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} \psi \\
TS_{Prog,B}, st \models^{\sigma}_{HML,Prog,Y} <e> \phi & \textit{iff} & \exists\, (st, [[e]]^{\sigma}, st') \in TS_{Prog,B} \quad \textit{and} \\
& & TS_{Prog,B}, st' \models^{\sigma}_{HML,Prog,Y} \phi.
\end{array}
$$

We extend below Definition 5.1.27 to the satisfaction of HML formulae on programs.

Definition 6.1.15 *HML Satisfaction Relation.*
The HML satisfaction relation, noted $\models_{HML} \subseteq \mathbf{TS} \times \mathbf{St} \times \text{PROP}$*, is such that:*

$$
\models_{HML} = \bigcup_{Spec \in \text{SPEC}, X \in \mathbf{X}} \Big(\bigcup_{\sigma : X \to Sem(Pres(Spec)) \in \text{ASSIGN}} \models^{\sigma}_{HML,Spec,X} \Big) \bigcup_{Prog \in \text{PROG}, Y \in \mathbf{X}} \Big(\bigcup_{\sigma : Y \to B \in \text{ASSIGN}} \models^{\sigma}_{HML,Prog,Y} \Big).
$$

Definition 5.1.30 is extended below to the satisfaction relation on models of programs and HML formulae.

Definition 6.1.16 *Satisfaction Relation.*
Let $Mod \in \textsc{Mod}$ *be a model of a CO-OPN/2 specification or a program with* Init *the first state after the creation of all static objects. Let* $\phi \in \textsc{Prop}$ *be a HML formula. The satisfaction relation, noted* $\models \subseteq \textsc{Mod} \times \textsc{Prop}$, *is such that:*

$$Mod \models \phi \;\Leftrightarrow\; Mod, \mathrm{Init} \models_{HML} \phi.$$

If Mod is the step semantics of a CO-OPN/2 specification $Spec$, then $\mathrm{Init} = \mathrm{Init}_{Spec}$; if Mod is the transition system associated to a program $Prog$, then $\mathrm{Init} = \mathrm{Init}_{Prog}$.

6.1.4 Contractual Programs

A HML property of a program $Prog$ is a HML formula such that there exists an assignment of the variables that let the formula be satisfied by the model of $Prog$.

Definition 6.1.17 *HML Properties of Program.*
Let $Prog$ *be a program,* B *be a semantics of* $Pres(Prog)$, $Y = (Y_s)_{s \in S_{Prog}}$ *be a* S_{Prog}*-disjointly-sorted set of variables,* $\textsc{Prop}_{Prog,Y}$ *be the set of HML formulae that can be expressed on* $Prog$ *and* Y. *A HML property* ψ *on* $Prog$ *with variables in* Y *is a HML formula on* $Prog$ *and* Y *satisfied by the transition system of* $Prog$, *i.e.,*

$$TS_{Prog,B} \models \psi.$$

The set of all HML properties of $Prog$ *with variables in* Y, *noted* $\Psi_{Prog,Y}$, *is such that:*

$$\Psi_{Prog,Y} = \{\psi \in \textsc{Prop}_{Prog,Y} \mid TS_{Prog,B} \models \psi\}.$$

Remark 6.1.18 *A HML formula* ψ *on* $Prog$ *is a HML property of* $Prog$ *iff*

$$TS_{Prog,B}, \mathrm{Init}_{Prog} \models_{HML} \psi.$$

As for contractual CO-OPN/2 specifications, a contract on a program is a set of properties of the program such that the *same* assignment σ is used for the satisfaction relation $\models_{HML}$.

Definition 6.1.19 *Contract of a Program.*
Let $Prog$ *be a program,* $Y = (Y_s)_{s \in S_{Prog}}$ *be a* S_{Prog}*-disjointly-sorted set of variables, and* B *a semantics of* $Pres(Prog)$ *the presentation of* $Prog$. *A contract on* $Prog$ *and* Y, *noted* Ψ, *is a set of properties of* $Prog$ *with variables in* Y:

$$\Psi \subseteq \Psi_{Prog,Y} \;,$$

such that there is $\sigma : Y \rightarrow B$*, an assignment of the variables, and*

$$TS_{Prog,B}, \mathrm{Init}_{Prog} \models^{\sigma}_{HML,Prog,Y} \Psi.$$

We can now define a contractual program as a pair: program and contract.

Definition 6.1.20 *Contractual Program.*
Let $Prog$ *be a program,* $Y = (Y_s)_{s \in S_{Prog}}$ *be a* S_{Prog}*-disjointly-sorted set of variables, and* $\Psi \subseteq \Psi_{Prog,Y}$ *be a contract on* $Prog$*. A contractual program, noted* $CProg$*, is a pair:*

$$CProg = \langle Prog, \Psi \rangle.$$

The model of a contractual program is the same as the model of its program part.

Definition 6.1.21 *Model of a Contractual Program.*
Let $CProg = \langle Prog, \Psi \rangle$ *be a contractual program,* B *be the semantics of* $Pres(Prog)$*, and* $TS_{Prog,B}$ *be the model of* $Prog$*. The set of models of* $CProg$*, noted* $\textsc{Mod}_{CProg}$*, is given by:*

$$\textsc{Mod}_{CProg} = \{TS_{Prog,B}\}.$$

Notation 6.1.22 *Contractual Programs.*
We denote CPROG *the set of all contractual programs.*

Example 6.1.23 *A Contract for* $Prog_0$*.*
Given $Prog_0$ *of Example 6.1.8, and the set of variables*

$$Y_0 = \{javapack\}_{\text{JavaPackaging}} ,$$

formulae ψ^0_1*, to* ψ^0_4 *below form a contract* $\Psi_0 = \{\psi^0_1, \psi^0_2, \psi^0_3, \psi^0_4\}$ *:*

$$\begin{aligned}
\psi^0_1 = \ & <javapack.\text{create}><\text{theheap . insertElement}(javapack)> \\
& <\text{theheap . removeElement}(javapack)> \mathbf{T} \\
\psi^0_2 = & \neg(<javapack.\text{create}><\text{theheap . removeElement}(javapack)> \mathbf{T}) \\
\psi^0_3 = \ & <javapack.\text{create}><javapack.\,\text{fill(true)}> \mathbf{T} \\
\psi^0_4 = \ & <\text{theheap . notify}> \mathbf{T}.
\end{aligned}$$

Formula ψ^0_1 *states that a dynamically created instance of* `JavaPackaging` *class can be inserted into and then removed from static object* `theheap`*. Formula* ψ^0_2 *states that it is not possible to remove an instance of* `JavaPackaging` *class from static object* `theheap` *without having previously inserted it. Formula* psi^0_3 *states that it is possible to call method* `fill` *with input parameter* `true` *of an instance of* `JavaPackaging` *class. Finally, formula* ψ^0_4 *states that it is possible to call method* `notify` *of static object* `theheap`*.*

According to the performed executions, these formulae are actually properties of $Prog_0$ *for the assignment* δ_0 *such that,* $\delta_0(javapack) = \mathrm{init}^{B_0}_{\mathrm{JavaPackaging}}$ *(*B_0 *is a semantics of the presentation of* $Prog_0$*), and state* Init_{Prog_0}*. Thus,*

$$TS_{Prog_0,B_0}, \mathrm{Init}_{Prog_0} \models^{\delta_0}_{HML,Prog_0,Y_0} \Psi_0.$$

Thus, we define the following contractual program:

$$CProg_0 = \langle Prog_0, \Psi_0 \rangle.$$

Example 6.1.24 *A Contract for* $Prog_1$*.*
Given $Prog_1$ *of Example 6.1.8, and*

$$Y_1 = \{javapack_1, \dots, javapack_{51}\}_{\mathrm{JavaPackaging}} \cup \{javadeluxepack\}_{\mathrm{JavaDeluxePackaging}} \,,$$

formulae ψ^1_1 *to* ψ^1_7 *below form a contract* $\Psi_1 = \{\psi^1_1, \psi^1_2, \psi^1_3, \psi^1_4, \psi^1_5, \psi^1_6, \psi^1_7\}$*:*

$$\begin{array}{rl}
\psi^1_1 = & <javapack_1.\mathrm{create}><\mathrm{theconveyorbelt.insertElement}(javapack_1)> \\
& <\mathrm{theconveyorbelt.removeElement}(javapack_1)> \mathbf{T} \\
\psi^1_2 = & \neg(<javapack_1.\mathrm{create}> \\
& <\mathrm{theconveyorbelt.removeElement}(javapack_1)> \mathbf{T}) \\
\psi^1_3 = & <javapack_1.\mathrm{create}><javapack_1.\mathrm{fill}_{\mathrm{JavaPackaging}}(\mathrm{true})> \mathbf{T} \\
\psi^1_4 = & <javapack_1.\mathrm{create}><javapack_2.\mathrm{create}> \\
& <\mathrm{theconveyorbelt.insertElement}(javapack_1)> \\
& <\mathrm{theconveyorbelt.insertElement}(javapack_2)> \\
& (<\mathrm{theconveyorbelt.removeElement}(javapack_1)> \\
& <\mathrm{theconveyorbelt.removeElement}(javapack_2)> \wedge \\
& \neg(<\mathrm{theconveyorbelt.removeElement}(javapack_2)> \\
& <\mathrm{theconveyorbelt.removeElement}(javapack_1)>))\mathbf{T} \\
\psi^1_5 = & <javapack_1.\mathrm{create}> \dots <javapack_{50}.\mathrm{create}><javapack_{51}.\mathrm{create}> \\
& <\mathrm{theconveyorbelt.insertElement}(javapack_1)> \dots \\
& <\mathrm{theconveyorbelt.insertElement}(javapack_{50})> \\
& \neg(<\mathrm{theconveyorbelt.insertElement}(javapack_{51})>)\mathbf{T} \\
\psi^1_6 = & <javadeluxepack.\mathrm{create}><javadeluxepack.\mathrm{fill}_{\mathrm{JavaDeluxePackaging}}(\mathrm{false})> \\
& <javadeluxepack.\mathrm{fill}_{\mathrm{JavaDeluxePackaging}}(\mathrm{true})> \mathbf{T} \\
\psi^1_7 = & <\mathrm{theconveyorbelt.notify}> \mathbf{T}.
\end{array}$$

Formulae ψ^1_1 *to* ψ^1_3 *are similar to formulae* ψ^0_1 *to* ψ^0_3*. Formula* ψ^1_7 *is similar to formula* ψ^0_4*. Formula* ψ^1_4 *states that static object* `theconveyorbelt` *behaves like a FIFO buffer. Formula* ψ^1_5 *limits the size of the* `theconveyorbelt` *object to* 50*. Formula* ψ^1_6 *states that an instance of the* `JavaDeluxePackaging` *class may be filled with both* `true` *and* `false` *value.*

These formulae are actually properties if we consider the assignment δ_1 *such that,* $\delta_1(javapack_1) = \mathrm{init}^{B_1}_{\mathrm{JavaPackaging}}$, $\delta_1(javapack_2) = \mathrm{new}^{B_1}_{\mathrm{JavaPackaging}}(\mathrm{init}^{B_1}_{\mathrm{JavaPackaging}})$, *etc., and*
$\delta_1(javadeluxepack) = \mathrm{init}^{B_1}_{\mathrm{JavaDeluxePackaging}}$ *(*B_1 *is a semantics of the presentation of* $Prog_1$*), and state* Init_{Prog_1}. *Thus,*

$$TS_{Prog_1,B_1}, \mathrm{Init}_{Prog_1} \models^{\delta_1}_{HML,Prog_1,Y_1} \langle Prog_1, \Psi_1\rangle.$$

Thus, we define the following contractual program:

$$CProg_1 = \langle Prog_1, \Psi_1\rangle.$$

6.2 CO-OPN/2 Implementation

Contractual CO-OPN/2 specifications and contractual programs are very similar. However, we distinguish the three following differences: (1) the body part of the Class modules of programs are different from the body part of the Class modules of CO-OPN/2 specifications; (2) the create method is not available by default in programming languages, it is replaced by a method having the name of the class without parameters; (3) the sub-typing, sub-sorting relationships are not defined for programs.

Therefore, the implement relation and the formula implementation are very close to the refine relation (Definition 5.2.12) and the formula refinement (Definition 5.2.22) respectively. However, due to the three differences above, subtle changes arise. This section defines the implement relation, the formula implementation, the implementation relation, and shows the compatibility of the refinement relation defined in Chapter 5 and the implementation relation.

6.2.1 Implement Relation

An implement relation is similar to a refine relation: it is a relation on elements of a contractual CO-OPN/2 specification and elements of a contractual program. Two differences arise with the refine relation:

- since a program defines no sub-typing and sub-sorting relationships, we do not constrain pairs of CO-OPN/2 types or sorts s, s', such that s is a sub-type or a sub-sort of s' ($s \leq s'$), to be related to program types or sorts that are in a sub-type or sub-sort relationship. Consequently, we do not constrain terms of the form sub or super to be related with similar terms;
- the implement relation allows two or more ADT sorts or two or more ADT operations of the specification to be related with the same ADT sort or the same ADT operation of the program respectively. The reason for this is that programming languages

usually have a very restricted set of ADT sorts, and there is no possibility, in programming languages, to create new ADT sorts. On the contrary, we do not allow two CO-OPN/2 Class modules to be related to the same Class module of program, because programming languages allow easily to create as many classes as necessary.

We define first elements of contractual programs, and then the implement relation.

Elements of a contractual program are defined in a way similar to elements of a contractual CO-OPN/2 specification; they are given by the global signature, the global interface and the variables used to express HML formulae.

Definition 6.2.1 *Elements of a Contractual Program.*
Let $CProg = \langle Prog, \Psi \rangle$ *be a contractual program,* $Y = (Y_s)_{s \in S_{Prog}}$ *be a* S_{Prog}*-disjointly-sorted set of variables,* $\Psi \subseteq \Psi_{Prog,Y}$ *a contract on* $Prog$ *and* Y*. The set of elements of* $CProg$*, noted* ELEM_{CProg}*, is such that*

$$\text{ELEM}_{CProg} = S^A_{Prog} \cup S^C_{Prog} \cup F^A_{Prog} \cup F^C_{Prog} \cup M_{Prog} \cup O_{Prog} \cup Y.$$

The implement relation is a relation on elements of a contractual CO-OPN/2 specification and a contractual program, that is: functional, injective on element of Class modules, and total on elements of contracts.

Definition 6.2.2 *Implement Relation.*
Let $CSpec = \langle Spec, \Phi \rangle$*,* $CProg = \langle Prog, \Psi \rangle$ *be a contractual CO-OPN/2 specification, and a contractual program respectively. An implement relation on* $CSpec$ *and* $CProg$*, noted* λ^I*, is a relation on elements of* $CSpec$ *and elements of* $CProg$*:*

$$\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg} \ ,$$

such that : $\lambda^I = \lambda^I_{S^A} \cup \lambda^I_{S^C} \cup \lambda^I_{F^A} \cup \lambda^I_{F^C} \cup \lambda^I_M \cup \lambda^I_O \cup \lambda^I_X$*, where:*

$$\begin{array}{ll}
\lambda^I_{S^A} \subseteq S^A \times S^A_{Prog} & \lambda^I_M \subseteq M \times M_{Prog} \\
\lambda^I_{S^C} \subseteq S^C \times S^C_{Prog} & \lambda^I_O \subseteq O \times O_{Prog} \\
\lambda^I_{F^A} \subseteq F^A \times F^A_{Prog} & \lambda^I_X \subseteq X \times Y \ , \\
\lambda^I_{F^C} \subseteq F^C \times F^C_{Prog} &
\end{array}$$

and

$$
\begin{aligned}
(f,f') \in \lambda^I_{FA} &\Rightarrow (f: s_1,\ldots,s_n \to s, f': s'_1,\ldots,s'_n \to s' \ or \\
&\quad\ f :\to s, f' :\to s')\ and \\
&\quad\ (s,s'),(s_i,s'_i) \in \lambda^I_{SA} \cup \lambda^I_{SC}\ (1 \le i \le n) \\
(f,f') \in \lambda^I_{FC} &\Rightarrow (f = \mathrm{init}_c, f' = \mathrm{init}_{c'}\ \ or \\
&\quad\ f = \mathrm{new}_c, f' = \mathrm{new}_{c'})\, and \\
&\quad\ (c,c') \in \lambda^I_{SC} \\
(m,m') \in \lambda^I_M &\Rightarrow m_c : s_1,\ldots,s_k,\ m'_{c'} : s'_1,\ldots,s'_k\ and \\
&\quad\ (c,c') \in \lambda^I_{SC}, (s_i,s'_i) \in \lambda^I_{SA} \cup \lambda^I_{SC}\ (1 \le i \le k) \\
(o_c, o'_{c'}) \in \lambda^I_O &\Rightarrow o : c,\ o' : c'\ and\ (c,c') \in \lambda^I_{SC} \\
(x,y) \in \lambda^I_X &\Rightarrow x \in X_s, x' \in Y_{s'}\ and\ (s,s') \in \lambda^I_{SA} \cup \lambda^I_{SC} \\
(l,l'),(l,l'') \in \lambda^I &\Rightarrow l' = l'' \\
(l,l'),(l'',l') \in \lambda^I \backslash (\lambda^I_{SA} \cup \lambda^I_{FA}) &\Rightarrow l = l'' \\
l \in \Phi &\Rightarrow \exists l' \in \mathrm{E{\scriptstyle LEM}}_{CProg}\ s.t\,(l,l') \in \lambda^I.
\end{aligned}
$$

Since we want to show that $CProg_0$ and $CProg_1$ are respectively correct implementations of $CSpec_0$ and $CSpec_1$ defined in Chapter 5, examples below give the corresponding implement relations.

Example 6.2.3 *Implement Relation on $CSpec_0$ and $CProg_0$.*
Given $CSpec_0$, $CProg_0$ of Examples 5.2.8 and 6.1.23 respectively, we define an implement relation $\lambda^I_0 \subseteq \mathrm{E{\scriptstyle LEM}}_{CSpec_0} \times \mathrm{E{\scriptstyle LEM}}_{CProg_0}$ on $CSpec_0$ and $CProg_0$ in the following way:

$$
\begin{aligned}
\lambda^I_{0_{SA}} &= \{(\mathrm{chocolate}, \mathrm{boolean}), (\mathrm{praline}, \mathrm{boolean})\} \\
\lambda^I_{0_{SC}} &= \{(\mathrm{packaging}, \mathrm{JavaPackaging}), (\mathrm{heap}, \mathrm{JavaHeap})\} \\
\lambda^I_{0_{FA}} &= \{(\mathrm{P_{praline}}, \mathrm{true_{boolean}})\} \\
\lambda^I_{0_{FC}} &= \{(\mathrm{new_{heap}}, \mathrm{new_{JavaHeap}}), (\mathrm{init_{heap}}, \mathrm{init_{JavaHeap}}), \\
&\qquad (\mathrm{new_{packaging}}, \mathrm{new_{JavaPackaging}}), (\mathrm{init_{packaging}}, \mathrm{init_{JavaPackaging}})\} \\
\lambda^I_{0_M} &= \{(\mathrm{put_{heap,packaging}}, \mathrm{insertElement_{JavaHeap,JavaPackaging}}), \\
&\qquad (\mathrm{get_{heap,packaging}}, \mathrm{removeElement_{JavaHeap,JavaPackaging}}), \\
&\qquad (\mathrm{fill_{packaging,chocolate}}, \mathrm{fill_{JavaPackaging,boolean}})\} \\
\lambda^I_{0_O} &= \{(\mathrm{the\text{-}heap}, \mathrm{theheap})\} \\
\lambda^I_{0_X} &= \{(pack_1, javapack)\}.
\end{aligned}
$$

Basically, elements of the CO-OPN/2 `Heap` and `Packaging` Class modules are related to corresponding elements of the Java `JavaHeap` and `JavaPackaging` classes. The CO-OPN/2 `chocolate` and `praline` sorts are related to the Java `boolean` primitive type. The P_{praline} generator is related to $\mathtt{true}_{\mathrm{boolean}}$. λ^I_0 given here is minimal, it is not defined for elements which are not in the contract, e.g., T_{truffle} or method `full-praline`.

Example 6.2.4 *Implement Relation on $CSpec_1$ and $CProg_1$.*
Given $CSpec_1$, $CProg_1$ of Examples 5.2.14 and 6.1.24 respectively, we define an implement relation $\lambda_1^I \subseteq \text{ELEM}_{CSpec_1} \times \text{ELEM}_{CProg_1}$ on $CSpec_1$ and $CProg_1$ in the following way:

$$\begin{aligned}
\lambda^I_{1_{SA}} =& \{(\text{chocolate}, \text{boolean}), (\text{praline}, \text{boolean}), (\text{truffle}, \text{boolean})\} \\
\lambda^I_{1_{SC}} =& \{(\text{packaging}, \text{JavaPackaging}), (\text{deluxe-packaging}, \text{JavaDeluxePackaging}), \\
& (\text{conveyor-belt}, \text{JavaConveyorBelt})\} \\
\lambda^I_{1_{FA}} =& \{(\text{P}_{\text{praline}}, \text{true}_{\text{boolean}}), (\text{T}_{\text{truffle}}, \text{false}_{\text{boolean}})\} \\
\lambda^I_{1_{FC}} =& \{(\text{new}_{\text{conveyor-belt}}, \text{new}_{\text{JavaConveyorBelt}}), (\text{init}_{\text{conveyor-belt}}, \text{init}_{\text{JavaConveyorBelt}}), \\
& (\text{new}_{\text{packaging}}, \text{new}_{\text{JavaPackaging}}), (\text{init}_{\text{packaging}}, \text{init}_{\text{JavaPackaging}}), \\
& (\text{new}_{\text{deluxe-packaging}}, \text{new}_{\text{JavaDeluxePackaging}}), (\text{init}_{\text{deluxe-packaging}}, \text{init}_{\text{JavaDeluxePackaging}})\} \\
\lambda^I_{1_M} =& \{(\text{put}_{\text{conveyor-belt,packaging}}, \text{insertElement}_{\text{JavaConveyorBelt,JavaPackaging}}), \\
& (\text{get}_{\text{conveyor-belt,packaging}}, \text{removeElement}_{\text{JavaConveyorBelt,JavaPackaging}}), \\
& (\text{fill}_{\text{packaging,chocolate}}, \text{fill}_{\text{JavaPackaging,boolean}}), \\
& (\text{fill}_{\text{deluxe-packaging,chocolate}}, \text{fill}_{\text{JavaDeluxePackaging,boolean}})\} \\
\lambda^I_{1_O} =& \{(\text{the-conveyor-belt}, \text{theconveyorbelt})\} \\
\lambda^I_{1_X} =& \{(pack_i, javapack_i)\ (1 \leq i \leq 51), (dpack, javadeluxepack)\}.
\end{aligned}$$

Similarly to λ_0^I, the implement relation λ_1^I relates elements of the CO-OPN/2 `ConveyorBelt`, `Packaging` and `DeluxePackaging` Class modules to corresponding elements of the Java `JavaHeap`, `JavaPackaging` and `JavaDeluxePackaging` classes. CO-OPN/2 `chocolate`, `praline` and `truffle` sorts are related to the Java `boolean` primitive type. The P_{praline} generator is related to $\texttt{true}_{\text{boolean}}$, and T_{truffle} generator is related to $\texttt{false}_{\texttt{boolean}}$.

Remark 6.2.5 *A CO-OPN/2 implement relation, λ^I, given by Definition 6.2.2, is actually an implement relation as stated by Definition 3.2.8, since λ^I is total on elements of the contract.*

6.2.2 Formula Implementation

The implement relation is functional. Therefore, the implementation of a CO-OPN/2 term, of a CO-OPN/2 observable event, and of a HML formula on a CO-OPN/2 specification consists in replacing every CO-OPN/2 element by the element of the program to which it is related by the implement relation.

We present first the term implementation, second the event implementation, and third the HML formula implementation.

Definition 6.2.6 *Term Implementation.*
Let $CSpec = \langle Spec, \Phi\rangle$ and $CProg = \langle Prog, \Psi\rangle$ be a contractual CO-OPN/2 specification

and a contractual program respectively. Let $T_{\Sigma,X}$ *be the set of terms of* $Spec$ *with variables in* X*, and* $T_{\Sigma_{Prog},Y}$ *be the set of terms of* $Prog$ *with variables in* Y*. Let* $\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$ *be an implement relation on elements of* $CSpec$ *and elements of* $CProg$*. The term implementation induced by* λ^I*, noted* $\Lambda^I_T : T_{\Sigma,X} \to T_{\Sigma_{Prog},Y}$*, is a partial function, such that:*

$$\Lambda^I_T(x) = \begin{cases} y & \text{if } (x,y) \in \lambda^I, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda^I_T(f) = \begin{cases} f' & \text{if } f :\to s \text{ and } (f,f') \in \lambda^I, \\ \text{undefined otherwise} \end{cases}$$

$$\Lambda^I_T(f(t_1,\ldots,t_n)) = \begin{cases} f'(\Lambda^I_T(t_1),\ldots,\Lambda_T(t_n)), & \text{if } (f,f') \in \lambda^I, \text{ and} \\ & \Lambda^I_T(t_i) \text{ is defined } (1 \le i \le n), \\ \text{undefined otherwise.} \end{cases}$$

Since implement relations are weaker than refine relations for the sub-typing and sub-sorting relationships, it may happen that a contractual program defines no sub-typing, while the contractual CO-OPN/2 specification defines a sub-typing. CO-OPN/2 terms containing sub_{c,c_1} and super_{c,c_1} can be rewritten with terms containing exclusively new_{c_1} and init_{c_1} (see Definition 4.2.1). Consequently, even though the contractual program defines no sub-typing, these CO-OPN/2 terms can be transformed into terms of the program.

Example 6.2.7 *Implementation of Terms with* sub *and* super*.*
Let $CSpec = \langle Spec, \Phi\rangle$ *and* $CProg = \langle Prog, \Psi\rangle$ *be a contractual CO-OPN/2 specification and a contracual program respectively. Let* $\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$ *be an implement relation on elements of* $CSpec$ *and elements of* $CProg$*. The following object identifiers terms are implemented in the following way:*

$$\begin{aligned} \Lambda^I_T(\text{init}_c) &= \text{init}_{c'} && \text{if } (c,c') \in \lambda^I \\ \Lambda^I_T(\text{new}_c(\text{init}_c)) &= \text{new}_{c'}(\text{init}_{c'}) && \text{if } (c,c') \in \lambda^I \\ \Lambda^I_T(\text{sub}_{c,c_1}(\text{new}_c(\text{init}_c))) &= \Lambda^I_T(\text{new}_{c_1}(\text{sub}_{c,c_1}(\text{init}_c))) \\ &= \Lambda^I_T(\text{new}_{c_1}(\text{init}_{c_1})) \\ &= \text{new}_{c'_1}(\text{init}_{c'_1}) && \text{if } (c_1,c'_1) \in \lambda^I \\ \Lambda^I_T(o_c) &= o'_{c'} && \text{if } (o_c,o'_{c'}) \in \lambda^I. \end{aligned}$$

The event implementation is similar to the event refinement, except for events containing the create method. In that case, the event is implemented by an event of the program containing the default constructor of the class (whose name is the name of the class).

Definition 6.2.8 *Event Implementation.*
Let $CSpec = \langle Spec, \Phi\rangle$*,* $CProg = \langle Prog, \Psi\rangle$ *be a contractual CO-OPN/2 specification, and a contractual program respectively. Let* $Event_{Spec,X}$ *be the set of observable*

events of Spec and X, $Event_{Prog,Y}$ be the set of observable events of Prog and Y, and $\lambda^I \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CProg}$ be a refine relation on CSpec and CProg. The event implementation induced by λ^I, noted $\Lambda^I_{Event} : Event_{Spec,X} \to Event_{Prog,Y}$, is a partial function such that:

$$\Lambda^I_{Event}(t.m) = \begin{cases} \Lambda^I_T(t).m' & \textit{if } \Lambda^I_T(t) \textit{ is defined and } (m,m') \in \lambda^I, \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I_{Event}(t.m(t_1,\ldots,t_k)) = \begin{cases} \Lambda^I_T(t).m'(\Lambda^I_T(t_1),\ldots,\Lambda^I_T(t_k)) & \textit{if } \Lambda^I_T(t), \Lambda^I_T(t_i)\ (1 \le i \le n) \textit{ is} \\ & \textit{defined and } (m,m') \in \lambda^I, \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I_{Event}(t.\mathrm{create}) = \begin{cases} \Lambda^I_T(t).c'() & \textit{if } \Lambda^I_T(t) \textit{ is defined, } \Lambda^I_T(t) \in (T_{\Sigma_{Prog},Y})_{c'}\ , \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I_{Event}(t.\mathrm{destroy}) = \begin{cases} \Lambda^I_T(t).\mathrm{destroy} & \textit{if } \Lambda^I_T(t) \textit{ is defined,} \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I_{Event}(e_1\ //\ \ldots\ //\ e_n) = \begin{cases} \Lambda^I_{Event}(e_1)\ //\ \ldots\ //\ \Lambda^I_{Event}(e_n) & \textit{if } \Lambda^I_{Event}(e_i) \textit{ is defined} \\ & (1 \le i \le n), \\ \textit{undefined otherwise.} \end{cases}$$

Definition 6.2.9 *CO-OPN/2 Formula Implementation.*
Let $CSpec = \langle Spec, \Phi\rangle$, $CProg = \langle Prog, \Psi\rangle$ be a contractual CO-OPN/2 specification, and a contractual program respectively, and $\lambda^I \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CProg}$ be an implement relation on elements of CSpec and elements of CProg. The formula implementation induced by λ^I, noted $\Lambda^I : \textsc{Prop}_{Spec,X} \to \textsc{Prop}_{Prog,Y}$, is a partial function such that:

$$\Lambda^I(\mathbf{T}) = \mathbf{T}$$

$$\Lambda^I(\neg\phi) = \begin{cases} \neg\Lambda^I(\phi) & \textit{if } \Lambda^I(\phi) \textit{ is defined,} \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I(\phi \wedge \psi) = \begin{cases} \Lambda^I(\phi) \wedge \Lambda^I(\psi) & \textit{if } \Lambda^I(\phi) \textit{ and } \Lambda^I(\psi) \textit{ are defined,} \\ \textit{undefined otherwise} \end{cases}$$

$$\Lambda^I(<e>\ \phi) = \begin{cases} <\Lambda^I_{Event}(e)>\ \Lambda^I(\phi) & \textit{if } \Lambda^I_{Event}(e) \textit{ and } \Lambda^I(\phi) \textit{ are defined,} \\ \textit{undefined otherwise.} \end{cases}$$

Proposition 6.2.1 *CO-OPN/2 Formula Implementation is a total function on formulae of the contract.*
Let $CSpec = \langle Spec, \Phi\rangle$, $CProg = \langle Prog, \Psi\rangle$ be a contractual CO-OPN/2 specification and a contractual program respectively. Let $\lambda^I \subseteq \textsc{Elem}_{CSpec} \times \textsc{Elem}_{CProg}$ be a CO-OPN/2 implement relation on elements of CSpec and elements of CProg. The CO-OPN/2 formula implementation induced by λ^I, $\Lambda^I : \textsc{Prop}_{Spec,X} \to \textsc{Prop}_{Prog,Y}$, is a total function on the formulae of the contract Φ of CSpec.

Proof.
The CO-OPN/2 implement relation λ^I is total on elements of the contract, thus Λ_T^I is total on terms of the contract, and consequently Λ_{Event}^I is total on $\cup_{\phi \in \Phi} Event_\phi$, the events of the properties of the contract of $CSpec$. This induces Λ^I to be total on the formulae of the contract. ■

Proposition 6.2.2 *CO-OPN/2 Formula Implementation is a Formula Implementation.* *Λ^I, as given by Definition 6.2.9, is a formula implementation as stated in Definition 3.2.11.*

Proof.
We must show the two following points:

- Λ^I is total on formulae of the contract.
  Indeed, Proposition 6.2.1 above shows this fact;
- if λ is a CO-OPN/2 refine relation, and λ^I is a CO-OPN/2 implement relation, and if $\lambda'^I = \lambda; \lambda^I$ is an implement relation, then $\Lambda'^I = \Lambda^I \circ \Lambda$.
  Indeed, term refinement and implementation, and event refinement and implementation are functional renamings. Thus, $\Lambda_T'^I = \Lambda_T^I \circ \Lambda_T$, $\Lambda_{Event}'^I = \Lambda_{Event}^I \circ \Lambda_{Event}$, and consequently $\Lambda'^I = \Lambda^I \circ \Lambda$.

■

We apply now the formula implementation to our runnning example.

Example 6.2.10 *Formula Implementation of the Contract of $CSpec_0$.*
Let $CSpec_0$ be the contractual CO-OPN/2 specification of Example 5.2.8, and $CProg_0$ be the contractual program of Example 6.1.23. Let λ_0^I be the implement relation of Example 6.2.3. The contract $\Phi_0 = \{\phi_1, \phi_2, \phi_3\}$ is implemented in the following way:

$$\Lambda_0^I(\phi_1) = \psi_1^0$$
$$\Lambda_0^I(\phi_2) = \psi_2^0$$
$$\Lambda_0^I(\phi_3) = \psi_3^0.$$

Example 6.2.11 *Formula Implementation of the Contract of $CSpec_1$.*
Let $CSpec_1$ be the contractual CO-OPN/2 specification of Example 5.2.14, and $CProg_1$ be the contractual program of Example 6.1.24. Let λ_1^I be the implement relation of Example 6.2.4. The contract $\Phi_1 = \{\phi_1^1, \phi_2^1, \phi_3^1, \phi_4^1, \phi_5^1, \phi_6^1\}$ is implemented in the following way:

$$\Lambda_1^I(\phi_1^1) = \psi_1^1 \qquad \Lambda_1^I(\phi_4^1) = \psi_4^1$$
$$\Lambda_1^I(\phi_2^1) = \psi_2^1 \qquad \Lambda_1^I(\phi_5^1) = \psi_5^1$$
$$\Lambda_1^I(\phi_3^1) = \psi_3^1 \qquad \Lambda_1^I(\phi_6^1) = \psi_6^1.$$

6.2.3 Implementation Relation

A contractual program correctly implements a contractual CO-OPN/2 specification via an implement relation λ^I, if the implementation of the contract of the contractual specification, obtained with the formula implementation Λ^I induced by λ^I, is a subset of the contract of the contractual program.

Definition 6.2.12 *Implementation of Contractual CO-OPN/2 Specifications via* λ^I.
Let $CSpec = \langle Spec, \Phi\rangle$, *and* $CProg = \langle Prog, \Psi\rangle$ *be a contractual CO-OPN/2 specification and a contractual program respectively. Let* $\lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$ *be an implement relation on* $CSpec$ *and* $CProg$, *and* Λ^I *be the formula implementation univocally defined from* λ^I. $\langle Prog, \Psi\rangle$ *is an implementation of* $\langle Spec, \Phi\rangle$ *via* λ^I, *noted* $\langle Spec, \Phi\rangle \leadsto^{\lambda^I} \langle Prog, \Psi\rangle$, *iff*

$$\Lambda^I(\Phi) \subseteq \Psi.$$

A contractual program implements a contractual CO-OPN/2 specification if there exists an implement relation such that the contractual program implements the contractual specification via the implement relation.

Definition 6.2.13 *Implementation Relation.*
The implementation relation, noted $\leadsto$, *is a relation on contractual CO-OPN/2 specifications and contractual programs:*

$$\leadsto \subseteq \text{CSPEC} \times \text{CPROG},$$

such that for every $CSpec = \langle Spec, \Phi\rangle \in \text{CSPEC}$, *and every* $CProg = \langle Prog, \Psi\rangle \in \text{CPROG}$, *then* $\langle Spec, \Phi\rangle \leadsto \langle Prog, \Psi\rangle$ *iff*

$\exists \lambda^I \subseteq \text{ELEM}_{CSpec} \times \text{ELEM}_{CProg}$ *an implement relation on* $CSpec$ *and* $CProg$, *s.t.*
$\langle Spec, \Phi\rangle \leadsto^{\lambda^I} \langle Prog, \Psi\rangle$.

The implementation phase occurs after a series of refinement steps. We must be sure that the contractual program, reached during the implementation phase, is an implementation of every contractual specification obtained during the refinement process. For this reason, we have to prove the compatibility between the refinement and the implementation relations (see Definition 3.3.4).

Proposition 6.2.3 *Compatibility of the Refinement and the Implementation Relations.*
The CO-OPN/2 refinement relation on contractual CO-OPN/2 specifications, $\sqsubseteq$, *and the CO-OPN/2 implementation relation on contractual CO-OPN/2 specifications and contractual programs,* $\leadsto$, *are compatible.*

Proof.
Follows from Proposition 3.3.1. ∎

We will now show, first that Java contractual program $CProg_0$ is a correct implementation of contractual CO-OPN/2 specification $CSpec_0$, but not a correct implementation of contractual CO-OPN/2 specification $CSpec_1$; and second, that Java contractual program $CProg_1$ is a correct implementation of contractual CO-OPN/2 specifications $CSpec_0$ and $CSpec_1$.

Example 6.2.14 *$CProg_0$ implements $CSpec_0$.*
Let $CSpec_0$, $CProg_0$ be the CO-OPN/2 contractual specification and the contractual program of Examples 5.2.8 and 6.1.23 respectively. Let λ_0^I be the implement relation of Example 6.2.3.

Example 6.2.10 show that:

$$\Lambda_0^I(\Phi_0) = \Psi_0.$$

Consequently, we have $CSpec_0 \rightsquigarrow^{\lambda_0^I} CProg_0$, and thus:

$$CSpec_0 \rightsquigarrow CProg_0.$$

Example 6.2.15 *$CProg_0$ does not implement $CSpec_1$.*
Let $CSpec_1$, and $CProg_0$ be the CO-OPN/2 contractual specification and the contractual program of Examples 5.2.14, and 6.1.23 respectively. $CProg_0$ cannot implement $CSpec_1$ because there is no implement relation on $CSpec_1$ and $CProg_0$. Indeed,

- *$CSpec_1$ defines the types `packaging` and `deluxe-packaging` and elements of this type are part of the contract Φ_1. $CProg_0$ defines the Java type `JavaPackaging`, which is meant to implement `packaging`, but does not define a Java type that can implement `deluxe-packaging`;*
- *formula $\phi_4^1 \in \Phi_1$ requires that the `the-conveyor-belt` type behaves like a FIFO buffer. It has no equivalent formula on $Prog_0$, and henceforth in Ψ_0, since $Prog_0$ behaves like a heap and not like a FIFO buffer.*

Example 6.2.16 *$CProg_1$ implements $CSpec_1$ and $CSpec_0$.*
Let $CSpec_0$, $CSpec_1$, and $CProg_1$ be the CO-OPN/2 contractual specifications and the contractual program of Examples 5.2.8, 5.2.14, and 6.1.24 respectively. Let λ_1^I be the implement relation of Example 6.2.4.

Example 6.2.11 shows that:

$$\Lambda_1^I(\Phi_1) = \Psi_1.$$

Consequently, we have $CSpec_1 \leadsto^{\lambda_1^I} CProg_1$*, and thus:*

$$CSpec_1 \leadsto CProg_1.$$

Since the implementation relation and the refinement relation are compatible, the following holds:

$$\Lambda_1^I(\Lambda_0(\Phi_0)) \subseteq \Psi_1,$$

i.e., $CSpec_0 \leadsto^{\lambda_0;\lambda_1^I} CProg_1$*, and thus:*

$$CSpec_0 \leadsto CProg_1.$$

6.3 Compositional CO-OPN/2 Implementation

Section 5.3 defines a hierarchical operator on contractual CO-OPN/2 specifications, that adds an incomplete contractual CO-OPN/2 specification to some complete contractual CO-OPN/2 specifications. The compositional CO-OPN/2 refinement is then defined as the replacement of every component by a component that refines it. Since the CO-OPN/2 implementation is very similar to CO-OPN/2 refinement, we define as well in a similar way a hierarchical operator for building compositional contractual programs, and a compositional implementation, that replaces every component of a compositional contractual CO-OPN/2 specification by a component that implement it.

6.3.1 Compositional Contractual Programs

A compositional contractual program is a set of complete contractual programs extended, by the means of a hierarchical operator, with an incomplete contractual program.

An incomplete program is a set of ADT modules and Class modules of program, such that the incomplete program may use elements not defined in these modules.

Definition 6.3.1 *Incomplete Program.*
An incomplete program denoted, $\Delta Prog$*, is a set of ADT modules of programs and a set of Class modules of programs, i.e.,*

$$\Delta Prog = \{(Md^{\mathsf{A}})_i \mid 1 \leq i \leq n\} \cup \{(Md^{\mathsf{C}}_{Prog})_j \mid 1 \leq j \leq m\}.$$

Notation 6.1.5, and Definition 6.1.9 (terms of program), Definition 6.1.10 (observable events of program), and Definition 6.1.11 (HML formulae on programs) are extended to incomplete programs.

An incomplete contractual program is a pair made of an incomplete program and a set of HML formulae expressed on the incomplete program. As for incomplete contractual CO-OPN/2 specifications, the HML formulae, constituting the contract part of an incomplete contractual program, are not necessarily HML properties.

Definition 6.3.2 *Incomplete Contractual Program.*
Let $\Delta Prog$ be an incomplete program, $Y = (Y_s)_{s \in S_{Prog}}$ be a S_{Prog}-disjointly-sorted set of variables, and $\Delta\Psi \subseteq \Psi_{\Delta Prog, X}$ be a set of HML formulae on $\Delta Prog$. An incomplete contractual program, noted $\Delta CProg$, is a pair:

$$\Delta CProg = \langle \Delta Prog, \Delta\Psi \rangle.$$

We will say indifferently *complete (contractual) program* and *(contractual) program.*

Hierarchical operators on contractual programs are similar to hierarchical operators on contractual CO-OPN/2 specifications: a set of complete contractual programs is extended with an incomplete contractual program. The result is a complete contractual program, otherwise it is not defined.

Definition 6.3.3 *Hierarchical Operator on Contractual Programs.*
Let $\Delta CProg = \langle \Delta Prog, \Delta\Psi \rangle$ be an incomplete contractual program. Let $CProg_i = \langle Prog_i, \Psi_i \rangle$ $(1 \leq i \leq k)$ be k contractual programs. A k-ary hierarchical operator on programs based on $\Delta CProg$ is a partial function, noted $f_{\Delta CProg} : \text{CPROG}^k \to \text{CPROG}$, such that:

$$f_{\Delta CProg}(CProg_1, \ldots, CProg_k) = \begin{cases} CProg = \langle Prog, \Psi \rangle, \textit{ such that:} \\ \quad Prog = \bigcup_{i \in \{1,\ldots,k\}} Prog_i \bigcup \Delta Prog \quad \textit{and} \\ \quad \Psi = \bigcup_{i \in \{1,\ldots,k\}} \Psi_i \bigcup \Delta\Psi \quad \textit{and} \\ \quad \langle Prog, \Psi \rangle \textit{ is a complete contractual} \\ \quad \textit{program,} \\ \textit{undefined otherwise.} \end{cases}$$

Remark 6.3.4 *There are cases where the composition of CO-OPN/2 specifications is undefined. The same cases apply for programs, and let their composition be not defined.*

6.3.2 Compositional Implementation

The CO-OPN/2 compositional implementation replaces every complete component of a compositional contractual CO-OPN/2 specification by a complete contractual program that implements it. In addition, it replaces the incomplete contractual CO-OPN/2 specification by an incomplete contractual program that syntactically implements it.

First we define incomplete programs, and then we show that the implementation component by component is actually compositional.

We extend Definition 6.2.1 (elements of a contractual program), Definition 6.2.2 (implement relation), and Definition 6.2.9 (formula implementation) to incomplete specifications and incomplete programs. Thus, we can define the syntactical implementation of incomplete contractual CO-OPN/2 specification by incomplete contractual programs.

Definition 6.3.5 *Syntactic Implementation of Incomplete Contractual CO-OPN/2 Specification.*
Let $\Delta CSpec = \langle \Delta Spec, \Delta\Phi \rangle$ *be an incomplete contractual CO-OPN/2 specification and* $\Delta CProg = \langle \Delta Prog, \Delta\Psi \rangle$ *be an incomplete contractual program. Let* λ^{Δ} *be an implement relation on elements of* $\Delta CSpec$ *and* $\Delta CProg$ *and* Λ^{Δ} *the corresponding formula implementation.* $\Delta CProg$ *syntactically refines* $\Delta CSpec$*, noted* $\Delta CSpec \leadsto^{\Delta} \Delta CProg$ *iff:*

$$\Lambda^{\Delta}(\Delta\Phi) \subseteq \Delta\Psi.$$

Theorem 6.3.1 *CO-OPN/2 Compositional Implementation.*
Let $\Delta CSpec = \langle \Delta Spec, \Delta\Phi \rangle$ *be an incomplete contractual CO-OPN/2 specification, and* $\Delta CProg = \langle \Delta Prog, \Delta\Psi \rangle$ *be an incomplete contractual program. Let* $f_{\Delta CSpec} : \text{CSPEC}^k \rightarrow \text{CSPEC}$ *be a* k*-ary compositional operator on contractual CO-OPN/2 specifications based on* $\Delta CSpec$*, and* $f_{\Delta CProg} : \text{CPROG}^k \rightarrow \text{CPROG}$ *be a* k*-ary compositional operator on contractual programs based on* $\Delta CProg$*. Let* $CSpec_i = \langle Spec_i, \Phi_i \rangle$ $(1 \leq i \leq k)$ *be* k *disjoint contractual CO-OPN/2 specifications, and* $CProg_i = \langle Prog_i, \Psi_i \rangle$*,* $(1 \leq i \leq k)$ *be* k *contractual programs with disjoint classes, such that*
$CSpec = \langle Spec, \Phi \rangle = f_{\Delta CSpec}(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_k, \Phi_k \rangle)$ *and*
$CProg = \langle Prog, \Psi \rangle = f_{\Delta CProg}(\langle Prog_1, \Psi_1 \rangle, \ldots, \langle Prog_k, \Psi_k \rangle)$ *are defined. The following holds:*

$$\Delta CSpec \leadsto^{\Delta} \Delta CProg \text{ and } \langle Spec_i, \Phi_i \rangle \leadsto \langle Prog_i, \Psi_i \rangle, 1 \leq i \leq k \quad \Rightarrow$$
$$f_{\Delta CSpec}(\langle Spec_1, \Phi_1 \rangle, \ldots, \langle Spec_k, \Phi_k \rangle) \leadsto f_{\Delta CProg}(\langle Prog_1, \Psi_1 \rangle, \ldots, \langle Prog_k, \Psi_k \rangle).$$

Proof.
We must prove that there exists $\lambda^I : \text{ELEM}_{CSpec} \rightarrow \text{ELEM}_{CProg}$, an implement relation, such that $\Lambda^I(\Phi) \subseteq \Psi$.

We have:

$$\text{ELEM}_{CSpec} = \bigcup_{i \in \{1,\ldots,k\}} \text{ELEM}_{CSpec_i} \bigcup \text{ELEM}_{\Delta CSpec} \text{ and}$$
$$\text{ELEM}_{CProg} = \bigcup_{i \in \{1,\ldots,k\}} \text{ELEM}_{CProg_i} \bigcup \text{ELEM}_{\Delta CProg}.$$

In addition, we have:

$$\Delta CSpec \leadsto^{\Delta} \Delta CProg \Rightarrow \exists \lambda^{\Delta} : \text{ELEM}_{\Delta CSpec} \rightarrow \text{ELEM}_{\Delta CProg} \text{ s.t. } \Lambda^{\Delta}(\Delta\Phi) \subseteq \Delta\Psi$$
$$\langle Spec_i, \Phi_i \rangle \leadsto \langle Prog_i, \Psi_i \rangle \Rightarrow \exists \lambda_i^I \text{ s.t. } \Lambda_i^I(\Phi_i) \subseteq \Psi_i, \ (1 \leq i \leq k).$$

Thus, we construct the implement relation $\lambda^I : \textsc{Elem}_{CSpec} \to \textsc{Elem}_{CProg}$ in the following way:

$$\lambda^I(e) = \begin{cases} \lambda_i^I(e), & \text{if } e \in \textsc{Elem}_{CSpec_i} \\ \lambda^\Delta(e) & \text{if } e \in \textsc{Elem}_{\Delta CSpec} \\ \text{undefined otherwise.} \end{cases}$$

λ^I is actually a refine relation. Indeed, first, λ^Δ, λ_i^I $(1 \leq i \leq k)$ are implement relations, thus λ^I is total on the contract; second, $CSpec_i$ $(1 \leq i \leq k)$ are all disjoint, and $CProg_i$ $(1 \leq i \leq k)$ have disjoint classes, thus λ^I is functional on every elements and injective on Class elements.

The formula implementation is given by:

$$\Lambda^I(\phi) = \begin{cases} \Lambda_i^I(\phi), & \text{if } \phi \in \Phi_i \\ \Lambda^\Delta(\phi), & \text{if } \phi \in \Delta\Phi \\ \text{undefined otherwise.} \end{cases}$$

Thus, $\Lambda^I(\Phi_i) \subseteq \Psi_i$, $(1 \leq i \leq k)$, and $\Lambda(\Delta\Phi) = \Delta\Psi$. Finally, we have trivially $\Lambda(\Phi) \subseteq \Psi$.
■

Remark 5.3.7 applies as well on the compositional implementation. Indeed, it is essential that $f_{\Delta CProg}(\langle Prog_1, \Psi_1\rangle, \ldots, \langle Prog_k, \Psi_k\rangle)$ be defined, otherwise the theorem cannot be guaranteed.

Remark 6.3.6 *In the case of CO-OPN/2 compositional refinement, it is necessary that the components of the high-level compositional contractual CO-OPN/2 specification be made of disjoint ADT and Class modules, and as well the components of the lower-level compositional contractual CO-OPN/2 specification. Otherwise, it is not guaranteed that the refine relation is actually a refine relation.*

In the case of CO-OPN/2 compositional implementation, the same condition applies. However, since the implement relation allows two different CO-OPN/2 (ADT) sorts to be refined by the same program sort, the components of the compositional contractual program may share ADT modules of programs, but must have disjoint sets of Class modules of program.

Chapter 7

Implementing CO-OPN/2 Specifications in Java

Chapter 6 defines a theory of implementation for the CO-OPN/2 specifications language, and object-oriented languages. This chapter is devoted to the special case of implementations using the Java programming language.

We think that every refinement process should end with a CO-OPN/2 specification that is as close as possible to the Java program, so that the implementation phase is trivially performed. By close, we mean two things: first, every instruction of the Java program is specified, and second, the transition system obtained with the CO-OPN/2 specification is the same as the one obtained with the Java program.

Therefore, this chapter first provides CO-OPN/2 specifications close to Java programs. Second, the running example of Chapter 6 is revisited, and a CO-OPN/2 specification close to the Java program defined in Chapter 6 is provided. Finally, some advices are given about how to build abstract contractual CO-OPN/2 specifications that can be refined to CO-OPN/2 specifications of Java programs, and implemented in Java, according to the implementation relation defined in Chapter 6.

7.1 CO-OPN/2 Specifications of Java Programs

We think that the most concrete contractual CO-OPN/2 specification that is reached at the end of a refinement process, should encompass the whole complexity of a Java program: instructions and behaviour. All instructions of the Java program should be considered in the contractual CO-OPN/2 specification. Thus, the contractual Java program itself is easily built from the contractual CO-OPN/2 specification. All behaviour arising in the Java program should be present in the transition system of the most concrete contractual CO-OPN/2 specification. Therefore, the contractual Java program is ensured to be a correct implementation, since the last contractual specification is actually a correct refinement of the most abstract specifications obtained during the refinement process.

This section explains how it is possible to build CO-OPN/2 specifications reaching the aim of being close to Java programs. It introduces several Java concepts. They are either part of the Java Programming Language [6, 48, 36] or part of the Java Virtual Machine [49]. For each of them, we give our design decisions for their specifications in the CO-OPN/2 language. Report [32] gives a fully detailed description of CO-OPN/2 specifications of Java concepts presented here.

7.1.1 Java Programming Language and Java Virtual Machine

The Java programming language is an object-oriented language, with the particularity that a given Java program can be executed on any operating system and host machine. Indeed, every Java program is compiled into a platform independent code, called *bytecode*. The bytecode can be interpreted by any Java Virtual Machine that is an interpreter dependent of the underlying system. Therefore, in addition to the traditional client/server paradigm, it is possible to use the *mobile code* paradigm, i.e., a piece of Java program is sent and executed remotely.

Each Java Virtual Machine can support many threads of execution at once. These threads independently execute Java code that operates on Java values and objects residing in a shared main memory. Threads may be supported by having many hardware processors, or by time-slicing one or many hardware processors. The Java Virtual Machine initially starts up with a single non-daemon thread which typically calls the method `main` of some *Class object*. For every class, there exists a special object, called Class object, whose name is the same as the name of the class. This object exists even if no instance objects of the class have been created.

CO-OPN/2 Specifications

The Java Virtual Machine is specified by the CO-OPN/2 `JVM` class depicted by Figure 7.1. Method `java` specifies the Java interpreter. Parameter `ClassName` (of type `String`) is the name of the Class object whose `main` method has to be executed; parameters `args` (whose type is an array of strings) are the parameters of the `main` method. Method `java` stores the pair made of the identity of Class object `ClassName` (of type `JavaObject`) and the parameters `args`. Method `main` of object `ClassName` with parameters `args` is actually called by transition `begin` after the identity of the call `<cnt,ClassName>` has been registered. The need for the registration of the call is explained in the sequel.

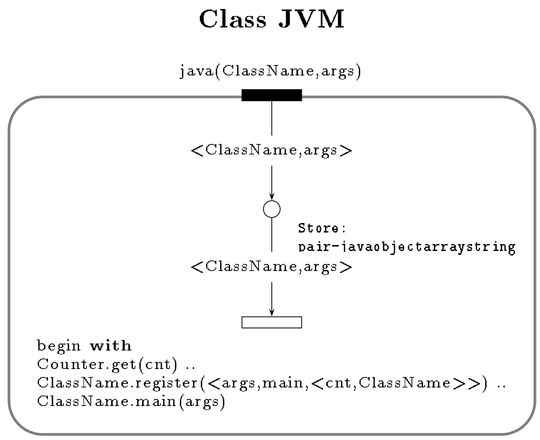


Figure 7.1: CO-OPN/2 Specification of a Java Virtual Machine

7.1.2 Java Types

There are 3 kinds of Java types: *Primitive*, *Reference* types and the `null` type.

Primitive types are the `boolean` type and the *numeric* types. The `boolean` type defines the two values `true`, `false`, and the usual operators on booleans. *Numeric* types are: (a) *integral*, i.e., signed two's complement integers: `byte` (8-bits), `short` (16-bits), `int` (32-bits), `long` (64-bits); unsigned integers: `char` (16-bits); (b) *floating-point* types, i.e., `float` (32-bits) and `double` (64-bits).

Reference types are the *class* types, the *interface* types, and the *array* types:

- Each class type is a sub-class of another class type. The Java class `Object` is the super-class of all class types. In Java, the name of the class and the name of the type defined by the class are the same.

  Sub-classes `inherit` the methods of their super-classes. A sub-class may keep a method unchanged, thus it inherits of the super-class implementation. A sub-class may change a method's implementation, thus it *overrides* the super-class method. The implementation provided by the super-class is no longer available for the sub-class, unless it invokes explicitly the super-class implementation, using the `super` keyword in calls of the form `super.m()`, where `m` is the father's implementation of the method `m`. The `super` keyword can be used from within a direct sub-class only, i.e., constructions of the form `super.super.m()` calling method `m` of the grandfather class are not allowed. A sub-class may add new methods, they are available only for the sub-class and its children, but not for its super-class.

- The Java programming language does not support multiple inheritance, i.e. each class has exactly one parent class, except for the `Object` class, which is the root and has no parent class. Java interfaces allow a class to extend several other classes, even though it has only one parent class. Java interfaces define constants (static and final variables), and interface of methods (every method is empty). A class which implements one or more interfaces has to implement the body of the methods listed in the interface.

- Elements of Java arrays are Java objects. Arrays are manipulated by reference, and behave like Java objects. Java considers that arrays are of a different reference type than class types, because a special syntax is defined for arrays.

Reference values are *pointers* to *objects*. An object is a dynamically created class instance or an array. Reference types form a hierarchy.

Primitive types allow to pass parameters by value, while reference types only allow to pass parameters by reference. In Java, in order to pass also primitive types by reference, each primitive type has a corresponding reference type. The `Boolean`, `Character`, `Double`, `Float`, `Integer` and `Long` classes are Java classes which enclose the corresponding primitive type.

The `null` type can always be converted to any reference type, it has only one possible value, the null value.

CO-OPN/2 Specifications

For every primitive type, we define a corresponding CO-OPN/2 ADT module such that every Java operator has a corresponding operation. For instance the Java `boolean` type is specified with the CO-OPN/2 ADT module `Booleans` which defines the `boolean` sort. (see Appendix A).

For every Java *class*, we propose to specify a dedicated CO-OPN/2 class. The inheritance tree of the CO-OPN/2 classes is exactly the same as the inheritance tree of the Java classes. The `Object` Java class is the super-class of all Java classes. In CO-OPN/2 this Class module is called the `JavaObject` class and defines the `javaobject` type (corresponding to the Java `Object` type). It is the super-class of all CO-OPN/2 classes related to Java. The way to build CO-OPN/2 classes specifying Java classes is explained in the following subsections.

We propose to specify Java *interfaces* as abstract CO-OPN/2 classes, and every variable defined in the Java interface as a CO-OPN/2 static object or a CO-OPN/2 constant (for ADT).

Java *arrays* are manipulated by reference, but are not defined with Java classes. Thus, we propose to define a CO-OPN/2 `JavaArray` Class module which defines the `java-array` type (corresponding to the Java `Array` type). It is defined as an array whose elements are of `javaobject` type. Java arrays do not inherit from the Java `Object` class, thus there is no inheritance relationship between the CO-OPN/2 `JavaArray` Class module and the `JavaObject` Class module. The `JavaArray` class uses the `JavaObject` class, because it specifies arrays of Java objects. An instance of the CO-OPN/2 `JavaArray` class has a reference, given by the CO-OPN/2 semantics, that can be used as a parameter by other CO-OPN/2 classes.

The Java `null` type can be used instead of any other Java type. The CO-OPN/2 semantics does not provide such an object. It is necessary to define a null object for each CO-OPN/2 type. For this reason, we will not specify the Java `null` type. When necessary, the specifier will formalise the use of the `null` type with an explicit specification.

Remark 7.1.1 *Definition 6.1.4 provides abstract definitions of programs. CO-OPN/2 specifications of Java programs are as well described with abstract definitions. The abstract definition of a program, and the abstract definition of the CO-OPN/2 specification close to the program are two different mathematical definitions.*

7.1.3 Java Methods

A Java method is a *sequential* code operating on data. It is through method invocations that data is modified or checked. Interfaces of methods, i.e., their name and parameters, are visible for a programmer, but their implementation is not visible for the programmer. The method's caller is blocked until the method returns.

Every method call is actually performed on behalf of a thread of execution. Threads are special Java objects, with a special method `run()` that describes the sequence of method calls requested by the thread to perform its execution. More precisely, every method call occurs from within the body of another method, which is currently being called, and so on till the most enclosing method which is the `run()` method of a thread. This thread has generated, by the means of its `run()` method, all this cascade of method calls, and is actually the caller of all these methods.

A Java method may be called simultaneously by several different threads or several times simultaneously by the same thread. A method handles *global variables*, *parameters* and *local variables*. In Java, as soon as a method is invoked, the parameters and the local variables of the method are duplicated, so that every method invocation induces a method execution with a separate memory space for parameters and local variables. On the contrary, global variables are not duplicated, and every method invocation accesses the same instance of the global variables. However, each time a global value is *used* or *assigned*, the global value is first *loaded* from the main memory, then used or assigned only once, and in the case of an assign, it is stored in the main memory, before a subsequent use or assign.

CO-OPN/2 Specifications

In CO-OPN/2, in order to identify each method invocation and execution, together with their private memory space for local variables and parameters, we introduce the notion of *caller's identity.* The caller's identity id is a pair id=<cnt,t>. The cnt part is an integer used to distinguish concurrent calls to a same method, it is different for every call. The t part is the reference of the thread which has initiated the cascade of method calls leading to the current method call. It stands for the Java reference of this thread. A special CO-OPN/2 Counter object provides unique counters, cnt. Before calling a method of an object, the thread must require this unique counter, and register the call it wants to perform; these two actions are performed in an unobservable manner.

We consider the following Java method:

```
public Object m(Object x){ ... ; y=o'.m'(x'); ... }
```

This method has an input parameter x of type Object and returns an output parameter of type Object as well. The Java method m(x) begins with "{" and ends with a "}". In between, several sequential Java instructions actually build the method's body. Amongst them, we find the instruction y=o'.m'(x'). We consider a Java thread t that calls method m by performing instruction y=o.m(x). Due to the Java semantics, both x and y are references of two Java objects. We assume o to be an instance of the Java Object class.

Figure 7.2 depicts: the CO-OPN/2 specification of the Java method m of object o; the call of method m' of object o'; the propagation of the thread's reference; and the handling of local variables and parameters.

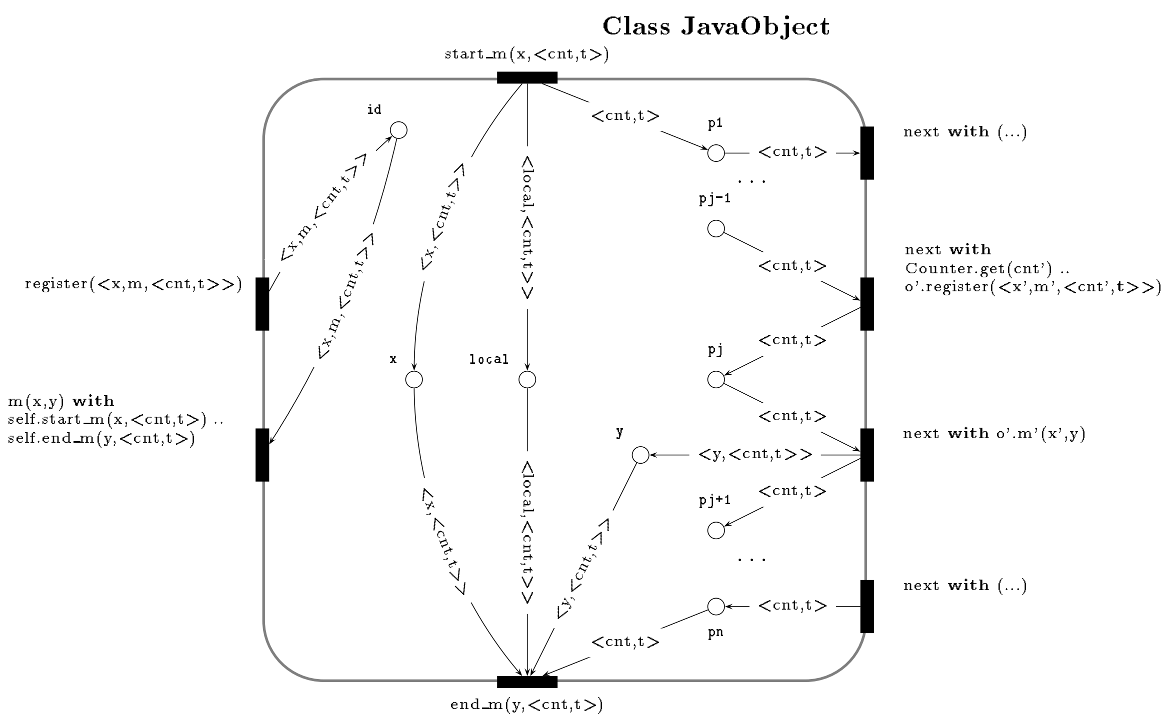


Figure 7.2: CO-OPN/2 Specification of a Java Method

The CO-OPN/2 `m(x,y)` method is called by the CO-OPN/2 object `t` (modeling the Java thread `t`). Method `m(x,y)` can be fired only if thread `t` has previously registered the call, i.e., it has registered parameter `x`, method `m`, and its identity `id` using method `register` of object `o`. Method `m(x,y)` requires the synchronization with the `start_m(x,<cnt,t>)` method followed by the `end_m(y,<cnt,t>)` method. Input parameter `x` is passed to method `start_m`, and output parameter `x` is retrieved from method `end_m`. These two methods stand for the actual begin and end of Java method `m` respectively. They are hidden methods. Thus, in terms of observable events, only method `m(x,y)` is visible, while `start_m(x,<cnt,t>)` and `end_m(y,<cnt,t>)` are hidden. Due to the CO-OPN/2 semantics, it is necessary to specify the begin and the end of a Java method with two dedicated CO-OPN/2 methods, in order to allow output parameters to be returned, and to delay the caller till the end of the method's computation.

The CO-OPN/2 `start_m(x,<cnt,t>)` method is called by the `m(x,y)` method. Method `start_m(x,<cnt,t>)` performs the following operations: (1) it stores input parameter `x` as a pair `<x,<cnt,t>>` into a dedicated place; (2) it stores the caller's identity `<cnt,t>` into a dedicated place; and (3) it creates an instance of every local variable needed by the method as a pair `<local,<cnt,t>>` into a dedicated place (one for each local variable). The `start_m(x,<cnt,t>)` corresponds to the "{" of the Java method. Storing every variable with the caller's identity has the following advantages: it helps discriminating every call to method `m(x,y)`; every call has its own private memory space for local variables and parameters.

Every instruction of the method's body is specified by one or more CO-OPN/2 methods, called `next`. Such a `next` method can be fired only if the previous `next` has finished, and as soon as itself finishes, it allows the consecutive `next` to be fired. This sequence of firing of `next` methods models the sequential execution of the method. The first `next` is firable only if `start_m(x,<cnt,t>)` method has been fired. The sequence of `next` methods respects the sequence of instructions of the Java method's body. A `next` always needs a caller's identity (`<cnt,t>`), in a place, removes it from this place and puts it into another place, where it is removed by the consecutive `next`. In the case of Figure 7.2, the body of method `m` requires to call method `m'` of object `o'`. In order to do this, the corresponding `next` method requires a new unique identifier `cnt'` by calling `Counter.get(cnt')`, and registers to object `o'` (calling `o'.register(<x',m',<cnt',t>>)`). The following `next` method then calls method `m'` of object `o'`. It is worth noting that the call to method `m'` is made on behalf of thread `t`, which is currently calling method `m`. Thus, the caller's identity `<cnt',t>` contains reference of thread `t`. Consequently, the call to `m'` *propagates the reference* of thread `t`.

The CO-OPN/2 `next` methods are called, in an unobservable manner, by a special CO-OPN/2 object specifying the scheduler of the Java Virtual Machine. The scheduler permanently loops: it calls one firable `next` method, waits for its complete execution, and then calls another firable `next` method (possibly of another object), etc.

The firing of the last `next` enables the `end_m(y,<cnt,t>)` method to be fired. The `end_m(y,<cnt,t>)` method removes the caller's identity from a dedicated place, as well as all the local variables and input/output parameters from their own places. In addition, it returns the output parameter `y`. The action of removing the caller's identity, and the local variables and parameters corresponds to the "}" of the Java method.

It is worth noting that input parameter `x` is passed to CO-OPN/2 method `m` as an object's identity, thus the method may have modified its internal state. The method's caller also has the knowledge of the input parameter's identity, thus, at the end of the method, the caller handles the object `x` with a possibly modified state.

Figure 7.2 shows an example of a method using parameters and local variables. The handling of global variables from within a method requires that global variables are loaded before they are used or assigned. The CO-OPN/2 specification of the use of global variables follows this schema: before using or assigning a global variable, the variable is duplicated into a local copy. The use or assign make use of the local copy.

Java Constructors

In Java, constructors are not inherited, therefore they are not subject to hiding or overriding. If a constructor body does not begin with an explicit constructor invocation, and the constructor being declared is not part of the primordial class `Object`, then the constructor body is implicitly assumed by the compiler to begin with a super-class constructor invocation `super()`. A call to `super()` can only occur from within a method of the direct sub-class. A call of the form `super.super()` which would invoke the default constructor of the grandfather class is not allowed.

CO-OPN/2 Specifications

In CO-OPN/2, a field called `Creation` contains all the methods that can be invoked to create an instance of a class. This field is never inherited. The method `create` exists by default for every class, and cannot be overridden by the specifier. If a non-default constructor is required, the specifier must add in the `Creation` field the non-default constructor. The CO-OPN/2 semantics states that, if, for example, the method `new-constructor` belongs to the `Creation` field of a class, then a call `o.new-constructor`, where `o` is an instance of the class, is actually treated as a call to `o.create .. o.new-constructor`. Multiple constructors can coexist in the `Creation` field of a CO-OPN/2 specification.

Java constructors are specified in a slightly different way than Java methods. Indeed, Java method requires that a thread that wants to call a method has to register the call. However, it is not possible to register a call for a non existing object. Therefore, we propose that the call is registered to the Class object (which always exists), and the constructor method itself verifies if the call has been previously registered to the Class object (instead of the object to create).

If no constructor is defined, CO-OPN/2 assumes that the `create` provided by default is used. Thus, unlike Java, there is no implicit call to the super-class constructor. Therefore, we propose the following: if a Java class has no explicit constructor, then the CO-OPN/2 specification of this class has an explicit constructor, called `super()`, and that is the exact copy of the default constructor of the direct super-class.

A Java constructor may support an overloading of parameters, i.e., the same constructor name can be used with parameters that can vary in quantity and type. Such a constructor is modelled in CO-OPN/2 using several different methods names, one for each possible Java constructor.

7.1.4 Java Keywords

The Java `static` keyword is a modifier that can be applied to method and variable declarations. There is only one copy of each `static` variable, regardless of the number of instances of the class. Every class is provided with a Class object, i.e., a special `static` object, whose name is the name of the class. A `static` method can be invoked through an instance of the class or through the the Class object. Non-`static` methods cannot be invoked through the Class object.

A `public` class or interface is visible everywhere, a `public` method is visible everywhere its class is visible. A `private` method or field variable is not visible outside its class definition. A `protected` method of field variable is visible only within its class, sub-classes, or within the package of which its class is a part. A `final` class cannot be sub-classed, a `final` method cannot be overridden, a `final` variable means that the variable has a constant value. The `extends` keyword is used in a class declaration to specify the super-class. The `implements` keyword is used to indicate that the class implements one or more interfaces. The `abstract` keyword is used to declare methods that have no implementation. Classes declared as `abstract` cannot be instantiated.

CO-OPN/2 Specifications

The Java `static` keyword is specified by the means of the CO-OPN/2 `Object` field. Every CO-OPN/2 specification Class module, that specifies a Java class, defines a CO-OPN/2 static object whose name is the name of the Java class. This CO-OPN/2 static object stands for the Class object associated to the Java class. CO-OPN/2 does not provide an equivalent of Java `static` methods. Therefore, we propose to specify these methods as other Java methods. In the case of non-`static` methods, the specifier should invoke them only through dynamically created CO-OPN/2 objects.

The Java `public` and `protected` keywords have no direct CO-OPN/2 keyword associated. However, the definition of methods or objects in the interface, and the use of the CO-OPN/2 keyword `Use` let the method or the object be `public` or `protected`. Similarly, the Java `private` keyword has no direct CO-OPN/2 keyword associated, the use of methods in the body of a CO-OPN/2 specification lets the method be private or not. The Java `final` keyword has no corresponding CO-OPN/2 keyword, the specifier must be override such classes or methods. The Java `extends` keyword is specified by the means of the CO-OPN/2 `inherit` keyword. The Java `implements` keyword has no CO-OPN/2 field or keyword associated. Java `abstract` keyword applied to classes is specified by the means of the CO-OPN/2 `Abstract` keyword. Java `abstract` methods are like other Java methods, but their body is empty, i.e., there is no `next` method. The Java `synchronized` keyword has to be specified with several CO-OPN/2 methods.

7.1.5 The Java Object Class

The Java `Object` class is the root of the hierarchy of Java classes, i.e., it is the super-class of every Java class. Every Java object is provided with: (1) a mechanism for acquiring and releasing a lock on an object; (2) a method `wait()` that enables a thread to be blocked after having called this method; (3) methods `notify()` and a `notifyAll()` that respectively resume a randomly chosen thread or every thread having performed a `wait()`; (4) a mechanism for synchronizing threads based on the notion of locks. The Java `Object` class contains other features, but we limit our specifications to the above points.

Java Locks

In Java, synchronization is implemented by accessing exclusively an internal *lock* associated with each Java `Object`. Each lock acts as a counter. If the count value is not zero because another thread holds the lock, the current thread is delayed (*blocked*) until the count is zero. The count value is incremented on entry, and decremented on exit.

CO-OPN/2 Specifications

Each class instance `o` of the CO-OPN/2 `JavaObject` class is provided with its own special `locker` place. This place stores the reference of the thread that is currently locking the object, together with the number of times it has acquired the lock. Thus, the type of the `locker` place is given by the cartesian product of `Thread` and `Integer`. An extra `locked` place is used to specify that the object is currently locked by no thread.

Two methods interact with place `locker`: `lock(t)` and `unlock(t)`. The `lock(t)` method acquires the lock of object `o` on behalf of the thread `t`. After the firing of method `lock(t)`, thread `t` is the locker of object `o`. Similarly, after the firing of the `unlock(t)` method, thread `t` releases one lock of object `o`.

Figure 7.3 depicts a part of the `JavaObject` class: the `locker` and the `locked` places, and the two CO-OPN/2 methods `lock(t)` and `unlock(t)`. The `locker` place stores pairs `<t,i>`, where `t` is a thread's identity and `i` is the number of locks that thread `t` has

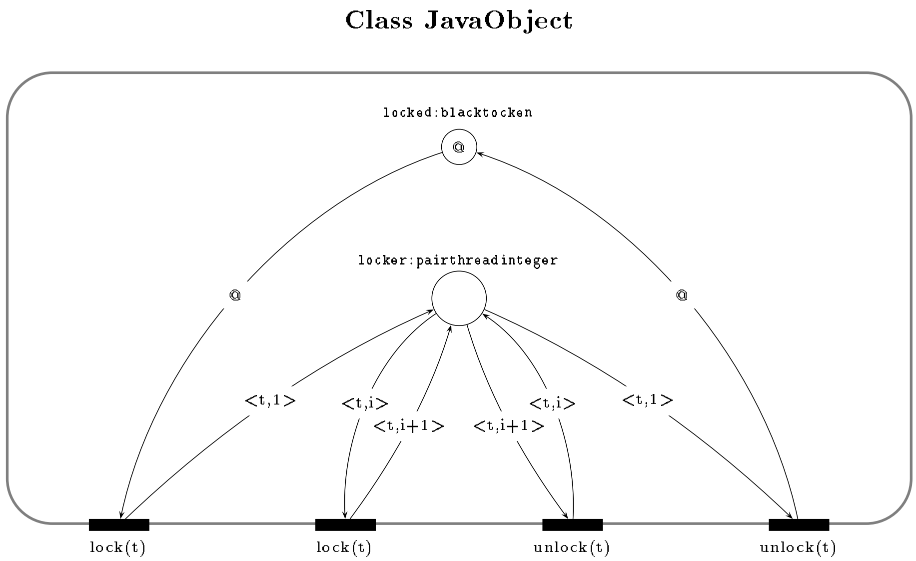


Figure 7.3: CO-OPN/2 Specification of Java Locks

acquired on object `o`. The `locked` place stores the value @ when no thread is currently locking the object.

The `lock(t)` method is given by two axioms. The first axiom (given by the CO-OPN/2 `lock(t)` method on the left of the figure) specifies that if there is no current locker object, then `t` becomes the current locker with one lock on the current object: value @ in place `locked` is removed and value `<t,1>` is inserted in place `locker`. The second axiom (given by the CO-OPN/2 `lock(t)` method on the middle of the figure) specifies that if the current locker is already `t`, then its number of locks is increased by one: token `<t,i>` is replaced by token `<t,i+1>`. It is worth noting that if `t` is not the current locker, then neither the first axiom nor the second axiom for `lock(t)` can be fired, thus, `t` is blocked until one of these two axioms is firable.

The `unlock(t)` method is given by two axioms. The first axiom (given by the CO-OPN/2 `unlock(t)` method on the middle of the figure) specifies that if the current locker is `t` and if it possesses more than one lock on the current object, then `t` releases one lock: token `<t,i+1>` is replaced by token `<t,i>`. The second axiom (given by the CO-OPN/2 `unlock(t)` method on the right of the figure) specifies that if the current locker is `t` and if it possesses exactly one lock on the current object, then, `t` releases its last lock on the current object, and is no longer the current locker: value `<t,1>` is removed from place `locker` and value @ is inserted in place `locked`.

As the CO-OPN/2 `JavaObject` class is the super-class of all the CO-OPN/2 classes related to Java, every sub-class is provided with the same mechanism of lock as described above.

Wait, Notify, NotifyAll

Java method `wait()` enables a thread to be removed from the scheduled threads. Methods `notify()` and `notifyAll()` resume respectively one or every thread having performed a `wait()`.

Every object, in addition to having an associated lock, has an associated wait set, which is a set of threads. When an object is first created, its wait set is empty. Methods `wait()`, `notify()`, and `notifyAll()` interact with the lock, the wait set and the scheduling mechanism for threads.

A thread can invoke method `wait()` only if it has already locked the object. The `wait()` method then adds the thread to the wait set, disables the thread for thread scheduling purposes, and performs as many unlock operations as the numbers of locks performed by the thread on the object. The thread then remains inactive until one of the three following things happens: (1) some other thread invokes the `notify()` method for that object, and the inactive thread happens to be the one arbitrarily chosen as the one to notify; (2) some other thread invokes the `notifyAll()` method for that object; (3) if the call by the inactive thread to the `wait()` method specifies a time-out interval, then the specified amount of real time has elapsed. The inactive thread is then removed from the wait set and re-enabled for thread scheduling. It then locks the object again (which may involve competing in the usual manner with other threads); once it has gained control of the lock, it performs additional lock operations such that the state of the object's lock is exactly as it was when the `wait()` method was invoked. Finally, it returns from the invocation of the `wait()` method.

The `notify()`, `notifyAll()` methods can be invoked for an object, only when the current thread has already locked the object's lock. In the case of the `notify()` method, one thread is arbitrarily chosen in the wait set, removed from the wait set and re-enabled; in the case of the `notifyAll()` method, all the threads in the wait set are removed from the wait set and re-enabled. If method `wait()` has not been previously called, methods `notify()`, `notifyAll()` have not effect.

CO-OPN/2 Specifications

The CO-OPN/2 `JavaObject` class maintains a special place named `wait_set` whose type is a pair made of the identity of (1) the calling thread and the number of locks it holds, and (2) the caller's identity. The Java methods `wait()`, `notify()`, and `notifyAll()` are specified in a similar way as other Java methods.

As the CO-OPN/2 `JavaObject` class is the super-class of all the CO-OPN/2 classes related to Java, every sub-class is provided with wait sets and the CO-OPN/2 methods specifying Java `wait()` and `notify()` methods. It is the same for each class instance.

Figure 7.4 depicts a part of the `JavaObject` class: the `wait_set` place and the specification of the Java methods wait() and notify(). The body of Java method wait() is depicted on the right part of the figure, while the body of Java method notify() is depicted on the left part of the figure. This figure does not show the case where an inactive thread becomes active again because a time-out has elapsed; and the case where the notify() method is invoked before the invocation of the wait() method.

The CO-OPN/2 wait method requires simply the synchronization with the start_wait(<cnt,t>) and the end_wait(<cnt,t>) methods of a given caller's identity previously registered. The start_wait(<cnt,t>) inserts the caller's identity <cnt,t> into place p11. The first next method in the right part of the figure: (1) removes token <t,i> from place locker; (2) inserts token @ in place locked, i.e. it releases all locks that t maintains on the object; (3) stores this number of locks in place wait_set; (4) moves the caller's identity, <cnt,t> from place p11 to place p12. Token <t,i> in place locker means that thread t locks the object with i locks. If t is not currently locking the object, the next method cannot be fired, and t is delayed until it locks the object. As soon as t obtains a lock on the object, the next becomes firable.

At this point, no method concerning the execution of Java method wait(), with caller's identity <cnt,t> is firable, unless notify is invoked.

Class JavaObject

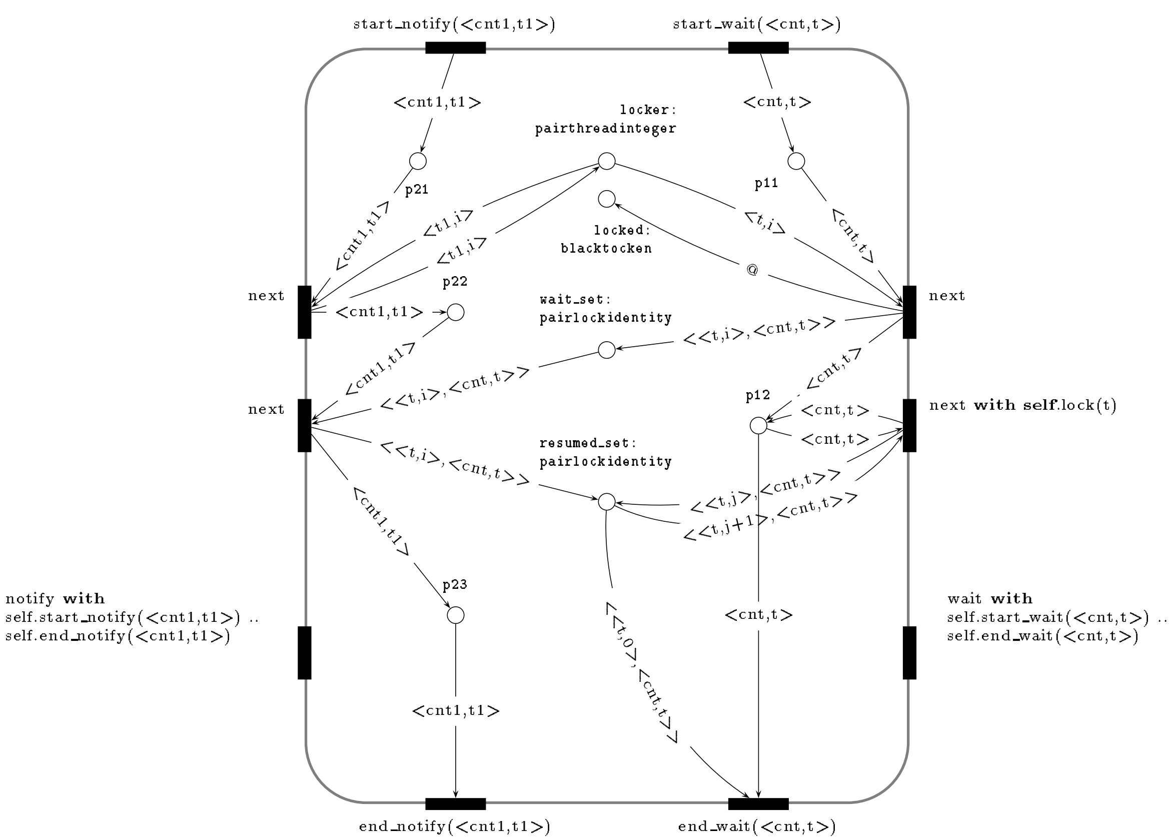


Figure 7.4: CO-OPN/2 Specification of wait(), notify()

We consider now a thread t1 calling method notify after having previously registered its call. The start_notify(<cnt1,t1>) stores the caller's identity into place p21. The first next method on the left of the figure checks if t1 owns the lock of the object. If it is not the case, then method next is not firable until t1 acquires at least one lock. If we assume that t1 owns at least one lock on the object, then method next inserts the caller's identity <cnt1,t1> into place p21. The second next on the left part of the figure can then be fired. It moves a token, randomly chosen, from the wait_set place to the resumed_set place. It also moves the caller's identity <cnt1,t1> of the thread which performed the start_notify(<cnt1,t1>) from the p22 place to the p23 place. Finally, the end_notify(<cnt1,t1>) removes the <cnt1,t1> from place p23 and returns. The CO-OPN/2 specification of the Java notify() method essentially moves one thread from the wait set to the resumed set.

We come back now to the wait method. As soon as the thread, which performed the start_wait(<cnt,t>) method, arrives in the resumed_set, the second method next on the right part of the figure can be fired. It re-acquires all the locks that have been released by t, i.e., it calls self.lock as many times as the number of locks. When all the locks have been re-acquired, the end_wait(<cnt,t>) method can be fired, and returns.

Java Synchronized Methods

In order to allow exclusive access to an object, Java offers only one primitive which is the synchronized keyword. A Java synchronized method m is declared in the following way:

```
public synchronized Object m(Object x) { ... }
```

In order to execute a synchronized method, a thread has to compete for the lock of the object which is the method's owner. Subsequently, this thread is called the locker thread. Synchronized methods work in the following way:

- A synchronized method ensures that only one thread at a time can be executing this method. It is the locker thread;
- The locker thread can be executing concurrently several synchronized or non synchronized methods of a given object;
- Several synchronized methods of the same object ensure that only the locker thread can execute them at the same time. Note that this thread can execute several times the same synchronized method and some of them simultaneously;
- Consider a given object with some of its methods declared as synchronized and some of them not. In this case, exclusive access to the object is not ensured, because *any* thread (locker or not) can execute at any time a non synchronized method, even if the locker thread is already executing a synchronized method;

- Exclusive access is guaranteed only if *every* method is declared as `synchronized`. Otherwise, the exclusive access is not guaranteed.

A `synchronized` method automatically performs a lock operation when it is invoked; its body is not executed until the *lock* operation has successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked. If the method is `static`, it locks the lock associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an *unlock* operation is automatically performed on that same lock.

CO-OPN/2 Specifications

A `synchronized` Java method is specified in the same way as other Java methods. The acquisition of the lock is performed internally by the method's body. Figure 7.5 depicts the CO-OPN/2 specification of a `synchronized` method. We assume that a thread `t` performs instruction `y=o.m(x)`, and Java method `m` is declared `synchronized`.

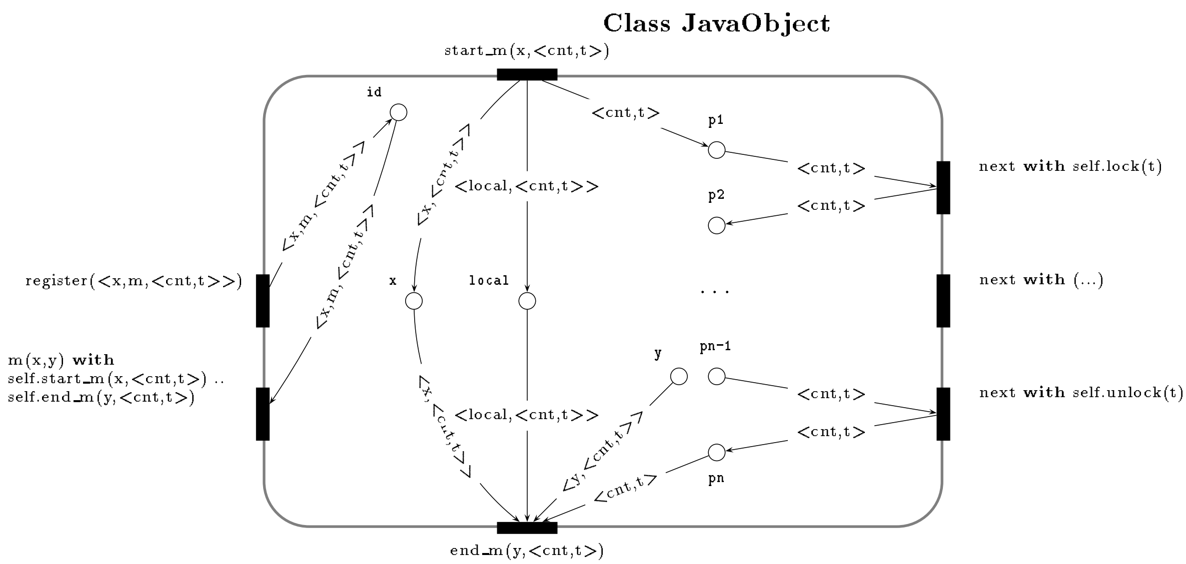


Figure 7.5: CO-OPN/2 Specification of Java Synchronized Methods

The difference with a non synchronized method is that two extra `next` methods are needed: one which is fired just after the `start_m(x,<cnt,t>)` method, and another one which is fired just before the `end_m(y,<cnt,t>)` method. The first `next` is responsible to acquire the lock of object `o` on behalf of thread `t` (calling `self.lock(t)`). The last `next` is responsible to release the lock of object `o` which is in possession of the caller, i.e., `t` (calling `self.unlock(t)`). The specification of the Java method `m` is nested between this pair of `next`. Thus, the method's body can be executed only if the lock has been acquired by the caller's thread, and as soon as the method's body is finished the lock is released.

Remark 7.1.2 *Note that we need both* `cnt` *and* `t` *to discriminate method calls. Indeed, if we use only* `cnt`*, it is not possible to know if a given thread is holding a lock on an object, because the* `cnt` *is unique for every call and does not give indication on the thread which is behind the call. If we use only* `t`*, it is possible to manage the lock problem, but it would be impossible to discriminate two concurrent calls of the same method by the same thread (recursion), even if the method is a* `synchronized` *method.*

Java Synchronized Statements

A Java `synchronized` statement is a more basic construct than `synchronized` method. It is of the form:

```
synchronized(z) { I }
```

where `z` is an object, and `I` is a block of instructions. A `synchronized` statement is always part of the body of a method. In order to execute a `synchronized` statement, a thread has to compete for the lock on the object `z`.

CO-OPN/2 Specifications

A Java method having in its body a `synchronized` statement is specified in the same way as a `synchronized` method, except that the acquisition of the lock does not occur at the beginning of the method's execution, but at the point where the synchronized statement occurs. The lock is released at the end of the synchronized statement and not just before the end of the method.

7.1.6 Java Thread Class

Java threads are created and managed by the classes `Thread` and `ThreadGroup`. Usually, a thread is started with its Java `start()` method, and this method calls the Java `run()` method, which is the "body" of the thread. The thread runs until the `run()` method returns or until the `stop()` method of its `Thread` object is called. The caller's identity is a pair `<cnt,t>` where `t=self` is the own identity of the thread. The propagation of the thread's reference *ends* when a new thread is created, i.e., when a method `start()` is reached in the cascade of method's calls. The reference of the caller is no longer propagated; instead, it is the reference of the newly created thread that is propagated, firstly from its `start()` method to its `run()` method, and subsequently to all the methods that are called from within its `run()` method. It is also possible for a thread to call directly the `run` method of another thread. In this case, the caller's identity is a pair `<cnt,t>` where `t` is the identity of the thread which called the `run` method.

The static methods of the `Thread` class operate on the currently running thread. The instance methods may be called from one thread to operate on a different thread.

CO-OPN/2 Specifications

CO-OPN/2 Class module `JavaThreads` specifies the Java `Thread` class. It defines type `javathread`. Figure 7.6 gives a partial view of the CO-OPN/2 specification of the Java `start()` and `run()` methods. The Java `start()` is a `synchronized` method, thus it is specified accordingly, i.e, the body is embedded into a call to `self.lock(t)` and a call to `self.unlock(t)`.

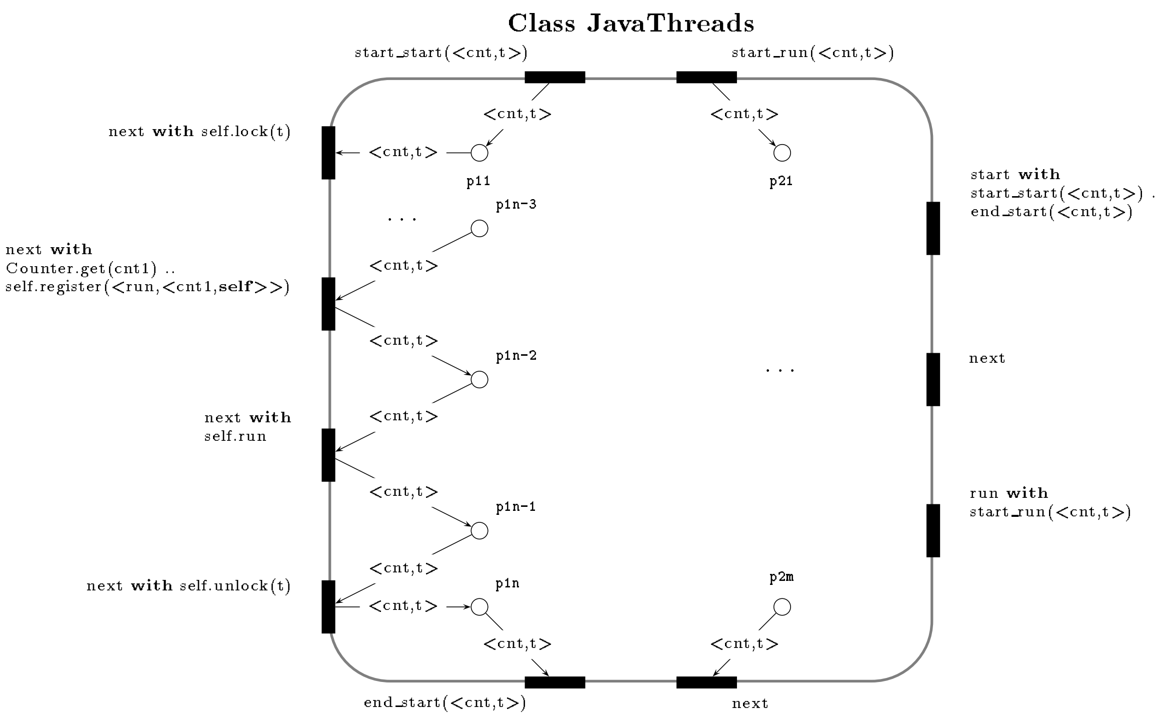


Figure 7.6: CO-OPN/2 Specification of a Java Thread

Just before returning, the `start` method calls the `run` method of the thread which is started, and *breaks* the propagation of thread's reference. Indeed, the registration of the call to method `run` is not made on behalf of thread `t` that called method `start`, but on behalf of the current thread itself. This point is the actual point where a *new execution flow* is started, which will control its own cascade of method calls.

The Java `run()` method is specified like any other Java method, with the particularity of not being a blocking method. Consequently, the caller of the `start()` method is not blocked waiting for the `run()` method to be finished. For this reason, the CO-OPN/2 specification of the Java `run()` method ends with a `next`, called by the Java scheduler.

7.1.7 Java Applet Class

Java applets are piece of code that are moved from one machine to another one. The Java `init()` method is used to perform any one-time initialisation that is necessary when the applet is first created. The Java `start()` method is called by the system. It is like the `init()` method, but it may be called multiple times throughout the applet's life. The Java `stop()` method stops the applet from executing. The Java `destroy()` method frees up any resources that the applet is holding. The Java Virtual machine captures events occurring in the graphical user interface provided by an applet and, in an unobserved manner, invokes method `action(Event e, Object o)` (returning a boolean value) of the applet for the corresponding event. The `Applet()` constructor provided by the Java `Applet` class is a default constructor. All these methods are called by an applet viewer or a Web browser, they are never called by another object.

Remark 7.1.3 *In order to represent the capture of events by the applet, we propose that the mathematical representation, $(Md^{\mathsf{C}}_{\boldsymbol{Prog}})_{\mathrm{MyApplet}}$, of a Java class* ***MyApplet****, a sub-class of Java class* ***Applet****, contains as many methods as the number of events that the applet can handle, even though they are not present in the Java source code.*

CO-OPN/2 Specifications

CO-OPN/2 Class module `JavaApplets` specifies the Java `Applet` class, and defines type `javaapplet`. We model methods `init()`, `start()`, and `stop()`, and `action(e,o,b)` only. Java does not provide any body for these methods, i.e. their body is empty. For this reason, the corresponding CO-OPN/2 specification, depicted in figure 7.7, provides only the pairs of CO-OPN/2 methods: (1) `start_init(id)`, `end_init(id)`; (2) `start_start(id)`, `end_start(id)`; (3) `start_stop(id)`, `end_stop(id)`; (4) `start_action(e,o,id)`, `end_action(b,id)`.

We do not provide a constructor, because the Java constructor of the `Applet` class is a default one, thus the CO-OPN/2 default constructor, `create`, is used for this purpose.

In order to specify the capture of events occurring in the graphical user interface provided by an applet, we propose to add to every applet as many CO-OPN/2 methods as the number of events that the `action()` method is able to handle. These extra methods have no corresponding Java method, they simply enable to observe the interaction of the user with the GUI, and to call, in an unobservable manner, the `action()` for the captured event.

We skip all the other Java methods being part of the Java `Applet` class.

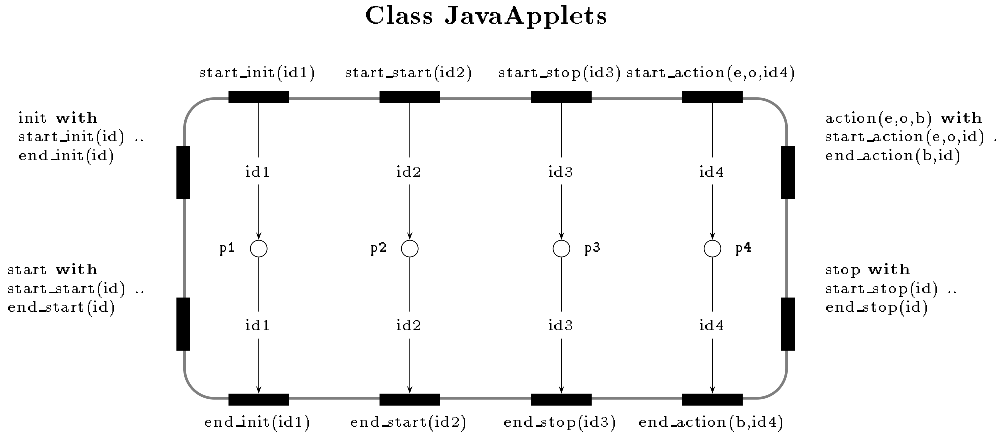


Figure 7.7: CO-OPN/2 Specification of a Java Applet

7.1.8 Java Sockets

The Java Programming language defines several classes to work with sockets, particularly the `ServerSocket` class and the `Socket` class. Two types of communication through a socket are available: (1) a communication based on an underlying reliable connection-based stream protocol; (2) a communication based on an underlying unreliable datagram protocol. A stream protocol is the default.

We focus more precisely on reliable streams. A communication through a socket based on a reliable connection-based stream protocol implies the following: (1) a connection is established between the partners of the communication before any exchange of messages is performed; (2) messages between partners are received in the same order than the order in which they are sent; (3) no message is lost during the communication. More precisely, the establishment of the connection is established in the following way: an instance of `ServerSocket` class is created and waits for socket connections on a given host and a given port. Every instance of `Socket` class is created with the knowledge of the host and the port where the `ServerSocket` instance is waiting. As soon as the `Socket` instance is created the `ServerSocket` accepts (by the means of an `accept()` method) the connection and receives two streams (input and output) to actually send and receive data. The communication is then established.

CO-OPN/2 Specifications

CO-OPN/2 Class module `JavaSockets`, defining type `javasocket`, specifies the Java `Socket` class. The creation of every instance of `JavaSockets` Class module causes the creation of two instances of `JavaArrayBytes` Class module. This Class module specifies a Java array of bytes. One of these queues is used by the client to write information and by the server to read information, while the other one is used by the server to write information and by the client to read information. They stand for the input and output streams. Before returning, the constructor registers to an underlying system the two streams as well as the host and the port where to connect.

CO-OPN/2 Class module `JavaServerSockets`, defining type `javaserversocket`, specifies the Java `ServerSocket` class. The `accept()` method is specified such that it gets registered connections from the underlying system; and returns the input and output streams.

The underlying system is specified as a buffer that stores 4-tuples (two streams, name of host, and port number).

7.1.9 Java Vector Class

Java `Vector` class defines ordered structures storing Java object identifiers. Several methods enable to insert an element at a given position, `insertElementAt(obj,index)`; read an element, `elementAt(i)`; remove an element at a given position, `removeElementAt(obj,index)`.

CO-OPN/2 Specifications

CO-OPN/2 Class module `JavaVectors` defines type `javavector` and specifies the Java `Vector` class. It is specified as an array of Java objects.

7.2 Running Example

Running example of Chapter 5 shows the refinement of contractual CO-OPN/2 specification $CSpec_0$ (see Example 5.2.8), defining a heap of normal chocolate packaging, by a contractual CO-OPN/2 specification $CSpec_1$ (see Example 5.2.14), defining a FIFO of normal and deluxe chocolate packaging. Chapter 6 gives a contractual Java program $CProg_1$ (see Example 6.1.24) implementing $CSpec_1$ and hence $CSpec_0$.

The purpose of this section is to define $CSpec_2$ that refines $CSpec_1$ and which is very close to contractual program $CProg_1$. In addition, it gives the refine relation on $CSpec_1$ and $CSpec_2$, and the implement relation on $CSpec_2$ and $Prog_1$.

Contractual CO-OPN/2 Specification $CSpec_2$

Figure 7.8 depicts a part of the CO-OPN/2 specification of the Java class `JavaConveyorBelt` given by Figure 6.2. It depicts only methods `insertElement()` and `removeElement()`. Since Java class `JavaConveyorBelt` class extends Java `Vector` class and hence the Java `Object` class, methods `insertElementAt(box)`, `removeElementAt()`, `size()`, and `wait()`, notify(), etc., are also available. The CO-OPN/2 Class module JavaConveyorBelt defines the java-conveyor-belt type and the static object the-java-conveyor-belt.

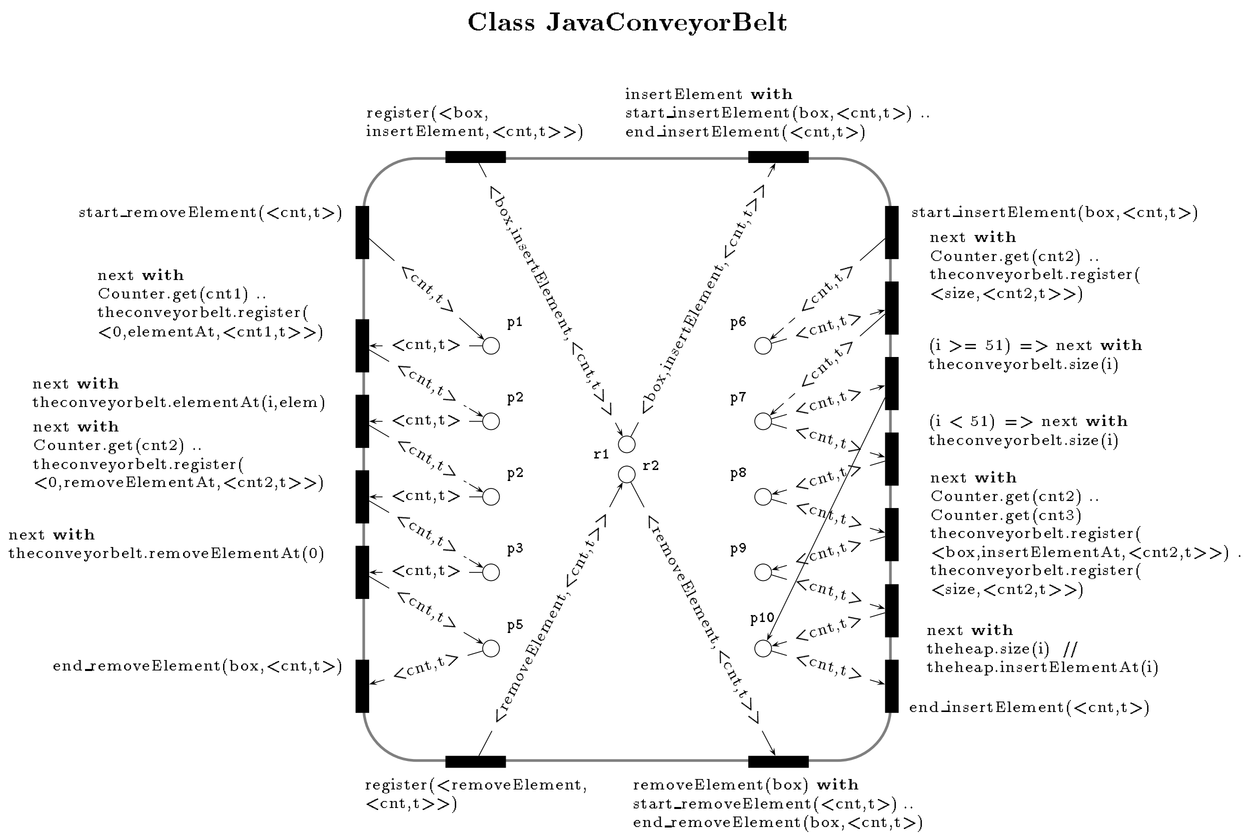


Figure 7.8: The CO-OPN/2 Specification of Java Class JavaConveyorBelt

Left part of Figure 7.8 shows method removeElement(), while right part shows insertElement(). Their specification follows from Subsection 7.1.3, i.e., every instruction of the Java method's body is specified using CO-OPN/2 next methods. It is just interesting to note the specification of the test theconveyorbelt.size()<51 (ligne 6 of Figure 6.2). Two next methods have been used, (second and third next methods on the right): one for ending immediately the method (by enabling the firing of method end_insertElement), and the other one for continuing with the next instruction (by enabling the firing of the fourth next method).

It is worth noting that between ligne 6 and ligne 7, as well as between ligne 12 and ligne 13 of Figure 6.2, a lot of other Java instructions may occur. This is particularly visible on the CO-OPN/2 specification, since other next methods can be fired between the fourth and the fifth next, on the right of Figure 7.8; and between the second and the fourth next on the left of Figure 7.8. Thus, for the left part, even though we think that we are actually removing element 0, it can happen that element 0 has already been removed and replaced by some other element, or even worse, all elements have been removed and there is no element at position 0. This cause no problem if only one flow of control exists. Otherwise, method `removeElement()` and `insertElement()` should be declared as `synchronized` in the Java class `JavaConveyorBelt`.

Similarly we define the CO-OPN/2 specification of Java `JavaPackaging` and `JavaDeluxePackaging`. They defines the `java-packaging` and `java-deluxe-packaging` types respectively. The CO-OPN/2 ADT module `Booleans` and `Integers` specify the Java `boolean` and the Java `int` types respectively.

Contractual CO-OPN/2 specification $CSpec_2 =< Spec_2, \Phi_2 >$ is such that:

$$\begin{aligned} Spec_2 = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Integers}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaObject}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaVectors}}, \\ & (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaPackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaDeluxePackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaConveyorBelt}}\}. \end{aligned}$$

The contract Φ_2 is similar to the contract Ψ_1 of contractual program $CProg_1$, variables are given by the set:

$$X_2 = \{javapack_1, \ldots, javapack_{51}\}_{\text{javapackaging}} \cup \{javadeluxepack\}_{\text{javadeluxepackaging}}.$$

$$\begin{aligned}
\phi_{2_1} = \; & <javapack_1.\text{create}><\text{the-java-conveyor-belt . insertElement}(javapack_1)> \\
& <\text{the-java-conveyor-belt . removeElement}(javapack_1)> \mathbf{T} \\
\phi_{2_2} = \; & \neg(<javapack_1.\text{create}> \\
& <\text{the-java-conveyor-belt . removeElement}(javapack_1)> \mathbf{T}) \\
\phi_{2_3} = \; & <javapack_1.\text{create}><javapack_1.\,\text{fill}_{\text{java-packaging}}(\text{true})> \mathbf{T} \\
\phi_{2_4} = \; & <javapack_1.\text{create}><javapack_2.\text{create}> \\
& <\text{the-java-conveyor-belt . insertElement}(javapack_1)> \\
& <\text{the-java-conveyor-belt . insertElement}(javapack_2)> \\
& (<\text{the-java-conveyor-belt . removeElement}(javapack_1)> \\
& <\text{the-java-conveyor-belt . removeElement}(javapack_2)> \wedge \\
& \neg(<\text{the-java-conveyor-belt . removeElement}(pack_2)> \\
& <\text{the-java-conveyor-belt . removeElement}(pack_1)>))\mathbf{T} \\
\psi_{2_5} = \; & <javapack_1.\text{create}> \ldots <javapack_{50}.\text{create}><javapack_{51}.\text{create}> \\
& <\text{the-java-conveyor-belt . insertElement}(javapack_1)> \ldots \\
& <\text{the-java-conveyor-belt . insertElement}(javapack_{50})> \\
& \neg(<\text{the-java-conveyor-belt . insertElement}(javapack_{51})>)\mathbf{T} \\
\phi_{2_6} = \; & <javadeluxepack.\text{create}><javadeluxepack.\,\text{fill}_{\text{java-deluxe-packaging}}(\text{false})> \\
& <javadeluxepack.\,\text{fill}_{\text{java-deluxe-packaging}}(\text{true})> \mathbf{T} \\
\phi_{2_7} = \; & <\text{the-java-conveyor-belt . notify}> \mathbf{T}.
\end{aligned}$$

Refine Relation

The refine relation on $CSpec_1$ and $CSpec_2$ is obviously given by λ_1 below. It is very similar to the implement relation λ_1^I on $CSpec_1$ and $CProg_1$ (see Example 6.2.4), since contractual CO-OPN/2 specification $CSpec_2$ is meant to replace contractual program $CProg_1$.

$$\begin{aligned}
\lambda_{1_{SA}} =& \{(\text{chocolate}, \text{boolean}), (\text{praline}, \text{boolean}), (\text{truffle}, \text{boolean})\} \\
\lambda_{1_{SC}} =& \{(\text{packaging}, \text{java-packaging}), (\text{deluxe-packaging}, \text{java-deluxe-packaging}), \\
& (\text{conveyor-belt}, \text{java-conveyor-belt})\} \\
\lambda_{1_{FA}} =& \{(\text{P}_{\text{praline}}, \text{true}_{\text{boolean}}), (\text{T}_{\text{truffle}}, \text{false}_{\text{boolean}})\} \\
\lambda_{1_{FC}} =& \{(\text{new}_{\text{conveyor-belt}}, \text{new}_{\text{java-conveyor-belt}}), (\text{init}_{\text{conveyor-belt}}, \text{init}_{\text{java-conveyor-belt}}), \\
& (\text{new}_{\text{packaging}}, \text{new}_{\text{java-packaging}}), (\text{init}_{\text{packaging}}, \text{init}_{\text{java-packaging}}), \\
& (\text{new}_{\text{deluxe-packaging}}, \text{new}_{\text{java-deluxe-packaging}}), (\text{init}_{\text{deluxe-packaging}}, \text{init}_{\text{java-deluxe-packaging}})\} \\
\lambda_{1_M} =& \{(\text{put}_{\text{conveyor-belt,packaging}}, \text{insertElement}_{\text{java-conveyor-belt,java-packaging}}), \\
& (\text{get}_{\text{conveyor-belt,packaging}}, \text{removeElement}_{\text{java-conveyor-belt,java-packaging}}), \\
& (\text{fill}_{\text{packaging,chocolate}}, \text{fill}_{\text{java-packaging,boolean}}), \\
& (\text{fill}_{\text{deluxe-packaging,chocolate}}, \text{fill}_{\text{java-deluxe-packaging,boolean}})\} \\
\lambda_{1_O} =& \{(\text{the-conveyor-belt}, \text{the-java-conveyor-belt})\} \\
\lambda_{1_X} =& \{(pack_i, javapack_i)\ (1 \leq i \leq 51), (dpack, javadeluxepack)\}.
\end{aligned}$$

Implement Relation

The implement relation on $CSpec_2$ and $CProg_1$ is given by λ_2^I below. It is just a renaming of the type, sort, method and object names of $CSpec_2$ into respective names of $CProg_1$.

$$
\begin{aligned}
\lambda^I_{2_{SA}} =& \{(\text{boolean}, \text{boolean})\} \\
\lambda^I_{2_{SC}} =& \{(\text{javaobject}, \text{Object}), (\text{javavector}, \text{Vector}), \\
& (\text{java-packaging}, \text{JavaPackaging}), (\text{java-deluxe-packaging}, \text{JavaDeluxePackaging}), \\
& (\text{java-conveyor-belt}, \text{JavaConveyorBelt})\} \\
\lambda^I_{2_{FA}} =& \{(\text{true}_{\text{boolean}}, \text{true}_{\text{boolean}}), (\text{false}_{\text{boolean}}, \text{false}_{\text{boolean}})\} \\
\lambda^I_{2_{FC}} =& \{(\text{new}_{\text{javaobject}}, \text{new}_{\text{Object}}), (\text{init}_{\text{javaobject}}, \text{init}_{\text{Object}}), \\
& (\text{new}_{\text{javavector}}, \text{new}_{\text{Vector}}), (\text{init}_{\text{javavector}}, \text{init}_{\text{Vector}}), \\
& (\text{new}_{\text{java-conveyor-belt}}, \text{new}_{\text{JavaConveyorBelt}}), (\text{init}_{\text{java-conveyor-belt}}, \text{init}_{\text{JavaConveyorBelt}}), \\
& (\text{new}_{\text{java-packaging}}, \text{new}_{\text{JavaPackaging}}), (\text{init}_{\text{java-packaging}}, \text{init}_{\text{JavaPackaging}}), \\
& (\text{new}_{\text{java-deluxe-packaging}}, \text{new}_{\text{JavaDeluxePackaging}}), \\
& (\text{init}_{\text{java-deluxe-packaging}}, \text{init}_{\text{JavaDeluxePackaging}})\} \\
\lambda^I_{2_{M}} =& \{(\text{insertElement}_{\text{java-conveyor-belt,java-packaging}}, \text{insertElement}_{\text{JavaConveyorBelt,java-packaging}}), \\
& (\text{removeElement}_{\text{java-conveyor-belt,java-packaging}}, \\
& \text{removeElement}_{\text{JavaConveyorBelt,java-packaging}}), \\
& (\text{fill}_{\text{java-packaging,boolean}}, \text{fill}_{\text{JavaPackaging,boolean}}), \\
& (\text{fill}_{\text{java-deluxe-packaging,boolean}}, \text{fill}_{\text{JavaDeluxePackaging,boolean}})\} \\
\lambda^I_{2_{O}} =& \{(\text{the-java-conveyor-belt}, \text{theconveyorbelt})\} \\
\lambda^I_{2_{X}} =& \{(javapack_i, javapack_i)\ (1 \le i \le 51), (javadeluxepack, javadeluxepack)\}.
\end{aligned}
$$

Remark 7.2.1 *We have that λ_1^I of Example 6.2.4 is actually equal to the composition $\lambda_1 \ ; \lambda_2^I$.*

7.3 Advices for Implementing in Java

The CO-OPN/2 specifications language and the Java programming language share some similarities essentially because they are both object-oriented. However, they differ by several points: ADT modules cannot be defined in Java; every Java class is sub-class of the Java `Object` class; constructors behave differently in Java and in CO-OPN/2; etc. In order to conduct a refinement process towards a Java implementation, it is necessary to act with caution during the refinement process. Otherwise, the implementation theory defined in Chapter 6 does not apply. This section lists some points that should be respected in order to correctly and easily perform the implementation phase.

Refinement process ends with CO-OPN/2 specifications of Java program

Contrarily to the other points below, this point is more an advice than an obligation. Ending the refinement process with a contractual CO-OPN/2 specification entirely built with CO-OPN/2 classes specifying Java classes has the following advantages. First, the implementation is trivially performed, since every instruction of the program is already specified. Second, the Java program will behave like the most concrete contractual CO-OPN/2 specification. Thus, no unexpected behaviour arises during the implementation phase, since it has already been observed at the CO-OPN/2 specification level. Consequently, the contract of the most concrete contractual CO-OPN/2 specification is preserved by the program, and this ensures that the program is a correct implementation. Section 7.2 evidences the following fact: methods `inserElement()` and `removeElement()` of Class module `JavaConveyorBelt` are not specified as Java `synchronized` methods, and this can cause errors in the case of multiple flows of control.

CO-OPN/2 ADT modules

According to the implement relation given in Definition 6.2.2, CO-OPN/2 ADT terms *appearing in the contract* have to be related to a term of a Java primitive type. The Java primitive types are: `int`,`long`, etc. Since this list is very restricted, it is not possible to relate any CO-OPN/2 ADT term to a term of one of these types. For this reason, it is necessary to avoid the use of complex ADT modules that cannot be related to Java primitive types and to use instead a Class module that wraps it.

However, for CO-OPN/2 ADT terms that *does not appear in contract*, it is not necessary to wrap them into a Class module. For instance, the CO-OPN/2 `ConveyorBelt` Class module (see Example 5.2.14) uses ADT module `FifoPackaging` internally, and no formula of the contract concerns this module. Therefore, it is not necessary to wrap it in a Class module.

Constructors

CO-OPN/2 implement relation states that CO-OPN/2 default constructors are related to Java default constructors. Most of the time, a default constructor is not sufficient, and a Java class defines as well non-default constructors. Therefore, we recommend to use non-default CO-OPN/2 constructors very early in the refinement process, even though a default constructor is sufficient.

Systems and JVM

A software system is always starting at a given moment. When the system is implemented in Java, the start of the system corresponds to the invocation of command `java`

`ClassName args` which starts the `main(args)` method of Java class `ClassName`. CO-OPN/2 Class module `JVM` (see Figure 7.1) specifies the Java Virtual Machine, a method `java(ClassName,args)`, and the call to the `main(args)` method. When a whole system has to be specified, we recommend to use a method `init(Name,args)` from the most abstract contractual CO-OPN/2 specification. Method `init(Name,args)` is refined to method `java(ClassName,args)`, and finally implemented with command `java ClassName args`. An example of use of method `init(Name,args)` is provided by the case study, described in Chapter 9.

Graphical User Interfaces

We have treated Java Graphical User Interfaces (GUIs) in a particular way. Additional methods are used both in the abstract definition of a program using GUI and in the CO-OPN/2 specification of the program. These methods enable to capture events occurring of the interaction of the user with the GUI, and invoke the corresponding method (`action()`) of the Java object handling the event. These methods are not methods appearing in the source program.

Chapter 8

Verifying Refinement and Implementation using Test Generation

---

Chapters 5 and 6 develop respectively a theory of stepwise refinement, and a theory of implementation of contractual CO-OPN/2 specifications, which are based on contracts expressed using HML formulae. The use of HML formulae is motivated by the fact that they are currently employed in the theory of test generation developped for CO-OPN/2. The purpose of the current chapter is to propose a means, using this theory of test generation, to practically verify: that a set of HML formulae expressed on a CO-OPN/2 specification is actually a contract (horizontal verification); that refinement steps are correct (vertical verification); and that the implementation phase is correct too (program verification).

The theory of test enables to generate a reduced test set, representative of the *whole* behaviour of a CO-OPN/2 specification, such that if the model of a program satisfies the test set, then this model is bisimular to that of the specification. In the theory of refinement and implementation by contracts, we need only to test if the model of the program is bisimulable to that of the specification on *the part specified by the contract.* Therefore, the basic idea for applying the theory of test for verifying a refinement step, consists of generating test sets on the basis of the contract, instead of the whole set of formulae satisfied by the model of the specification.

This chapter first presents the theory of test generation, then it explains the use of test generation for the horizontal verification, for the vertical verification, and finally for the program verification.

8.1 Introduction to Test Generation

The theory of test, developped by Barbey, Buchs and Péraire in [12, 11, 52], generates a *minimal* set of test cases able to ensure that if a program satisfies the test set, then the program satisfies its specification, i.e., the model of the program is *bisimular* to that of the specification. Test cases are pairs made of a HML formula and a boolean value. Bisimulation is easily provided since HML formulae are able to discriminate models as finely as the bisimulation equivalence. The minimal set of test cases is obtained at the end of a *test selection process* that starts with an *exhaustive* test set and reduces it by applying a series of *reduction hypotheses* on the program. The theory of test generation is completed by a tool that generates test cases from a given CO-OPN/2 specification.

This section briefly introduces some preliminary definitions, the theory of formal testing, the test selection process, and finally the practical test selection.

8.1.1 Preliminary Definitions

A test case is a pair made of a HML formula and a value, either $true$ or $false$. A set of such test cases is called a test set.

Definition 8.1.1 *Test Cases and Test Sets.*
A test case is a pair $\langle f, r\rangle$, *where* $f \in \textsc{Prop}$ *is a ground HML formula, and* $r \in \{true, false\}$.

A test set is a set of test cases.

Notation 8.1.2 *Test Sets.*
We denote TEST *the class of all possible sets of test cases.*

A test set is satisfied by a program if for every test case $\langle f, r\rangle$ of the test set, the transition system of the program satisfies f iff $r = true$. The satisfaction relationship $\models_O$ on programs and test sets states in which cases a test set is satisfied by a program. Sub-script O stands for the *oracle*, which is a decision procedure that verifies if a program satisfies the test set.

Definition 8.1.3 *Satisfaction Relationship on Programs and Tests.*
The satisfaction relationship on programs and tests, noted $\models_O \subseteq \textsc{Prog} \times \textsc{Test}$, *is such that:*

$$
\begin{aligned}
(Prog \models_O T) \quad \Leftrightarrow \quad & (\forall \langle f, r\rangle \in T \; then \\
& Mod_{Prog} \models_{HML} f \; and \; r = true \; or \\
& Mod_{Prog} \not\models_{HML} f \; and \; r = false) \, ,
\end{aligned}
$$

where $Prog \in$ PROG is a program, Mod_{Prog} is the transition system of $Prog$, $T \in$ TEST is a test set, and $\models_{HML}$ is the HML satisfaction relation given by Definition 6.1.15.

8.1.2 Formal Testing

The aim of formal testing, as defined by Barbey, Buchs, and Péraire in [12, 11, 52], is to find a test set such that if a program *Prog* satisfies the test set, then the program satisfies its specification *Spec*, noted $Prog \models Spec$. *Prog* satisfies a given specification *Spec* if *Prog* is bisimular to *Spec*.

Definition 8.1.4 *Bisimulation.*
Let TS_1, TS_2 be two transition systems, $State_{TS_1}$, $State_{TS_2}$ be the set of states of TS_1, TS_2 respectively, and $Init_1$, $Init_2$ be the initial state of TS_1, TS_2 respectively. TS_1 is bisimular to TS_2, noted $TS_1 \triangleq TS_2$, if there is a relation $R \subseteq State_{TS_1} \times State_{TS_2}$ such that:

1. $Init_1 \; R \; Init_2$
2. *If* $st_1 \; R \; st_2$ *and* $(st_1, e, st_1') \in TS_1$ *then* $\exists \, (st_2, e, st_2') \in TS_2$ *s.t.* $st_1' \; R \; st_2'$
3. *If* $st_1 \; R \; st_2$ *and* $(st_2, e, st_2') \in TS_2$ *then* $\exists \, (st_1, e, st_1') \in TS_1$ *s.t.* $st_1' \; R \; st_2'$.

The relation R is called a strong bisimulation.

Definition 8.1.5 *Satisfaction Relationship on Programs and Specifications.*
The satisfaction relationship on programs and specifications, noted $\models \subseteq$ PROG $\times$ SPEC, is such that:

$$Prog \models Spec \quad \Leftrightarrow \quad Mod_{Prog} \triangleq SSem_A(Spec) \text{ and there is signature morphism between the global signature of } Prog \text{ and the global signature of } Spec.$$

This definition implies that the set of events of the transition system of *Prog* is the same as the set of events of the transition system (i.e., step semantics) of *Spec*.

Given *Prog*, a program and *Spec*, a specification, the aim of formal testing is to find a test set T such that:

$$(Prog \models Spec) \;\Leftrightarrow\; (Prog \models_O T). \qquad (i)$$

Such a test set is called *pertinent*.

Test cases are built with HML formulae, two transition systems are equivalent iff they satisfy the same set of HML formulae.

Definition 8.1.6 *HML Equivalence.*
The HML equivalence relationship, noted $\sim_{HML} \subseteq \mathsf{TS} \times \mathsf{TS}$, *is such that:*

$$(TS_1 \sim_{HML} TS_2) \quad \Leftrightarrow \quad (\forall f \in \textsc{Prop},\ TS_1 \models_{HML} f \ \Leftrightarrow TS_2 \models_{HML} f),$$

where $TS_1, TS_2 \in \mathsf{TS}$ *are two transition systems.*

The full agreement theorem, proved by Hennessy and Milner in [41], shows that HML formulae distinguish image-finite[1] transition systems as finely as the bisimulation equivalence. Indeed, it underscores the fact that two transition systems are bisimular iff they satisfy the same set of HML formulae.

Theorem 8.1.1 *Full Agreement.*
Let TS_1, TS_2 *be two transition systems, then the following holds:*

$$(TS_1 \triangleq TS_2) \quad \Leftrightarrow \quad (TS_1 \sim_{HML} TS_2).$$

Given a specification $Spec$, the exhaustive test set derived from $Spec$ is given by the *whole* set of test cases satisfied or not by the step semantics of $Spec$. H_O is a set of hypotheses, called the *oracle hypotheses*, ensuring that the oracle knows how to decide the success or the failure of a test case.

Definition 8.1.7 *Exhaustive Test Set.*
Let Spec be a CO-OPN/2 specification, $SSem_A(Spec)$ *be the step semantics of Spec, and* H_O *the oracle hypotheses. The exhaustive test set, noted* $\textsc{Exhaust}_{Spec,H_O} \in \textsc{Test}$, *is a test set such that:*

$$\begin{aligned}\textsc{Exhaust}_{Spec,H_O} = \{\langle f, r\rangle \in \textsc{Prop} \times \{true, false\} \mid \\ (SSem_A(Spec) \models_{HML} f \ and\ r = true) or \\ (SSem_A(Spec) \not\models_{HML} f \ and\ r = false)\}.\end{aligned}$$

The full agreement theorem enables to conclude that if a program $Prog$ satisfies the exhaustive test set of a specification $Spec$ then the program satisfies the specification $Spec$:

$$(Prog \text{ satisfies } H_O) \Rightarrow (Prog \models Spec \ \Leftrightarrow\ Prog \models_O \textsc{Exhaust}_{Spec,H_O}). \qquad (ii)$$

Thus, thanks to the full agreement theorem, the exhaustive test set $T = \textsc{Exhaust}_{Spec,H_O}$ is a test set that let formula (i) be true.

[1] a transition system is image-finite if every reachable state of the transition system has a finite number of successor states.

8.1.3 Test Selection

In order to verify if a program *Prog* satisfies a specification *Spec*, it suffices to prove formula (*ii*). However, $\text{EXHAUST}_{Spec,H_O}$ is a huge set. Therefore, additional hypotheses are made *on the program* in order to reduce the size of the test set.

More generally, given an initial test context (H_0, T_0), i.e., a pair made of a small set of hypotheses H_0 and a huge test set T_0, an *iterative refinement* of the test context is performed, in order to reach a new test context (H_n, T_n) with a bigger set of hypotheses and a smaller test set. We use the term iterative refinement as it has been used in [11]. Thus, it must not be confused with the refinement of specifications as defined in this thesis.

The *iterative refinement* of the test context leads to a chain of test contexts:

$$(H_0, T_0), \ldots, (H_n, T_n)$$

such that $H_{i-1} \subseteq H_i$ and $T_{i-1} \supseteq T_i$, $(1 \leq i \leq n)$, and:

$$(Prog \text{ satisfies } H_{i+1}) \Rightarrow (Prog \text{ satisfies } H_i) \text{ and}$$
$$(Prog \models_O T_i \Leftrightarrow Prog \models_O T_{i+1}) \quad (0 \leq i \leq n-1).$$

By transitivity, the following proposition holds:

Proposition 8.1.1 *Iterative refinement of the Test Context.*
Let Prog be a program. Let $(H_0, T_0), \ldots, (H_n, T_n)$ *be a chain of test contexts such that* $H_{i-1} \subseteq H_i$ *and* $T_{i-1} \supseteq T_i$*,* $(1 \leq i \leq n)$*. The following holds:*

$$(Prog \text{ satisfies } H_n) \Rightarrow (Prog \models_O T_0 \Leftrightarrow Prog \models_O T_n).$$

Thus, in order to reduce the exhaustive test set of a specification *Spec*, an iterative refinement is performed on the initial test context $(H_O, \text{EXHAUST}_{Spec,H_O})$. It leads to the test context, noted $(H, T_{Spec,H})$, where $H = H_O \cup H_R$, and H_R is an additional set of reduction hypotheses.

The theory of test generation uses exclusively *pertinent* test sets, i.e., a program satisfies the test set iff it satisfies the specification. Thus, due to Proposition 8.1.1, formula (*ii*) above becomes:

$$(Prog \text{ satisfies } H) \Rightarrow (Prog \models Spec) \Leftrightarrow (Prog \models_O T_{Spec,H}). \qquad (iii)$$

In order to prove that the program *Prog* is bisimular to the specification *Spec*, it suffices to prove that *Prog* satisfies the hypotheses H and the test set $T_{Spec,H}$.

Remark 8.1.8 *Since in practice, it is difficult to verify the hypotheses* H*, a weaker result is actually reached. If Prog* $\not\models_O T_{Spec,H}$ *then we are sure that the Prog does not satisfy*

Spec. If program Prog $\models_O T_{Spec,H}$*, this actually means that there is no test case in* $T_{Spec,H}$ *such that Prog does not satisfy it. However, since hypotheses* H *are not formally proved, it is not excluded that Prog does not satisfy some test case of the exhaustive test set. Therefore, in the case of success, i.e., Prog* $\models_O T_{Spec,H}$*, we can only be confident that Prog* $\models$ *Spec.*

8.1.4 Practical Test Selection

In order to practically derive a test set having a reasonable size, the test selection process starts from the set $\text{EXHAUST}_{Spec,H_O}$ and retains the minimum set of test cases representative enough to guarantee that all cases are covered, provided some hypotheses, H, are satisfied. The set $\text{EXHAUST}_{Spec,H_O}$ is not explicitly constructed, it is replaced by a set made of exactly one test case $\langle f, r\rangle$ where f is a variable that stands for every HML formula, and r is a variable that stands for *true* or *false*.

During the test selection process, *uniformity* and *regularity* hypotheses are stated *on the program* so that the set $\{\langle f, r\rangle\}$ is progressively replaced by a set of formulae with variables. Finally, *subdomain decomposition* is performed, and a set of ground formulae is obtained.

Uniformity hypotheses make the assumption that if a test containing a variable holds for one instantiation of this variable, then the test holds for every instantiation of this variable. Variables, appearing in HML formulae used for test purposes, have a slightly different meaning than those used for contracts. In a test case, variables stand for any possible term, while in a contract, variables are existentially quantified.

Regularity hypotheses make the assumption that if a test is successful for terms having a complexity (number of events, depth, and occurrences of a method) less or equal to certain bounds, then the test is successful for every term whatever its complexity.

Subdomain decomposition consists of establishing disjoint sets of terms, and of applying reduction hypotheses for every domain.

Péraire [52] has completed the theory of test generation for CO-OPN/2 specifications with a tool able to generate reduced sets of test cases.

8.2 Horizontal Verification

The aim of horizontal verification consists of showing that a CO-OPN/2 specification *Spec*, and a set of HML formulae Φ, expressed on the specification, actually form a contractual CO-OPN/2 specification, i.e., $\text{MOD}_{Spec} \models \Phi$ (see Definition 5.2.1). In this case, the specification itself *is* the program to test.

In the theory of test generation, test selection process is applied to the exhaustive test set $\textsc{Exhaust}_{Spec,H_O}$, made of all HML formulae satisfied by the model of the specification, as well as all HML formulae not satisfied by the model of the specification (see Definition 8.1.7). Therefore, this exhaustive test set corresponds to:

$$\textsc{Exhaust}_{Spec,H_O} = \{\langle f, r\rangle \in \textsc{Prop} \times \{true, false\} \mid f \in \Phi_{Spec} \text{ and } r = true\}.$$

Remember that Φ_{Spec} is the set of *all* HML formulae satisfied by the model of *Spec* (see Definition 5.2.1). Negative formulae of Φ_{Spec} correspond to the formulae that the model must not satisfy. Without loss of generality, we assume that contracts are made only of ground HML formulae. Indeed, first the set $\textsc{Exhaust}_{Spec,H_O}$ as defined in the theory of test generation is a set of ground HML formulae, and second, variables are used in contracts only to alleviate the work of the specifier, and are existentially quantified. If a contract contains HML formulae with variables, these formulae can be replaced by ground formulae.

For horizontal verification, the test selection process starts with an exhaustive set of test cases built from Φ, the contract to verify, instead of Φ_{Spec}. This set is exhaustive wrt Φ, but not wrt the whole specification.

Definition 8.2.1 *Exhaustive Test Set of CSpec.*
Let $CSpec = \langle Spec, \Phi\rangle$ be a pair made of a CO-OPN/2 specification Spec, and a set of HML formulae Φ. Let $SSem_A(Spec)$ be the step semantics of Spec, and H_O a set of oracle hypotheses. The exhaustive test set of CSpec, noted $\textsc{Exhaust}_{CSpec,H_O} \in \textsc{Test}$*, is a test set such that:*

$$\textsc{Exhaust}_{CSpec,H_O} = \{\langle f, r\rangle \in \textsc{Prop} \times \{true, false\} \mid f \in \Phi \textit{ and } r = true\}.$$

We state that the initial test context is $(H_O, \textsc{Exhaust}_{CSpec,H_O})$.

The iterative refinement of test context is applied, i.e., additional hypotheses are made on *Spec*, and a smaller test set is generated from $\textsc{Exhaust}_{CSpec,H_O}$. The test context reached after this process is noted $(H, T_{CSpec,H})$.

Applying Proposition 8.1.1 to CO-OPN/2 specifications provides the following result.

Proposition 8.2.1 *Iterative refinement of the Test Context.*
Let $CSpec = \langle Spec, \Phi\rangle$ be a pair made of a CO-OPN/2 specification Spec, and a set of HML formulae Φ. Let $\textsc{Exhaust}_{CSpec,H_O}$ be the exhaustive test set of CSpec, and $T_{CSpec,H}$ be the test set generated from $\textsc{Exhaust}_{CSpec,H_O}$. The following holds:

$$(Spec \textit{ satisfies } H) \Rightarrow (Spec \models_O \textsc{Exhaust}_{CSpec,H_O} \Leftrightarrow Spec \models_O T_{CSpec,H}).$$

Since the exhaustive test set is trivially built from Φ, the following corollary, following from Proposition 8.2.1, enables to conclude that satisfying the test is equivalent to satisfying Φ.

Corollary 8.2.1 *Horizontal verification.*
Let $CSpec = \langle Spec, \Phi\rangle$ *be a pair made of a CO-OPN/2 specification Spec, and a set of HML formulae* Φ*. Let* $\textsc{Exhaust}_{CSpec,H_O}$ *be the exhaustive test set of* $CSpec$*, and* $T_{CSpec,H}$ *be the test set generated from* $\textsc{Exhaust}_{CSpec,H_O}$*. The following holds:*

$$(Spec \text{ satisfies } H) \Rightarrow (Spec \models_O T_{CSpec,H} \Leftrightarrow \textsc{Mod}_{Spec} \models \Phi).$$

Proof.
Proposition 8.2.1 provides (1) below. $\textsc{Exhaust}_{CSpec,H_O}$ is built from Φ by creating from every formula f of the contract a test case $\langle f, true\rangle$.
By definition of $\models_O$: $(Spec \models_O \langle f, true\rangle) \Leftrightarrow \textsc{Mod}_{Spec'} \models f$. This provides (2) below:

$$\begin{aligned}(Spec \text{ satisfies } H) &\overset{1}{\Rightarrow} (Spec \models_O T_{CSpec,H} \Leftrightarrow Spec \models_O \textsc{Exhaust}_{CSpec,H_O}) \\ &\overset{2}{\Rightarrow} (Spec \models_O T_{CSpec,H} \Leftrightarrow \textsc{Mod}_{Spec} \models \Phi).\end{aligned}$$

■

Corollary 8.2.1 enables to conclude that if a specification $Spec$ satisfies hypotheses H and the test set $T_{CSpec,H}$, then $CSpec = \langle Spec, \Phi\rangle$ is actually a contractual CO-OPN/2 specification.

Remark 8.2.2 *When the set of HML formulae to test is* Φ_{Spec}*, then the exhaustive set of the specification CSpec is the same as the one obtained in the theory of test for Spec, i.e., when* $\textsc{Exhaust}_{CSpec,H_O} = \textsc{Exhaust}_{Spec,H_O}$*. Consequently, the iterative refinement of the test contexts provides the same minimal test set, i.e.,* $T_{CSpec,H} = T_{Spec,H}$*.*

Practical Generation of Test Sets

We have seen in Subsection 8.1.4 that in order to construct $T_{Spec,H}$ from $\textsc{Exhaust}_{Spec,H_O}$ in practice, the set $\textsc{Exhaust}_{Spec,H_O}$ is is replaced by a set made of exactly one test case $\langle f, r\rangle$ where f is a variable that stands for every HML formula, and r is a variable that stands for $true$ or $false$. In order to practically construct $T_{CSpec,H}$ from $\textsc{Exhaust}_{CSpec,H_O}$ a similar procedure must be contemplated: one or more HML formulae with variables replace $\textsc{Exhaust}_{CSpec,H_O}$. In that case variables are universally quantified, since the theory of test generation uses universally quantified variables.

8.3 Vertical Verification

The aim of vertical verification is to assert if a given refinement step is correct. We intend to use the theory of test generation in order to verify the correctness of a refinement step made of $CSpec = \langle Spec, \Phi\rangle$ an abstract contractual CO-OPN/2 specification, and

$CSpec' = \langle Spec', \Phi' \rangle$ a concrete contractual CO-OPN/2 specification, i.e., we want to verify if $CSpec \sqsubseteq CSpec'$. $CSpec'$ plays the role of the program (of the theory of test), and $CSpec$ that of the specification.

Two cases must be distinguished. First, the contracts are *partial*, i.e., $\Phi \subset \Phi_{Spec}$. Second, the contracts are *total*, i.e., $\Phi = \Phi_{Spec}$.

When the contract is partial, test generation theory must be applied in a way such that the preservation of the contract in subsequent refinement steps is ensured. We show that a lower-level contractual specification refines a higher-level contractual specification if: it satisfies the test set generated from the exhaustive test set of the higher-level contractual specification, and if its own generated test set is part of the exhaustive test set of the higher-level contractual specification.

As we have alredy noticed in the case of horizontal verification, when the contract is total, the theory of test generation applies directly, since $\text{EXHAUST}_{CSpec,H_O} = \text{EXHAUST}_{Spec,H_O}$, and we show that if a lower-level contractual specification satisfies the test set generated from the exaustive test set of a higher-level contractual specification, then the lower-level contractual specification correctly refines the higher-level contractual specification.

This section presents the vertical verification, first in the case of partial contracts, and second, in the case of total contracts.

8.3.1 Partial Contract

The theory of refinement based on contracts allows a concrete contractual specification to refine an abstract contractual specification *without* their respective specification parts being bisimular. This is the case when the contracts are strict subsets of the whole set of HML formulae satisfied by the step semantics of the specifications.

Therefore, the initial text context cannot be $\text{EXHAUST}_{Spec,H_O}$ (see Definition 8.1.7); it is the same as that obtained for the horizontal verification, i.e., it is the exhaustive test set $\text{EXHAUST}_{CSpec,H_O}$ built from the contract (see Definition 8.2.1). Then the test selection process is applied, it iteratively increases the set of hypotheses, decreases the test set, and ensures that satisfying the smallest test set is equivalent to satisfying the initial test set.

Since the CO-OPN/2 refine relation is essentially a renaming, we assume that the refine relation λ is the identity on contractual specifications, and thus formula refinement Λ is the identity on HML formulae.

Applying Proposition 8.1.1 to CO-OPN/2 contractual specifications provides the following proposition.

Proposition 8.3.1 *Iterative refinement of the Test Context.*
Let $CSpec = \langle Spec, \Phi \rangle$*, and* $CSpec' = \langle Spec', \Phi' \rangle$ *be two CO-OPN/2 contractual specifications. Let* $\text{EXHAUST}_{CSpec,H_O}$ *be the exhaustive test set of* $CSpec$*, and* $T_{CSpec,H}$ *be the*

test set generated from $\text{EXHAUST}_{CSpec,H_O}$. *The following holds:*

$$(Spec' \text{ satisfies } H) \Rightarrow (Spec' \models_O \text{EXHAUST}_{CSpec,H_O} \Leftrightarrow Spec' \models_O T_{CSpec,H}).$$

Since the exhaustive test set of contractual specifications is trivially built from their contracts, the following corollary, following from Proposition 8.3.1, enables to show that satisfying the test set is equivalent to satisfying the whole contract.

Corollary 8.3.1 *Satisfying Test is Equivalent to Satisfying Contract.*
Let $CSpec = \langle Spec, \Phi \rangle$, *and* $CSpec' = \langle Spec', \Phi' \rangle$ *be two CO-OPN/2 contractual specifications. Let* $\text{EXHAUST}_{CSpec,H_O}$ *be the exhaustive test set of* $CSpec$, *and* $T_{CSpec,H}$ *be the test set generated from* $\text{EXHAUST}_{CSpec,H_O}$. *The following holds:*

$$(Spec' \text{ satisfies } H) \Rightarrow (Spec' \models_O T_{CSpec,H} \Leftrightarrow \text{MOD}_{Spec'} \models \Phi).$$

Proof.
Proposition 8.3.1 provides (1) below. $\text{EXHAUST}_{CSpec,H_O}$ is built from Φ by creating from every formula f of the contract a test case $\langle f, true \rangle$.
By definition of $\models_O$: $(Spec' \models_O \langle f, true \rangle) \Leftrightarrow \text{MOD}_{Spec'} \models f$. This provides (2) below:

$$\begin{aligned}(Spec' \text{ satisfies } H) &\overset{1}{\Rightarrow} (Spec' \models_O T_{CSpec,H} \Leftrightarrow Spec' \models_O \text{EXHAUST}_{CSpec,H_O}) \\ &\overset{2}{\Rightarrow} (Spec' \models_O T_{CSpec,H} \Leftrightarrow \text{MOD}_{Spec'} \models \Phi).\end{aligned}$$

■

Corollary 8.3.1 is not sufficient to prove that $CSpec'$ refines $CSpec$. The fact that $CSpec'$ satisfies the contract of $CSpec$ is not sufficient to guarantee that a further contractual specification $CSpec''$, satisfying the contract of $CSpec'$, satisfies as well the contract of $CSpec$. Additional conditions are necessary. Indeed, the theory of refinement based on contracts requires that the contract of $CSpec$ is part of the contract of $CSpec'$ in order to guarantee the preservation of the contract till the implementation. The corresponding requirement, when verifying the refinement using tests, consists of imposing that the test set generated from $\text{EXHAUST}_{CSpec,H_O}$ is part of the exhaustive test set of $\text{EXHAUST}_{CSpec',H'_O}$.

Proposition 8.3.2 *Preservation of Contract.*
Let $CSpec = \langle Spec, \Phi \rangle$, $CSpec' = \langle Spec', \Phi' \rangle$, *and* $CSpec'' = \langle Spec'', \Phi'' \rangle$ *be CO-OPN/2 contractual specifications. Let* $\text{EXHAUST}_{CSpec,H_O}$ *and* $\text{EXHAUST}_{CSpec',H'_O}$ *be the exhaustive test sets of* $CSpec$ *and* $CSpec'$ *respectively. Let* $T_{CSpec,H}$ *and* $T_{CSpec',H'}$ *be the test set generated from* $\text{EXHAUST}_{CSpec,H_O}$ *and* $\text{EXHAUST}_{CSpec',H'_O}$ *respectively, then the following holds:*

$$((Spec' \text{ satisfies } H) \wedge (T_{CSpec,H} \subseteq \text{EXHAUST}_{CSpec',H'_O}) \wedge (H \subseteq H')) \Rightarrow$$
$$((Spec'' \text{ satisfies } H') \Rightarrow (Spec'' \models_O T_{CSpec',H'} \Rightarrow \text{MOD}_{Spec''} \models \Phi)).$$

Proof.
Proposition 8.3.1 provides (1) below. Since $T_{CSpec,H} \subseteq \textsc{Exhaust}_{CSpec',H'_O}$ (2) holds. Finally, $H \subseteq H'$ and Corollary 8.3.1 allow us to conclude (3).

$$\begin{aligned}(Spec'' \text{ satisfies } H') &\overset{1}{\Rightarrow} (Spec'' \models_O T_{CSpec',H'} \Leftrightarrow Spec'' \models_O \textsc{Exhaust}_{CSpec',H'_O})\\ &\overset{2}{\Rightarrow} (Spec'' \models_O \textsc{Exhaust}_{CSpec',H'_O} \Rightarrow Spec'' \models_O T_{CSpec,H})\\ &\overset{3}{\Rightarrow} (Spec'' \models_O T_{CSpec,H} \Rightarrow \textsc{Mod}_{Spec''} \models \Phi).\end{aligned}$$

■

Proposition 8.3.2 above holds also if $T_{CSpec,H} \subseteq T_{CSpec',H'}$ (instead of $T_{CSpec,H} \subseteq \textsc{Exhaust}_{CSpec',H'_O}$). However, this is practically impossible to obtain, since generated test sets are made of only some relevant ground formulae, and it may happen that the test selection process choses formulae for generating $T_{CSpec,H}$ that are different from that chosen for generating $T_{CSpec',H'}$.

It is important to note that the theory of refinement based on contracts requires that $\Phi \subseteq \Phi'$ (provided that the refine relation is the identity). In terms of test sets, this means that $\textsc{Exhaust}_{CSpec,H_O} \subseteq \textsc{Exhaust}_{CSpec',H'_O}$. Proposition 8.3.2 does not guarantee this inclusion. However, it guarantees that an abstract contract is preserved during a whole refinement process, and this is sufficient to guarantee that refinements steps are correct.

For this reason, when verifying refinement using tests in practice, we alleviate the constraints of inclusion of the contracts, and we consider that the refinement is correct if contracts are preserved during the whole refinement process.

Theorem 8.3.2 *Vertical Verification.*
Let $CSpec = \langle Spec, \Phi\rangle$, and $CSpec' = \langle Spec', \Phi'\rangle$ be two CO-OPN/2 contractual specifications. Let $\textsc{Exhaust}_{CSpec,H_O}$ and $\textsc{Exhaust}_{CSpec',H'_O}$ be the exhaustive test set of $CSpec$ and $CSpec'$ respectively. Let $T_{CSpec,H}$ and $T_{CSpec',H'}$ be the test set generated from $\textsc{Exhaust}_{CSpec,H_O}$ and $\textsc{Exhaust}_{CSpec',H'_O}$ respectively. The following holds

$$(Spec' \textit{ satisfies } H) \wedge (T_{CSpec,H} \subseteq \textsc{Exhaust}_{CSpec',H'_O}) \wedge (H \subseteq H') \Rightarrow CSpec \sqsubseteq CSpec'.$$

Remark 8.3.1 *In the case of small contracts made of ground formulae, it is not necessary to use test generation, since the contract is probably equal to the generated test sest.*

Practical Verification

As described above, the **Co-opnTest** tool of Péraire [52] is used for generating test cases either from $\textsc{Exhaust}_{Spec,H_O}$ or from $\textsc{Exhaust}_{CSpec,H_O}$.

In order to verify that $T_{CSpec,H} \subseteq \textsc{Exhaust}_{CSpec',H'_O}$ in practice, or more generally that $\Phi \subseteq \Phi'$ we propose to use as well the **Co-opnTest** tool.

The use of **Co-opnTest** for verifying this inclusion is slightly different from the use of **Co-opnTest** for generating test cases. Indeed, we can roughly separate the tool into two parts: a syntactical part, and a semantical part. The semantical part takes into account CO-OPN/2 specifications with Class modules, i.e., with a dynamic behaviour. The syntactical part takes into account ADT modules. Since $T_{CSpec,H}$ and $\text{EXHAUST}_{CSpec',H'_O}$ (or Φ and Φ') are sets of ground HML formulae, we propose to syntactically verify the inclusion of the former into the latter.

Péraire [52] defines ADT modules specifying HML formulae, since the **Co-opnTest** tool actually transforms HML formulae into ADT terms in order to automatically derive Horn clauses for a Prolog resolution procedure. The idea for verifying $T_{CSpec,H} \subseteq \text{EXHAUST}_{CSpec',H'_O}$ consists of specifying this inclusion by the means of an ADT module (based on that of Péraire for HML formulae), and of defining a CO-OPN/2 specification for this module. It suffices then to generate test cases from the exhaustive test set of that CO-OPN/2 specification. If we find a test case that is not satisfied by the specification, then the refinement step is not correct. Otherwise, we can be confident that the refinement step is correct.

8.3.2 Total Contracts

Total contracts are such that $\Phi = \Phi_{Spec}$, where Φ_{Spec} denotes the whole set of ground HML formulae satisfied by the step semantics of a CO-OPN/2 specification $Spec$. In term of test cases, this means that $\text{EXHAUST}_{CSpec,H_O} = \text{EXHAUST}_{Spec,H_O}$, and the reduced test sets are such that $T_{CSpec,H} = T_{Spec,H}$.

A result similar to Theorem 8.3.2 is obtained. It is more powerful and more simply derived. Indeed, it suffices to prove that a lower-level contractual specification satisfies the test set generated from the exhaustive test set of a higher-level contractual specification, in order to ensure that the total high-level contract is included in the lower-level contract, and consequently to ensure that the refinement step is correct.

Theorem 8.3.3 *Vertical Verification.*
Let $CSpec = (Spec, \Phi_{Spec})$, and $CSpec' = (Spec', \Phi_{Spec'})$ be two CO-OPN/2 contractual specifications. Let $T_{Spec,H}$ be the test set generated from the exhaustive test set of Spec. The following holds:

$$(Spec' \text{ satisfies } H) \Rightarrow (Spec' \models_O T_{Spec,H} \Leftrightarrow CSpec \sqsubseteq CSpec').$$

Proof.
Corollary 8.3.1 is generic and applies also to total contracts. Since $T_{CSpec,H} = T_{Spec,H}$, we conclude (1) below. Since the contract of $CSpec'$ is $\Phi_{Spec'}$ we have necessarily that

$\Phi_{Spec} \subseteq \Phi_{Spec'}$, and by definition of $\sqsubseteq$ we obtain (2).

$$\begin{aligned}(Spec' \text{ satisfies } H) &\overset{1}{\Rightarrow} (Spec' \models_O T_{Spec,H} \Leftrightarrow \text{MOD}_{Spec'} \models \Phi_{Spec}) \\ &\overset{2}{\Rightarrow} (Spec' \models_O T_{Spec,H} \Leftrightarrow CSpec \sqsubseteq CSpec').\end{aligned}$$

■

8.4 Program Verification

Program verification is used to demonstrate that a given contractual program is actually a correct implementation of a given contractual CO-OPN/2 specification.

Section 6.2 shows that contractual programs are defined as contractual CO-OPN/2 specifications for their observable part. Thus, verifying that a contractual program correctly implements a contractual CO-OPN/2 specification is similar to verifying the correctness of a refinement step. Thus, similarly to refinement, in order to practically determine if $\langle Spec, \Phi\rangle \rightsquigarrow \langle Prog, \Psi\rangle$, i.e., if $\Phi \subseteq \Psi$ we make use of test generation. Without loss of generality, we make the same assumption as that made in the theory of test generation, i.e., we assume that the transition system of the program and that of the specification have the same set of events. Therefore, we assume that the formula implementation is the identity.

Since the program is the *last* step after the refinement process, it is necessary to verify that the program satisfies the contract of the contractual specification. However, it is not necessary to verify that the contract of the contractual specification is preserved by a further step, since there is no further step. Thus, it is not necessary to force the contract of the program to contain the contract of the specification. Therefore, the case of partial contracts and that of total contracts lead to the same result: in order to verify $\langle Spec, \Phi\rangle \rightsquigarrow \langle Prog, \Psi\rangle$, it is sufficient to verify that the model of the program satisfies the test set $T_{CSpec,H}$.

Indeed, we apply Proposition 8.1.1, and we obtain the following result:

Proposition 8.4.1 *Iterative refinement of the Test Context.*
Let $CSpec = \langle Spec, \Phi\rangle$ *be a CO-OPN/2 contractual specification, and* $CProg = \langle Prog, \Psi\rangle$ *be a contractual program. Let* $\text{EXHAUST}_{CSpec,H_O}$ *be the exhaustive test set of* $CSpec$*, and* $T_{CSpec,H}$ *be the test set generated from* $\text{EXHAUST}_{CSpec,H_O}$*. The following holds:*

$$(Prog \text{ satisfies } H) \Rightarrow (Prog \models_O \text{EXHAUST}_{CSpec,H_O} \Leftrightarrow Prog \models_O T_{CSpec,H}).$$

Similarly to vertical verification, we obtain that satisfying the test set is equivalent to satisfying the contract.

Corollary 8.4.1 *Satisfying Test is Equivalent to Satisfying Contract.*
Let $CSpec = \langle Spec, \Phi \rangle$ *be a CO-OPN/2 contractual specification, and* $CProg = \langle Prog, \Psi \rangle$ *be a contractual program. Let* $\text{EXHAUST}_{CSpec,H_O}$ *be the exhaustive test set of* $CSpec$*, and* $T_{CSpec,H}$ *be the test set generated from* $\text{EXHAUST}_{CSpec,H_O}$*. The following holds:*

$$(Prog \text{ satisfies } H) \Rightarrow (Prog \models_O T_{CSpec,H} \Leftrightarrow \text{MOD}_{Prog} \models \Phi).$$

Finally, since implementation relation consists of preserving the contract Φ, Corollary 8.4.1 immediately provides the fact that satisfying a test set is equivalent to be a correct implementation.

Theorem 8.4.2 *Program Verification.*
Let $CSpec = \langle Spec, \Phi \rangle$ *be a CO-OPN/2 contractual specification, and* $CProg = \langle Prog, \Psi \rangle$ *be a contractual program. Let* $T_{CSpec,H}$ *be the test set generated from* $\text{EXHAUST}_{CSpec,H_O}$*. The following holds:*

$$(Prog \text{ satisfies } H) \Rightarrow (Prog \models_O T_{CSpec,H} \Leftrightarrow CSpec \leadsto CProg).$$

Remark 8.4.1 *In the case of a total contract we have actually an inclusion of the contracts. Indeed, in this case we have* $\Phi = \Phi_{Spec}$*, and* $\Psi = \Psi_{Prog}$*, where* Ψ_{Prog} *is the set of all HML formulae satisfied by the model of Prog. As already explained* $T_{CSpec,H} = T_{Spec,H}$*. Since it is generic, Corollary 8.4.1 applies and the main result is:*

$$(Prog \text{ satisfies } H) \Rightarrow (Prog \models_O T_{Spec,H} \Leftrightarrow \text{MOD}_{Prog} \models \Phi).$$

Since $\Psi = \Psi_{Prog}$ *we have necessarily that* $\Phi \subseteq \Psi$*.*

Summary

Figure 8.1 shows the horizontal, and vertical verifications, as well as the program verification that have to be undertaken during a refinement process. The refinement process considered in Figure 8.1 starts with the pair $CSpec_0 =< Spec_0, \Phi_0 >$ as the most abstract contractual CO-OPN/2 specification. A first refinement leads to the pair $CSpec_1 =< Spec_1, \Phi_1 >$; the refinement process continues and reaches the pair $CSpec_n =< Spec_n, \Phi_n >$. Finally, the implementation phase provides the contractual program $CProg =< Prog, \Psi >$.

Horizontal verification asserts that every pair $CSpec_i =< Spec_i, \Phi_i > (0 \leq i \leq n)$ obtained during the refinement process is actually a contractual CO-OPN/2 specification, i.e., $Mod_{Spec_i} \models \Phi_i$. It consists of generating a test set T_{CSpec_i,H_i} from the exhaustive test set of $CSpec_i$ (for every $CSpec_i$ $(0 \leq i \leq n)$), and of verifying with an oracle that $Spec_i$ satisfies T_{CSpec_i,H_i}, i.e.,

$$Spec_i \models_O T_{CSpec_i,H_i} \ (0 \leq i \leq n).$$

In the case of total contracts $T_{CSpec_i,H_i} = T_{Spec_i,H_i}$ $(0 \leq i \leq n)$, where T_{Spec_i,H_i} is the test set generated from the exhaustive test set of $Spec_i$. In that case $Spec_i \models_O T_{CSpec_i,H_i}$ is a trivial result.

Vertical verification aims at verifying the correctness of the refinement steps, i.e., $\Phi_i \subseteq \Phi_{i+1}$ $(0 \leq i \leq n-1)$. It consists of verifying with an oracle that $Spec_{i+1}$ satisfies the test set generated from the exhaustive test set $Spec_i$, i.e.,

$$Spec_{i+1} \models_O T_{CSpec_i,H_i} \ (0 \leq i \leq n-1).$$

In the case of partial contracts it is necessary to verify as well that $T_{CSpec_i,H_i} \subseteq \text{EXHAUST}_{CSpec_{i+1},H_{i+1}}{}_O$.

Finally program verification enables to conclude that contractual program $CProg =< Prog, \Psi >$ correctly implements contractual CO-OPN/2 specification $CSpec_n =< Spec_n, \Phi_n >$, and hence every contractual CO-OPN/2 specification $CSpec_i$ $(0 \leq i \leq n)$. It consists of verifying with the oracle that $Prog$ satisfies T_{CSpec_n,H_n} ,i.e.,

$$Prog \models_O T_{CSpec_n,H_n}.$$

Figure 8.1 can be compared to Figure 3.1, which depicts the formal proofs the undertake during a refinement process. It is worth noting that every proof is replaced by the verification of test cases.

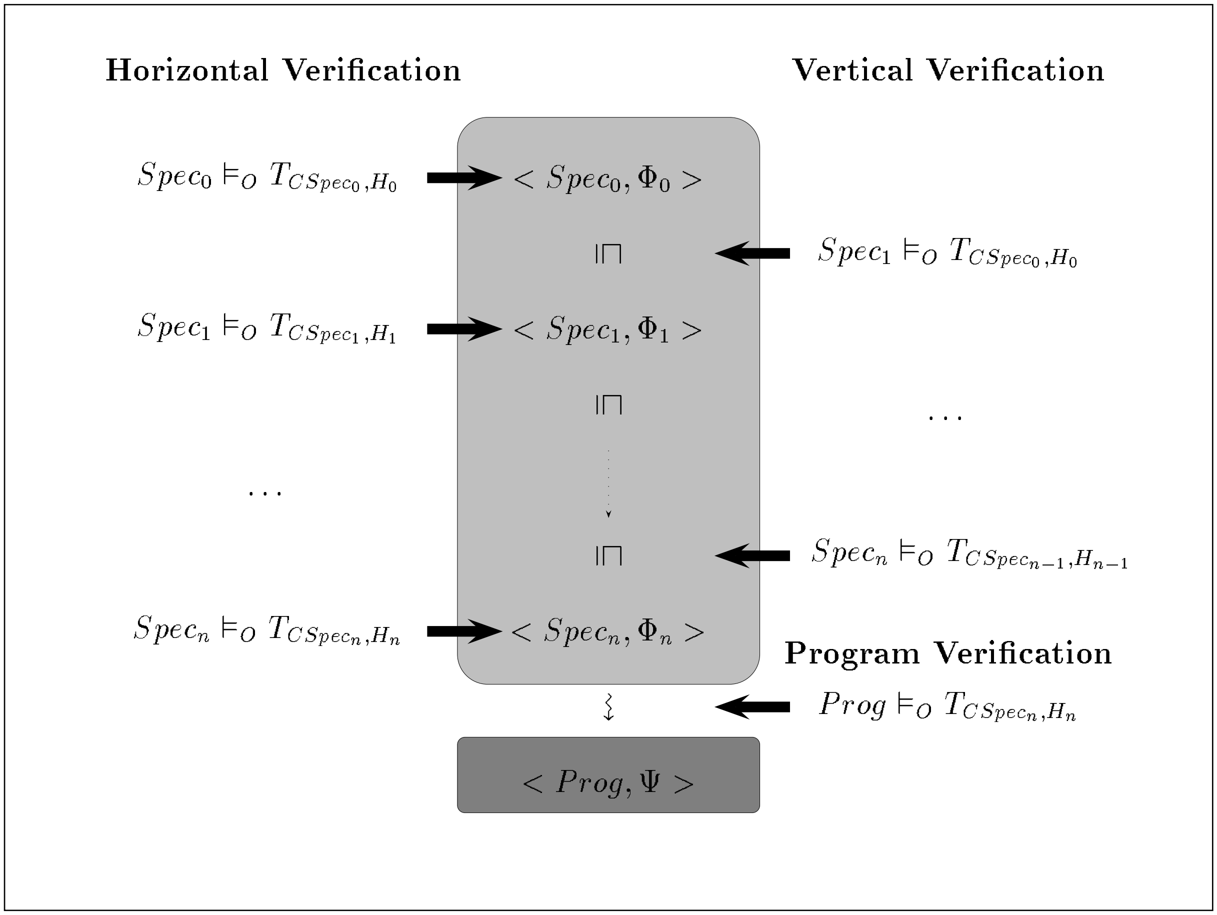


Figure 8.1: Horizontal, Vertical, and Program Verifications

Chapter 9

A Complete Example - From Requirements to Java Implementation

Chapter 3 defines a theory of refinement of formal specifications based on the use of contracts. According to these principles, Chapters 5 and 6 define a theory of refinement and implementation of CO-OPN/2 specifications. The purpose of the current chapter is to apply this theory to a concrete example.

A whole stepwise refinement process is conducted: starting from requirements informally stated, an initial contractual CO-OPN/2 specification is realized, and three refinement steps are conducted. For each step, the refine relation is given, and the proof that the refinement is correct is sketched. Once a detailed contractual CO-OPN/2 specification close to a Java program has been reached, according to Chapter 7, the implementation phase is performed, and its correctness is showed.

9.1 Informal Requirements

The Gamma paradigm [10] advocates a way of programming that is close to the chemical reactions. One or more chemical reactions are applied to a multiset: a chemical reaction removes some values from a multiset, computes some results and inserts them into the multiset. We consider the following example: computing the sum of the integers present in a multiset. Figure 9.1 depicts a multiset and a possible Gamma computation achieving the result 8.

We intend to develop an application allowing several users to insert integers into a multiset that is distributed across the Web. According to the Gamma paradigm, chemical reactions are applied on the multiset; they have to perform the sum of all the integers entered by all the users. We call DSGamma (Distributed Gamma) system, the system made of the

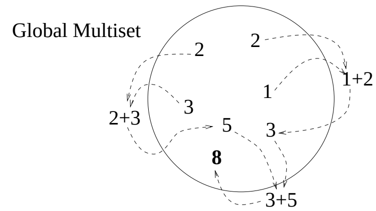


Figure 9.1: Gamma addition

users, the multiset and the chemical reactions. We present the informal requirements in three parts. The first one presents the system operations that must be provided to the users, the second one, the details about the data and internal computations, and the third one, informations about the desired implementation.

System operations: [**1**] A new user can be added to the system at any moment; [**2**] A user may add new integers into the system, at any moment, between his entering time and his exit time; [**3**] At any moment, the application eventually gives the result to a user, i.e., the sum of the integers entered in the system since the beginning; [**4**] A user may exit the system provided he has entered it.

State and internal behaviour: [**5**] The integers entered by the users are stored in a multiset; [**6**] The application realizes the sum of all the integers entered by all the users; [**7**] The sum is performed by chemical reactions according to the Gamma paradigm; [**8**] A chemical reaction removes two integers from the multiset, adds them up, and inserts the sum into the multiset; [**9**] There is only one type of chemical reaction, but several of them can occur simultaneously and concurrently on the multiset; [**10**] A chemical reaction may occur as soon as the state of the multiset is such that the chemical reaction can occur, i.e., as soon as there are at least two integers in the multiset.

Implementation: The system is implemented by the means of the Java programming language, and with an architecture using Java Applets.

9.2 Initial Specification: Centralised View

The initial CO-OPN/2 specification **I** provides the most abstract view of the DSGamma system that fulfils the informal requirements. There is a global multiset with several chemical reactions occurring concurrently on it. We have a non distributed data (the multiset), several processes (the chemical reactions), and each process, considered separately, is not distributed.

CO-OPN/2 Specifications

The initial CO-OPN/2 specification **I** is given by the least complete CO-OPN/2 specification that enables to define Class modules `Users`, defining type `user`, and `DSGammaSystem`, defining type `dsgamma-system` and static object `DSG`. These Class modules are depicted by figures 9.2, and 9.3 respectively.

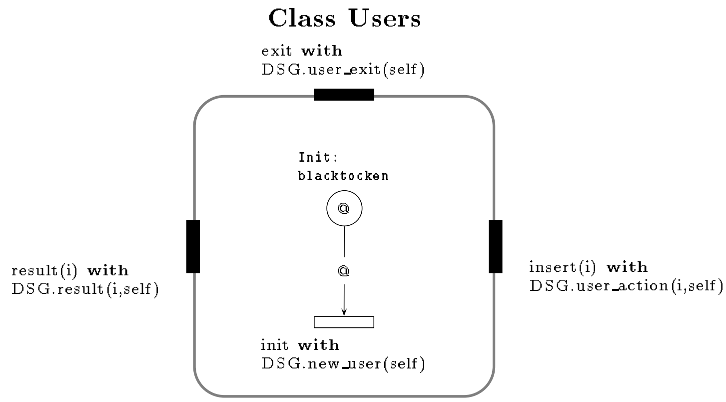


Figure 9.2: CO-OPN/2 Specification **I**: Users

Class module `Users` defines three methods: `insert(i)`, `result(i)`, and `exit`. These methods simply forward the request of the user to the underlying DSGamma system, `DSG`. As soon as a new user is created, the new user announces itself to the system, in an unobserved manner, by the means of transition `init` (firable only once).

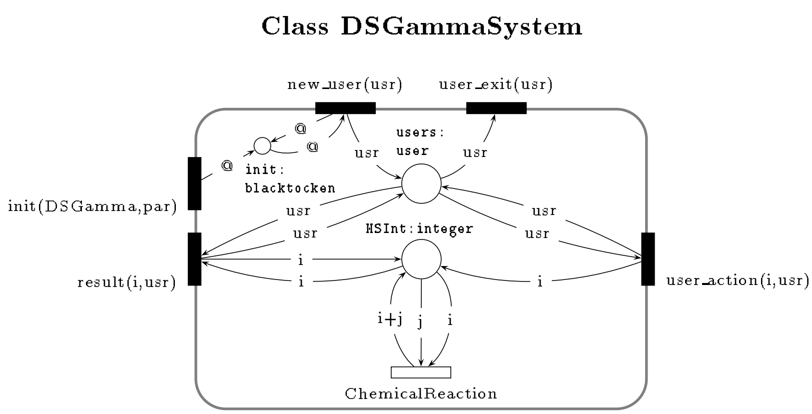


Figure 9.3: CO-OPN/2 Specification **I**: Centralized System

Class module `DSGammaSystem` defines five methods. Method `init(DSGamma,par)` is used to actually start the system. `DSGamma` is of type `string`, and `par` is of type `arraystring`, defined respectively in Class modules `Strings` and `ArrayStrings`. Method `init(DSGamma,par)` is used to start the system with parameters `par`; it simply enables the firing of method `new_user(usr)`. As explained in Section 7.3 this method will be mapped at the implementation phase to the `java` command.

Methods `new_user(usr)`, `user_action(i,usr)`, `result(i,usr)`, and `user_exit(usr)` realize actually the four services, system operations **[1]** to **[4]**, that the system provides to the users.

The `new_user(usr)` method inserts the users' identity into the `users` place of type `user` (defined by Class module `Users`). CO-OPN/2 `MSInt` place is of type `integer` (type `integer` is specified using ADT module `Integers` specifying signed integer numbers). This place models the multiset of integers entered by the users in the system. The CO-OPN/2 semantics of places is such that the content of place `MSInt` is actually given by a multiset. The `user_action(i,usr)` method checks if `usr` has already entered the system (i.e., if `usr` is in the place `users`), and inserts integer `i`, into the place `MSInt`. If the user `usr` has not yet entered the system, the method cannot be fired, thus the `i` value is not inserted into the multiset[1]. The `result(i,usr)` method checks if `usr` has already entered the system, and reads one integer `i` in the place `MSInt`. If `usr` is in the `users` place, the `user_exit(usr)` method removes `usr`.

The CO-OPN/2 `ChemicalReaction` transition models the chemical reaction. It takes two integers `i,j` from the `MSInt` place, and inserts their sum `i+j` in `MSInt`. Due to the CO-OPN/2 semantics (stabilisation process), transition `ChemicalReaction` is fired as long as it is firable, i.e., as long as there are at least two integers in `MSInt`. Meanwhile, no method can be fired. Therefore, method `result(i,usr)` is firable after `ChemicalReaction` has fired, and thus always returns the sum of all integers entered in the system since the system has been started.

CO-OPN/2 specification **I** is given by:

$$\begin{aligned}\mathbf{I} = \{&(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Integers}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{BlackTockens}},\\ &(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Strings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ArrayStrings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Users}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{DSGammaSystem}}\}.\end{aligned}$$

Indeed, in order to specify Class modules Users and DSGammaSystem, it is necessary to use as well Class module Strings, ArrayStrings and ADT modules BlackTockens, Integers, which needs the Naturals and the Booleans ADT modules.

[1]remember that if one element needed by a method or transition event is not available, then its execution is impossible.

Contract

The contract of CO-OPN/2 specification **I** is given by $\Phi_{\mathbf{I}} = \{\phi_{\mathbf{I}_1}, \ldots, \phi_{\mathbf{I}_6}\}$ below, for the set of variables $X_{\mathbf{I}} = \{usr_1, usr_2\}_{\text{user}} \cup \{i, j\}_{\text{integer}}$:

$$\phi_{\mathbf{I}_1} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])><usr_1.\,\text{create}><usr_2.\,\text{create}>\ \mathbf{T}$$

$$\phi_{\mathbf{I}_2} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])><usr_1.\,\text{create}><usr_1.\,\text{insert}(i)>\ \mathbf{T}$$

$$\phi_{\mathbf{I}_3} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])><usr_1.\,\text{create}><usr_1.\,\text{insert}(i)><usr_1.\,\text{result}(i)>\ \mathbf{T}$$

$$\phi_{\mathbf{I}_4} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])><usr_1.\,\text{create}><usr_2.\,\text{create}>$$
$$<usr_1.\,\text{insert}(i)\ //\ usr_2.\,\text{insert}(j)><usr_1.\,\text{result}(i+j)>\ \mathbf{T}$$

$$\phi_{\mathbf{I}_5} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])><usr_1.\,\text{create}><usr_2.\,\text{create}>$$
$$<usr_1.\,\text{insert}(i)><usr_2.\,\text{insert}(j)><usr_1.\,\text{result}(i+j)>\ \mathbf{T}$$

$$\phi_{\mathbf{I}_6} = <\text{DSG}\,.\,\text{init}(\text{DSGamma}, [])>\ ((<usr_1.\,\text{create}><usr_1.\,\text{exit}>) \wedge$$
$$\neg(<usr_1.\,\text{exit}><usr_1.\,\text{create}>))\mathbf{T}.$$

System operations **[1]** to **[4]** are partially covered by this contract. Indeed, system operations **[1]** to **[3]** require items that have to be true *at any moment*; system operation **[4]** requires that *any* user may exit provided he has entered the system. In order to completely cover these system operations, it is necessary to have an infinite contract covering every case, since the chosen logic does not allow to express several properties by the means of a single formula. Thus, in order to remain simple in this example, we have chosen only some of these properties.

Property $\phi_{\mathbf{I}_1}$ corresponds to system operation **[1]**; it states that DSGamma system `DSG` is started with no parameters, and that two users can be created, and hence entered in the system. Property $\phi_{\mathbf{I}_2}$ corresponds to system operation **[2]**; it states that once a user has entered the system, he can enter an integer. Properties $\phi_{\mathbf{I}_3}$ to $\phi_{\mathbf{I}_5}$ stand for system operation **[3]**; three cases have been considered: a single user enters an integers and gets the result; two users enter simultaneously an integer and one of them gets the result; two users enter sequentially an integer and one of them gets the result. Finally, property $\phi_{\mathbf{I}_6}$ stands for system operation **[4]**; it states that a user may exit after having entered the system, and a user cannot exit the system before entering it.

These formulae are actually properties of **I**, since every formula is a possible path beginning from state $\langle \bot, \varnothing, \bot \rangle$.

Definition 9.2.1 ***CI.***
We define the following contractual CO-OPN/2 specification:

$$\boldsymbol{CI} = < \boldsymbol{I}, \Phi_{\boldsymbol{I}} > .$$

Remark 9.2.2 *Requirements **[5]** to **[10]** are not expressible by the means of HML formulae. Indeed, these requirements deal with the internal behaviour of the system, and HML formulae can be built with observable events only. However, they are actually satisfied by CO-OPN/2 specification **I**.*

9.3 First Refinement: Data Distribution

The initial specification **I** provides a centralised view of the application. As we intend to obtain an implemented application distributed over the Web, it is now necessary to introduce distributivity in the specification. Refinement **R1** is concerned with data distributivity.

Refinement Process

The multiset of integers is physically distributed over several different locations. We call *local multiset* the portion MS_i of the multiset present in a given location, and we call *global multiset* the multiset obtained by the union of all the local multisets. Figure 9.4 gives an illustration of chemical reactions over the distributed multisets MS_i, that compute the result 8.

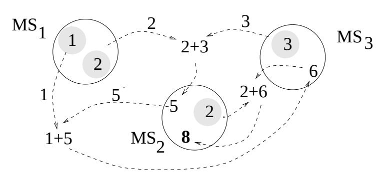


Figure 9.4: Distributed Gamma-like addition

Class module `Users` is the same as in specification **I**. Class module `DSGammaSystem` provides the same methods as the initial specification **I**. However, as the global multiset is split over several local multisets (one for each user), we redefine the behaviour of methods of Class module `DSGammaSystem` such that: (1) each user is mapped to a local multiset specified with a bag of integers; (2) the chemical reactions have to remove integers from one or more local multisets; (3) the integers present in the local multiset of a user who wants to leave the system must be properly dispatched to the other local multisets.

CO-OPN/2 Specifications

CO-OPN/2 specifications of the application with distributed multisets is given by Class module `Users` depicted by figure 9.2, and Class module `DSGammaSystem1` depicted by figure 9.5, which defines type `dsgamma-system1`, and static object `DSG`.

The `MSInt` place stores the local multiset of users currently in the system, while the `MSIntToEmpty` place stores the local multiset of users wishing to leave the system. They are Cartesian products of `users` and `bagintegers` of type `pairuserbag`, defined in ADT module `PairUserBags`; pairs are generated using operator `<>`. The specification of the type `baginteger` is made using ADT module `BagIntegers` which defines an empty bag { } and an operation `'` for adding new integers to the bag.

Class DSGammaSystem1

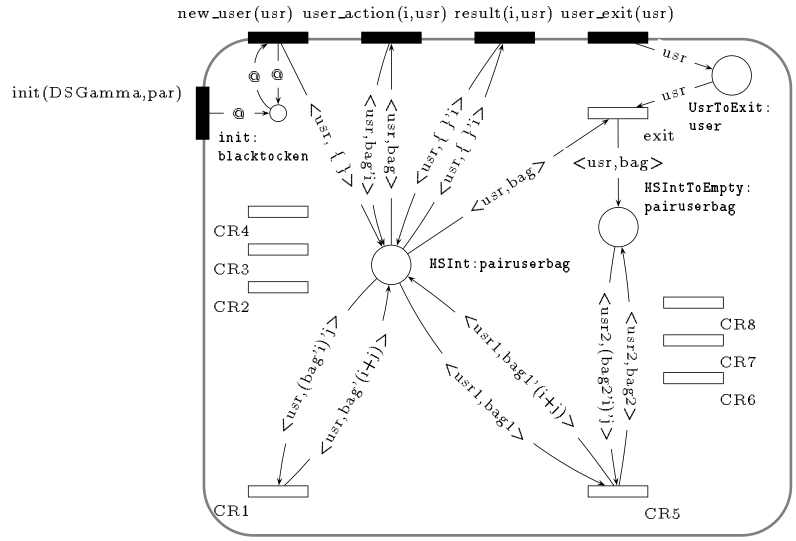


Figure 9.5: Refinement **R1**: Data Distribution

The `init(DSGamma,par)` method starts the system. The `new_user(usr)` method inserts pairs of integers and empty bags <`usr`,{} > into the `MSInt` place. A new user joins the system with an empty bag, representing an empty local multiset. The `user_action(i,usr)` method checks if `usr` has already entered the system, i.e., removes the pair `<usr,bag>` from the place `MSInt`, and inserts the `i` value into `bag`, i.e., inserts the pair `<usr,bag ' i>` into `MSInt`. Bag `bag ' i` stands for a new bag made of the union of `bag` and the set {`i`}. This method cannot be fired if `usr` has not already joined the system. The `result(i,usr)` method can be fired iff the bag of user `usr` contains exactly one element `i` (i.e., { } `' i`). It is worth noting that due to the CO-OPN/2 semantics, after each firing of the chemical reactions, *only one* integer remains in *one* local bag.

The `user_exit(usr)` method inserts the `usr` value in the place `UsrToExit`. The `exit` transition then removes the pair `<usr,bag>` from the `MSInt` place and inserts it into the `MSIntToEmpty` place. As the user is tightly coupled with a local multiset, it is necessary to introduce at this point a treatment for dispatching his values. Therefore, after having exited the system, a user may no longer enter a new integer, nor get the result, nor exit the system, unless it reenters the system, and the system itself cannot add integers into the user's local multiset.

Four chemical reactions (`CR1` to `CR4`) have been defined on `MSInt` only. They describe the four possible ways of removing two integers from one or two bags and inserting their sum into a (possibly other) bag. Four chemical reactions (`CR5` to `CR8`) have been defined on both `MSInt` and `MSIntToEmpty`. They are basically the same as the four chemical reactions defined on `MSInt` only, except for the fact that they have to remove integers from local multisets stored in the `MSIntToEmpty` place, and they have to insert integers into local multisets stored in the `MSInt` place. These four chemical reactions specify the fact that once a user has decided to leave the system, then his local multiset has to be emptied, no new integers may be inserted into his local multiset. For simplicity purpose, figure 9.5 depicts only the behaviour of chemical reactions `CR1` and `CR5`: for `CR1` two integers `i,j` are removed from the same local multiset, their sum is inserted into this local multiset; for `CR5` two integers `i,j` are removed from the same local multiset in `MSIntToEMpty`, and their sum is added to another local multiset in `MSInt`.

After a firing of the `CRi` transitions, only one integer remains in `MSInt`. The remaining integer is the sum of the integers present in all the bags of `MSInt` and `MSIntToEmpty` before the firing of `CRi`. If all users leave the system, the computation is halted until a new user enters the system.

CO-OPN/2 specification **R1** is given by:

$$\mathbf{R1} = \{(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Integers}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{BlackTockens}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{BagIntegers}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{PairUserBags}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Strings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ArrayStrings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Users}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{DSGammaSystem1}}\}.$$

Class modules `Users` and `DSGammaSystem1` require Class module `Strings`, `ArrayStrings`, and ADT modules `BlackTockens`, `BagIntegers` and `PairUserBags` which require ADT module `Integers`, `Naturals`, and `Booleans`.

Contract

The contract of CO-OPN/2 specification **R1** is given by $\Phi_{\mathbf{R1}} = \{\phi_{\mathbf{R1}_1}, \ldots, \phi_{\mathbf{R1}_7}\}$ below, for the set of variables $X_{\mathbf{R1}} = \{usr_1, usr_2\}_{user} \cup \{i, j\}_{\text{integer}}$:

$$\begin{aligned}
\phi_{\mathbf{R1}_1} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_2\text{. create}\rangle\ \mathbf{T}\\
\phi_{\mathbf{R1}_2} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_1\text{. insert}(i)\rangle\ \mathbf{T}\\
\phi_{\mathbf{R1}_3} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_1\text{. insert}(i)\rangle\langle usr_1\text{. result}(i)\rangle\ \mathbf{T}\\
\phi_{\mathbf{R1}_4} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_2\text{. create}\rangle\\
&\quad \langle usr_1\text{. insert}(i)\ //\ usr_2\text{. insert}(j)\rangle\langle usr_1\text{. result}(i+j)\rangle\ \mathbf{T}\\
\phi_{\mathbf{R1}_5} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_2\text{. create}\rangle\\
&\quad \langle usr_1\text{. insert}(i)\rangle\langle usr_2\text{. insert}(j)\rangle\langle usr_1\text{. result}(i+j)\rangle\ \mathbf{T}\\
\phi_{\mathbf{R1}_6} &= \langle \text{DSG . init(DSGamma, [])}\rangle\ ((\langle usr_1\text{. create}\rangle\langle usr_1\text{. exit}\rangle) \wedge\\
&\quad \neg(\langle usr_1\text{. exit}\rangle\langle usr_1\text{. create}\rangle))\mathbf{T}\\
\phi_{\mathbf{R1}_7} &= \langle \text{DSG . init(DSGamma, [])}\rangle\langle usr_1\text{. create}\rangle\langle usr_2\text{. create}\rangle\\
&\quad \langle usr_1\text{. insert}(i)\rangle\langle usr_1\text{. exit}\rangle\langle usr_2\text{. result}(i)\rangle\ \mathbf{T}.
\end{aligned}$$

Formulae $\phi_{\mathbf{R1}_1}$ to $\phi_{\mathbf{R1}_6}$ correspond to formulae $\phi_{\mathbf{I}_1}$ to $\phi_{\mathbf{I}_6}$. They are exactly the same because observable events of **I** and of **R1** are the same. Formula $\phi_{\mathbf{R1}_7}$ is a new formula.

It states the fact that a user leaving the system does not affect the computing of the result. These formulae are actually properties of **R1**.

Definition 9.3.1 ***CR1***.
We define the following contractual CO-OPN/2 specification

$$\textbf{\textit{CR1}} = < \textbf{\textit{R1}}, \Phi_{\textbf{\textit{R1}}} > .$$

Refine Relation

Given **CI**, **CR1** given by Definitions 9.2.1 and 9.3.1 respectively, we define a CO-OPN/2 refine relation $\lambda_0 \subseteq \textsc{Elem}_{\mathbf{CI}} \times \textsc{Elem}_{\mathbf{CR1}}$ in the following way:

$$\begin{aligned}
\lambda_{0_{SA}} =& \{(\text{integer}, \text{integer})\} \\
\lambda_{0_{SC}} =& \{(\text{string}, \text{string}), (\text{arraystring}, \text{arraystring}), (\text{user}, \text{user}), \\
& (\text{dsgamma-system}, \text{dsgamma-system1})\} \\
\lambda_{0_{FA}} =& \{(+_{\text{integer}}, +_{\text{integer}})\} \\
\lambda_{0_{FC}} =& \{(\text{new}_{\text{string}}, \text{new}_{\text{string}}), (\text{init}_{\text{string}}, \text{init}_{\text{string}}), \\
& (\text{new}_{\text{arraystring}}, \text{new}_{\text{arraystring}}), (\text{init}_{\text{arraystring}}, \text{init}_{\text{arraystring}}), \\
& (\text{new}_{\text{user}}, \text{new}_{\text{user}}), (\text{init}_{\text{user}}, \text{init}_{\text{user}}), \\
& (\text{new}_{\text{dsgamma-system}}, \text{new}_{\text{dsgamma-system1}}), (\text{init}_{\text{dsgamma-system}}, \text{init}_{\text{dsgamma-system1}})\} \\
\lambda_{0_M} =& \{(\text{exit}_{\text{user}}, \text{exit}_{\text{user}}), (\text{insert}_{\text{user,integer}}, \text{insert}_{\text{user,integer}}), \\
& (\text{result}_{\text{user,integer}}, \text{result}_{\text{user,integer}}), \\
& (\text{init}_{\text{dsgamma-system,string,arraystring}}, \text{init}_{\text{dsgamma-system1,string,arraystring}}), \\
& (\text{new_user}_{\text{dsgamma-system,integer,user}}, \text{new_user}_{\text{dsgamma-system1,integer,user}}), \\
& (\text{user_action}_{\text{dsgamma-system,integer,user}}, \text{user_action}_{\text{dsgamma-system1,integer,user}}), \\
& (\text{result}_{\text{dsgamma-system,user}}, \text{result}_{\text{dsgamma-system1,user}}), \\
& (\text{user_exit}_{\text{dsgamma-system,user}}, \text{user_exit}_{\text{dsgamma-system1,user}})\} \\
\lambda_{0_O} =& \{(\text{DSG}_{\text{dsgamma-system}}, \text{DSG}_{\text{dsgamma-system1}})\} \\
\lambda_{0_X} =& \{(usr_1, usr_1), (usr_2, usr_2), (i, i), (j, j)\}.
\end{aligned}$$

CO-OPN/2 specification **R1** contains the interface of CO-OPN/2 specification **I**. For this reason, the refine relation maps elements appearing in the contract of **CI** to elements of **CR1** having the same name.

Formula Refinement

Since refine relation λ_0 is the identity on elements of **CI**, formula refinement Λ_0 is the identity as well. Thus, we have trivially that $\Lambda_0(\Phi_{\mathbf{I}}) \subseteq \Phi_{\mathbf{R1}}$.

9.4 Second Refinement: Behaviour Distribution

Refinement **R1** provides a distributed view of the application at the data level. As we intend to obtain a Java application distributed over the Web, it is necessary to think about applets storing the local multiset related to the user who starts the applet. These applets need to communicate with each other in order to realize the DSGamma system. The Java programming language constrains an applet to connect exclusively to the host where it comes from. For this reason, refinement **R2** introduces a server. This leads to a behaviour distribution.

Refinement Process

The server acts as a buffer between all applets. The server is only able to receive integers from a set of applets, and to send these integers to this same set of applets, such that an integer goes randomly from one applet to another via the server.

The system operations and internal behaviours are specified such that: (1) the server is specified as a FIFO buffer; (2) each user is mapped to an applet; (3) the applets are responsible to maintain a local multiset of integers; (4) an applet has to insert integers entered by the user into its local multiset; (5) an applet has to collect pairs of integers, to make their sum, and to insert this sum into its local multiset; (6) an applet has to send integers to the server; (7) the applet has to correctly send its local multiset of integers to the server, once the user wants to leave the system; (8) the applets have to avoid a deadlock situation that would occur when the number of integers present in the whole system is less than the number of applets.

CO-OPN/2 Specification

The CO-OPN/2 Class modules of the application viewed with a client/server architecture are given by figures 9.6, 9.7 and 9.8. Class module `DSGammaSystem2` specifies the underlying system; it defines type `dsgamma-system2`, and static object `DSG`. Class module `GlobalRelays` specifies the server and defines type `globalrelay`. Class module `Applets` specifies the applets, and defines type `applet`.

Class module `DSGammaSystem2` simply specifies the start up of the system: method `init(DSGamma,par)` creates and stores a server `gr` as an instance of Class `GlobalRelays`. Class module `DSGammaSystem2` offers method `get_server(gr)`. This method is used by the newly created applets to learn the identity of the server they have to use in order to communicate with each other.

Class module `GlobalRelays` maintains a FIFO buffer of integers. An integer `i` is inserted at the end of this FIFO by the means of the `put(i)` method, and an integer is removed, from the beginning of this FIFO when it is non-empty, using `get(next of (b'i))`. ADT

Class DSGammaSystem2

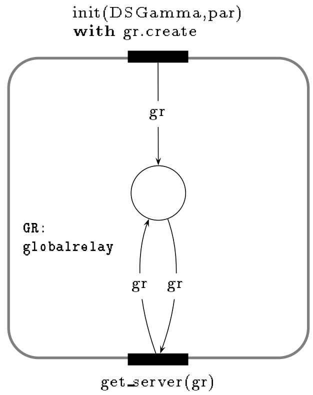


Figure 9.6: Refinement R2: Overall System

module `FifoIntegers` defines the type `fifointeger`, the empty fifo `[]`, as well as operator `'` for appending an integer at the end of the FIFO, and operators `remove from` and `next of` for removing and reading respectively the integer at the beginning of the FIFO.

Class GlobalRelays

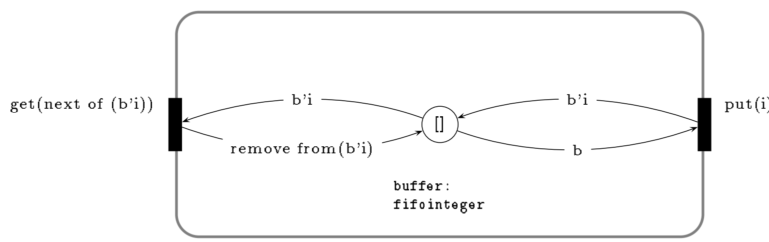


Figure 9.7: Refinement **R2**: Server Side

Class module `Applets` is meant to replace Class module `Users` of CO-OPN/2 specification **I**. Therefore, it specifies the same three CO-OPN/2 methods: `insert(i)`, `exit`, `result(i)`.

As soon as a new applet is created the `init` transition requires the server `gr` from DS-Gamma system `DSG`, in an unobservable manner (calling `DSG.get_server(gr)`). The `end` place is initialised with `false`, and the `beginning` place with `true`. The `end` place stores the value `false` if the user is currently in the system and stores the value `true` if the user exits. The `beginning` place stores the value `true` if a first integer has to be requested, and nothing if a first integer has already been obtained. This place is used to ensure that a new first integer is requested only after the previous sum has been computed. The `MSInt` place stores integers, it specifies the local multiset maintained by the applet in behalf of the user.

The `insert(i)` method inserts the integer `i` into the local multiset. The `exit` method replaces the token `false` by the token `true` in place `end`. In that way, all methods are no longer firable. The `result(i)` method returns an integer which is either a partial sum or a complete sum.

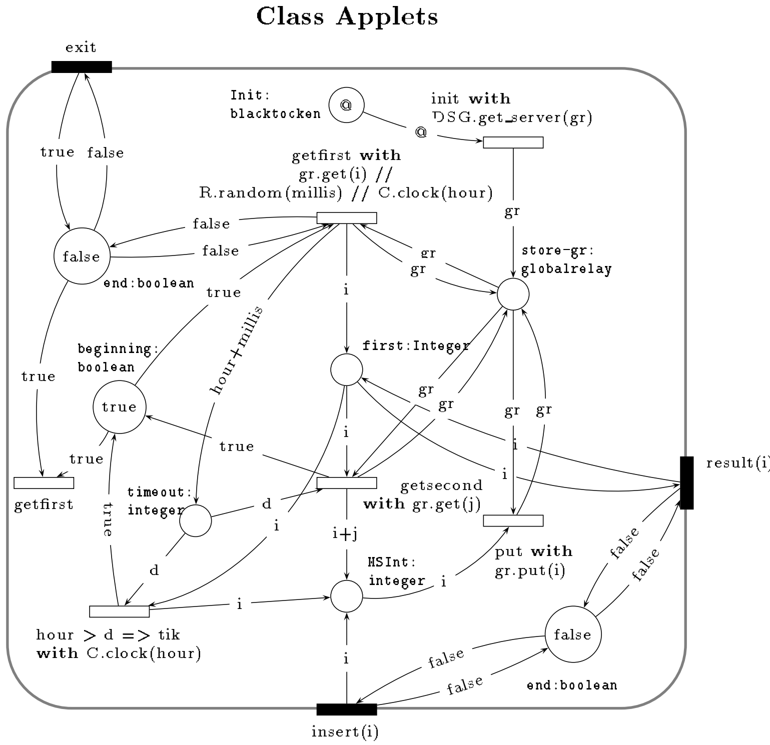


Figure 9.8: Refinement R2: Client Side

Chemical reactions are specified by the means of the four transitions: `getfirst`, `getsecond`, `tik`, `put`. The `getfirst` transition is responsible for obtaining the first integer being involved in a sum; as soon as it obtains a first integer from server `gr` it enables a timeout. The `getsecond` transition is responsible for removing a second integer from `gr`, and for disabling the timeout. The `tik` transition handles a timeout event occurring when a second integer has not been obtained by `getsecond` during the elapsed time. It is responsible for disabling the timeout and inserting the first integer (instead of a sum) into the local multiset. This timeout is necessary, because a deadlock occurs as soon as the number of integers present in the global multiset (the union of the local multisets) is smaller than or equal to the number of users, because all integers are blocked by different applets. During the deadlock, method `result(i)` is firable, it returns a partial sum. After a possibly long time, only one integer will remain in the system, because pairs of integers will succeed in meeting in the same applet. Note that due to the `tik` transitions, this integer will go from one applet to the other one. In this case, method `result(i)` returns the correct sum. The `put` transition randomly removes integers from the local multiset, and sends them to `gr`.

As soon as a user exits, the `getfirst` transition stops receiving integers. Progressively, `MSInt` place is emptied by transition `put`, and finally the applet ends its activity. If all the users leave the system simultaneously, then the applets will send all their integers, stored in MSInt, and stop receiving integers, thus gr will store all the integers. A remaining integer is obtained provided at least one user remains in the system.

CO-OPN/2 specification **R2** is given by:

$$\begin{aligned}\mathbf{R2} = \{&(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Integers}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{BlackTockens}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}},\\ &(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{FifoIntegers}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Clock}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Random}},\\ &(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Strings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ArrayStrings}},\\ &(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Applets}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{GlobalRelays}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{DSGammaSystem2}}\}.\end{aligned}$$

Contract

The contract of CO-OPN/2 specification **R2** is given by $\Phi_{\mathbf{R2}} = \{\phi_{\mathbf{R2}_1}, \ldots, \phi_{\mathbf{R2}_9}\}$ below, for the set of variables $X_{\mathbf{R2}} = \{a_1, a_2, a_3\}_{applet} \cup \{i, j, a, b\}_{\text{integer}} \cup \{gr\}_{\text{globalrelay}}$:

$$\begin{aligned}
\phi_{\mathbf{R2}_1} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_2.\text{create>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_2} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_1.\text{insert}(i)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_3} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_1.\text{insert}(i)\text{><}a_1.\text{result}(i)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_4} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_2.\text{create>}\\
&\quad \text{<}a_1.\text{insert}(i)\ //\ a_2.\text{insert}(j)\text{><}a_1.\text{result}(i+j)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_5} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_2.\text{create>}\\
&\quad \text{<}a_1.\text{insert}(i)\text{><}a_2.\text{insert}(j)\text{><}a_1.\text{result}(i+j)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_6} &= \text{<DSG.init(DSGamma, [])>}\ ((\text{<}a_1.\text{create><}a_1.\text{exit>}) \wedge\\
&\quad \neg(\text{<}a_1.\text{exit><}a_1.\text{create>}))\mathbf{T}\\
\phi_{\mathbf{R2}_7} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_2.\text{create>}\\
&\quad \text{<}a_1.\text{insert}(i)\text{><}a_1.\text{exit><}a_2.\text{result}(i)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_8} &= \text{<DSG.init(DSGamma, [])><}a_1.\text{create><}a_2.\text{create><}a_3.\text{create>}\\
&\quad \text{<}a_1.\text{insert}(i)\ //\ a_2.\text{insert}(j)\text{><}a_2.\text{result}(i)\text{><}a_1.\text{result}(j)\text{>}\\
&\quad \text{<}a_3.\text{result}(i+j)\text{>}\ \mathbf{T}\\
\phi_{\mathbf{R2}_9} &= \text{<}gr.\text{create><}gr.\text{put}(a)\text{><}gr.\text{put}(b)\text{>}\\
&\quad (\text{<}gr.\text{get}(a)\text{>} \wedge\neg \text{<}gr.\text{get}(b)\text{>})\mathbf{T}.
\end{aligned}$$

Formulae $\phi_{\mathbf{R2}_1}$ to $\phi_{\mathbf{R2}_7}$ are similar to formulae $\phi_{\mathbf{R1}_1}$ to $\phi_{\mathbf{R1}_7}$: users are simply replaced by applets. Formulae $\phi_{\mathbf{R2}_8}$ and $\phi_{\mathbf{R2}_9}$ are new formulae. Formula $\phi_{\mathbf{R2}_8}$ states that when the number of entered integers is less than the number of applets, it may occur that the system enters a deadlock state (i and j are blocked in applet a_2 and a_1 respectively) but the result is finally correctly computed (and visible for a_3)[2]. Formula $\phi_{\mathbf{R2}_9}$ states

[2]Formulae $\phi_{\mathbf{R2}_4}$ and $\phi_{\mathbf{R2}_5}$ have also less or equal integers than the number of applets, but these formulae correspond to the case where the deadlock does not occur and is not observed.

that instances of Class module `GlobalRelays` act as a FIFO. These formulae are actually properties of **R2**.

Definition 9.4.1 ***CR2.***
We define the following contractual CO-OPN/2 specification

$$\boldsymbol{CR2} =< \boldsymbol{R2}, \Phi_{\boldsymbol{R2}} > .$$

Refine Relation

Given **CR1**, **CR2** of Definitions 9.3.1 and 9.4.1 respectively, we define a CO-OPN/2 refine relation $\lambda_1 \subseteq \text{ELEM}_{\mathbf{CR1}} \times \text{ELEM}_{\mathbf{CR2}}$ in the following way:

$$\begin{aligned}
\lambda_{1_{SA}} &= \{(\text{integer}, \text{integer})\} \\
\lambda_{1_{SC}} &= \{((\text{string}, \text{string}), (\text{arraystring}, \text{arraystring}), (\text{user}, \text{applet}), \\
&\quad (\text{dsgamma-system1}, \text{dsgamma-system2})\} \\
\lambda_{1_{FA}} &= \{(+_{\text{integer}}, +_{\text{integer}})\} \\
\lambda_{1_{FC}} &= \{(\text{new}_{\text{string}}, \text{new}_{\text{string}}), (\text{init}_{\text{string}}, \text{init}_{\text{string}}), \\
&\quad (\text{new}_{\text{arraystring}}, \text{new}_{\text{arraystring}}), (\text{init}_{\text{arraystring}}, \text{init}_{\text{arraystring}}), \\
&\quad (\text{new}_{\text{user}}, \text{new}_{\text{applet}}), (\text{init}_{\text{user}}, \text{init}_{\text{applet}}), \\
&\quad (\text{new}_{\text{dsgamma-system1}}, \text{new}_{\text{dsgamma-system2}}), (\text{init}_{\text{dsgamma-system1}}, \text{init}_{\text{dsgamma-system2}})\} \\
\lambda_{1_M} &= \{(\text{exit}_{\text{user}}, \text{exit}_{\text{applet}}), (\text{insert}_{\text{user,integer}}, \text{insert}_{\text{applet,integer}}), \\
&\quad (\text{result}_{\text{user,integer}}, \text{result}_{\text{applet,integer}}), \\
&\quad (\text{init}_{\text{dsgamma-system1,string,arraystring}}, \text{init}_{\text{dsgamma-system2,string,arraystring}})\} \\
\lambda_{1_O} &= \{(\text{DSG}_{\text{dsgamma-system1}}, \text{DSG}_{\text{dsgamma-system2}})\} \\
\lambda_{1_X} &= \{(usr_1, a_1), (usr_2, a_2), (i, i), (j, j)\}.
\end{aligned}$$

Refine relation λ_1 maps `init` method and `DSG` object of Class module DSGammaSystem1 of **R1** to `init` method and `DSG` object respectively of Class module DSGammaSystem2 of **R2**. Since, the other methods are no longer in DSGammaSystem2 of **R2** and does not take part in contract $\Phi_{\mathbf{R1}}$, the refine relation is not defined for them. Since Class module Applets replaces Class module Users, elements of Class module Users are simply mapped to elements of Class module Applets with the same name.

Formula Refinement

Refine relation λ_1 is essentially a renaming of methods of Class module Users to methods of Class module Applets. Formula refinement Λ_1 is simply a renaming as well. Thus, we have actually $\Lambda_1(\Phi_{\mathbf{R1}}) \subseteq \Phi_{\mathbf{R2}}$.

9.5 Third Refinement: Communication Layer

Refinement **R2** provides a client/server view of the application, with applets communicating with each other through a server acting as a FIFO buffer. The applets communicate directly with the server. As the targeted application has to run across several physically distributed hosts, it is now time to introduce the sockets, i.e., the communication layer between the applets and the server. The specification provided at this stage is also intended to be the last one before the Java program. For this reason, refinement **R3** takes into account features of the Java programming language, according to Chapter 7. Therefore, it specifies all the Java components that will be part of the final program.

Refinement Process

The informal view of both specification **R3** and the implementation of the DSGamma system is given by figure 9.9. The server is bigger than it is in refinement **R2**, it is

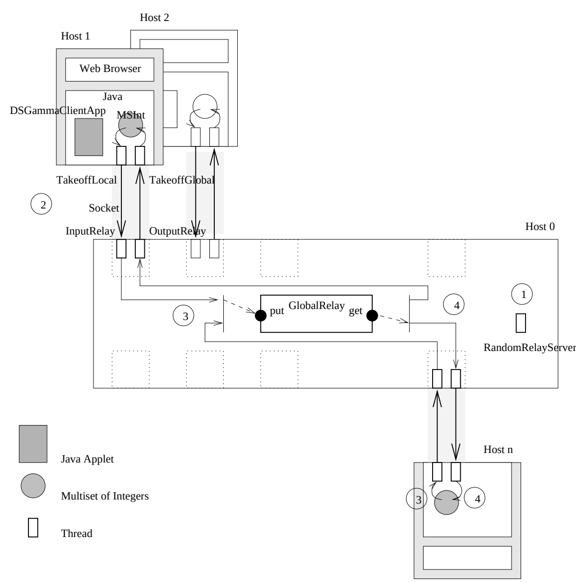


Figure 9.9: DSGamma Implemented Architecture

now given by class `RandomRelayServer` which is a sub-class of Class module `JavaThread` (position 1 on figure 9.9). It handles the following elements: an instance of Class module `JavaServerSockets` for handling connections with applets; an instance of Class module `GlobalRelay`, which handles a FIFO buffer specified with a `JavaVector`; and for each applet a pair of threads, of classes `OutputRelay`, `InputRelay`, which are dedicated to the handling of the communication with an applet (position 2 on figure 9.9).

The global multiset is logically given by the union of (1) several local multisets, each one located inside an applet; (2) the FIFO buffer maintained by the `GlobalRelay` object; and (3) the sockets buffers.

The applets are given by class DSGammaClientApp. They are more complex than what they are in refinement **R2**. As soon as an applet is created, two threads of classes TakeoffLocal, TakeoffGlobal are created. These threads are responsible for communicating with the server using the socket; and for the handling of the chemical reactions, the timeout and the quitting protocol (position 2 on figure 9.9). The applet also handles the local multiset MSInt, which is specified as an instance of Class module JavaVectors.

The communication layer is given by the sockets. Java sockets are specified by several Class modules: JavaSockets, JavaDataInputStreams, JavaDataOutputStreams, JavaInputStreams, JavaOutputStreams, and JavaServerSockets. For every applet connecting to the server, two streams are created: the first stream goes from the server to the applet, it is made of one instance of JavaDataInputStreams at the applet side and one instance of JavaDataOutputStreams at the server side. The second stream goes from the applet to the server; it is made of one instance of JavaDataInputStreams at the server side and one instance of JavaDataOutputStreams at the applet side. More simply said, every socket is specified with four buffers (two buffers per stream).

CO-OPN/2 Specifications

CO-OPN/2 specification of the application close to the Java program is given by several CO-OPN/2 classes specifying Java basics classes (among others the Java classes needed for handling sockets), several CO-OPN/2 classes specifying the server side, and several CO-OPN/2 classes specifying the client side (i.e., applet side), and a class for specifying the underlying Java Virtual Machine.

System: Class module JVM replaces Class module DSGammaSystem2 of refinement **R2**. It defines type jvm and static object JVM[3].

Class JVM

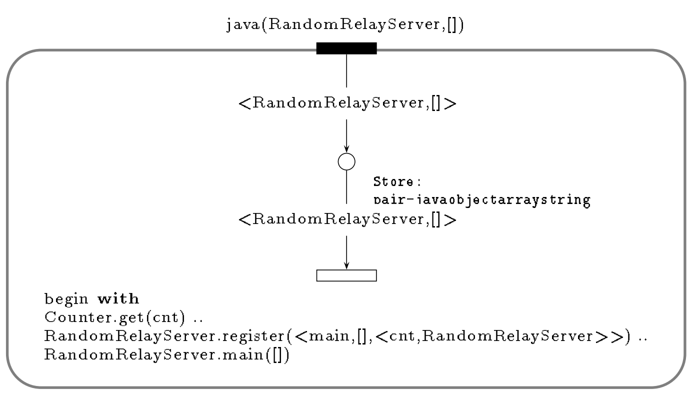


Figure 9.10: Refinement **R3**: Java Virtual Machine

[3]remember that every Class module specifying a Java class defines a static object having the same name as the name of the class. This object stands for the Java Class object of the class.

Method `java(RandomRelayServer,[])` enables the firing of the `begin` transition, which starts the `main` method of Java Class object `RandomRelayServer` with an empty string of arguments.

Server side: Class module `RandomRelayServer` defines type `randomrelayserver`. It is partially given by figure 9.11, is a sub-class of Class module `JavaThreads` (see Subsection 7.1.6). It defines a main method that creates an instance of RandomRelayServer. This thread is actually the server of all applets.

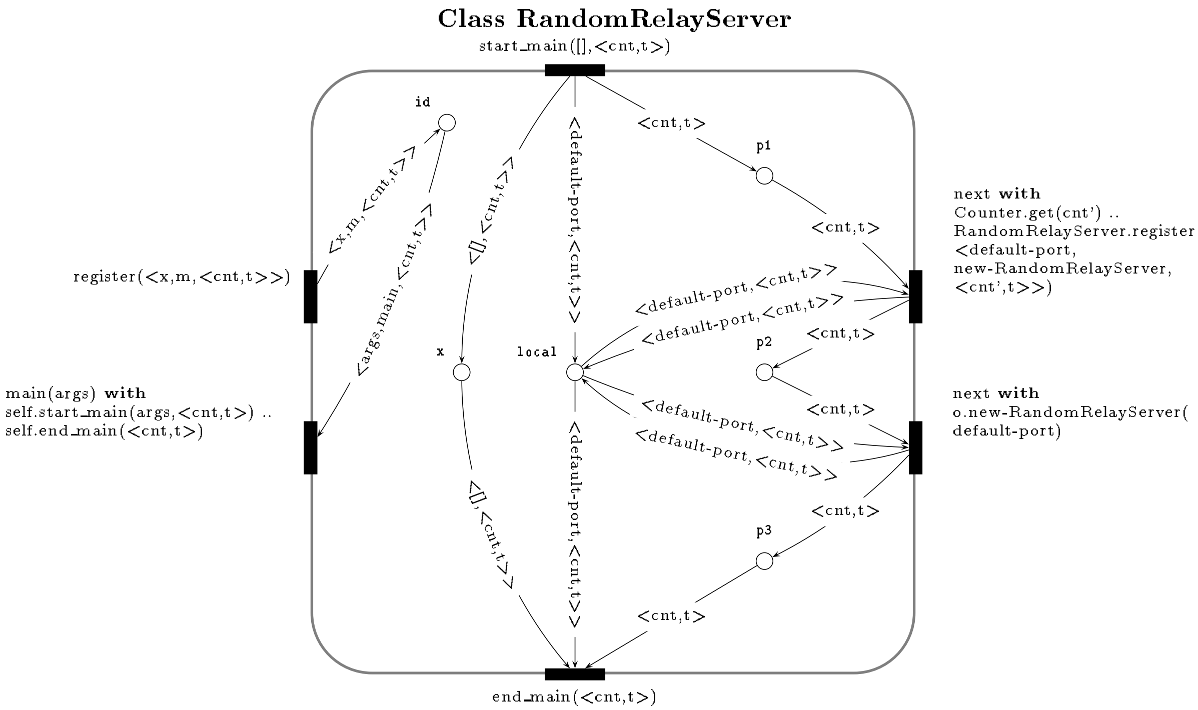


Figure 9.11: Refinement **R3**: Server

Non-default constructor `new-RandomRelayServer(port)` creates an instance `gr` of Class module `GlobalRelays` and an instance of a `JavaServerSockets` on port `port`. Method `run` of `RandomRelayServer` waits indefinitely for connections on the `JavaServerSockets`, and as soon as an applet connects, it creates two threads of class `OutputRelay`, `InputRelay` respectively connected to the applet's socket.

Additional Class modules at Server side: Class module `InputRelay` defines type `inputrelay`, it is a sub-class of Class module `JavaThreads`. The creation of an `InputRelay` thread implies the creation of an instance of `JavaDataInputStreams`. The main task of this thread is to read integers from an instance of `JavaDataInputStreams`, and to forward them to `gr` (positions 3 on figure 9.9). It is also responsible for the handling of end signals incoming from the applet.

Class module `OutputRelay` defines type `outputrelay`, it is a sub-class of Class module `JavaThreads`. The creation of an `OutputRelay` thread implies the creation of an instance of `JavaDataOutputStream`. The main task of this thread is to remove integers from `gr`, to write them to `JavaDataOutputStream` (positions 4 on figure 9.9). It is also responsible for handling end signals.

Class module `GlobalRelays` defines type `globalrelay`. It maintains a FIFO buffer by the means of an instance of `JavaVectors`. It has the same methods `put` and `get` as in refinement **R2**. These methods are `synchronized` methods, in order to protect the access to the FIFO buffer.

Applet side: Class module `DSGammaClientApp` defines type `dsgammaclientapp`. It is partially given by figure 9.12, is a sub-class of Class module `JavaApplets` (see Subsection 7.1.7). The `init` method creates instances of the following Class modules: (1) `JavaSockets`, `JavaDataInputStreams` and `JavaDataOutputStreams` (specifying the socket stream); (2) `JavaVectors` (specifying local multiset `MSInt`); (3) `TakeoffLocal`, `TakeoffGlobal`, threads (realizing the chemical reaction, the timeout, and a quitting protocol); and (4) `JavaTextFields`, `JavaTextAreas`, and `JavaButtons` (specifying elements of the GUI).

As described in 7.1.7, several extra methods, not defined in the Java program, are used in order to specify both the capture of an event, and its handling by the applet. Therefore, Class module `DSGammaClientApp` defines three methods `action_textfield(i)`, `action_stop_button`, and `action_result(i)`. These methods replace respectively methods `insert(i)`, `exit`, and `result(i)` of Class module `Applets` of refinement **R2**. Method `action_textfield(i)` is called when an integer is entered by the user into the system by the means of the instance of `TextField` provided in the GUI. Method `action_textfield(i)` simply calls method `action`, which then correctly gets the integer and stores it into `MSInt`. Similarly, method `action_stop_button` is called when the user wants to leave the system and presses the `stop_button`. Method `action_stop_button` simply calls method `action`, which handles the exit of the user. Finally, method `action_result(i)` is called when the user wants to see the result and presses the `result_button`. Method `action_result(i)` calls method `action` which prints the result (partial sum or complete sum), on an instance of Class module `JavaTextAreas`, when this button is pressed.

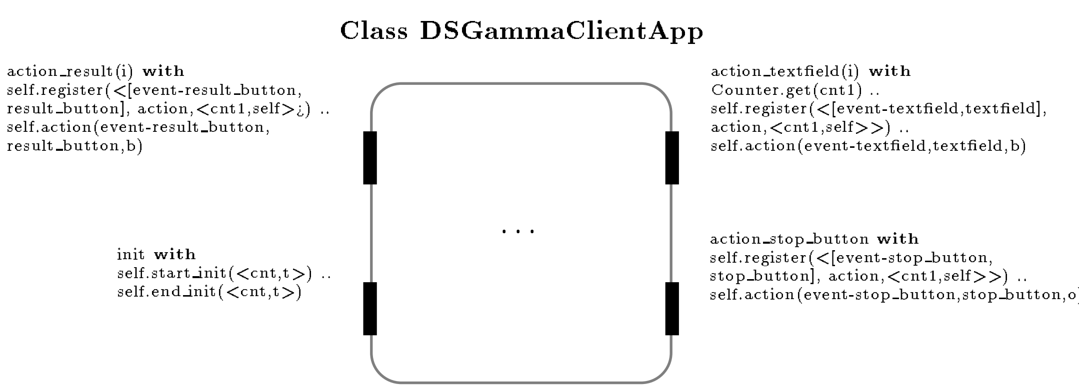


Figure 9.12: Refinement **R3**: Applet

Additional Class modules at the applet side: Class module `TakeoffLocal` defines type `takeofflocal`, and is a sub-class of Class module `JavaThreads`. An instance of `TakeoffLocal` permanently checks for integers in `MSInt`, removes one randomly and writes it to the instance of `JavaDataOutputStream` at the applet's side. It also handles end signals.

Class module `TakeoffGlobal` defines type `takeoffglobal`, and is a sub-class of Class module `JavaThreads`. An instance of `TakeoffGlobal` reads a first integer from the instance of `JavaDataInputStreams` at the applet's side. As soon as it has obtained it, it enables a timeout, and reads a second integer. If the second integer arrives before the timeout deadline, then it is added to the first one, and inserted into `MSInt`. Otherwise, a `tik` transition prevents a deadlock, by inserting the first integer into `MSInt`. It also handles end signals.

In refinement **R2**, the timeout is already specified, it is specified exactly in the same way in refinement **R3**. The quitting protocol of refinement **R2** is more simple, because there is no intermediate buffers storing integers. It is enhanced in refinement **R3**, in order to: (1) notify the server that the user wants to exit; (2) receive, from the server, integers present in the instance of `JavaDataOutputStreams` at the server's side; and finally (3) empty the local multiset `MSInt` a last time before stopping.

Communication layer: Class modules `JavaDataOuputStreams` and `JavaDataInputStreams` are used to insert or remove integers into or from a `JavaOuputStream` and a `JavaInputStream` respectively. Class modules `JavaOuputStream` and `JavaInputStream` work actually on arrays of bytes, i.e., Class module `JavaArrayBytes`. An instance of the `JavaSockets` class creates an instance of `JavaInputStreams` and an instance of `JavaOutputStreams` and realizes the TCP protocol (neither loses nor disorders the packets). Moreover, the `JavaSockets` class actually specifies the connection with a `JavaServerSockets` given a remote host and a port.

CO-OPN/2 specification **R3** is given by:

$$
\begin{aligned}
\mathbf{R3} = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Integers}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Bytes}}, \\
& (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{PairAppletIntegers}}, \\
& (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{ThreadIdentity}}, \ldots, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{PairIntegerThreadIdentity}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaObjects}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaTextFields}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaTextAreas}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaButtons}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaEvents}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaThreads}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaApplets}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaVectors}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaSockets}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaServerSockets}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaArrayBytes}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaInputStreams}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaOutputStreams}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaDataInputStreams}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaDataOutputStreams}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{TakeoffGlobal}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{TakeoffLocal}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{DSGammaClientApp}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{GlobalRelay}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{OutputRelay}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{InputRelay}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{RandomRelayServer}}, \\
& (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaStrings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaArrayStrings}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JVM}} \}.
\end{aligned}
$$

R3 is made of:

- some ADT modules necessary to define an internal behaviour close to that of a Java program (ADT module `Integers` to ADT module `PairIntegerThreadIdentity`);
- Class modules of Java basics classes needed to define parent classes of Java classes particular to the application (Class modules `JavaObjects` to `JavaVectors`);
- Class modules of Java basics classes necessary to define the sockets (Class modules `JavaSockets` to `JavaDataOutputStreams`);
- Class modules particular to the application, and needed at the client side (Class modules `TakeoffGlobal` to `DSGammaClientApp`);
- Class modules particular to the application, and needed at the server side (Class modules `GlobalRelay` to `RandomRelayServer`);
- Class modules necessary for specifying the Java Virtual Machine (Class module `JavaStrings` to `JVM`).

Contract

The contract of CO-OPN/2 specification **R3** is given by $\Phi_{\mathbf{R3}} = \{\phi_{\mathbf{R3}_1}, \dots, \phi_{\mathbf{R3}_9}\}$ below, for the set of variables $X_{\mathbf{R3}} = \{a_1, a_2, a_3\}_{\text{dsgammaclientapp}} \cup \{i, j, a, b\}_{\text{integer}} \cup \{gr\}_{\text{globalrelay}}$:

$$\phi_{\mathbf{R3}_1} = <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_2.\,\text{create}>\ \mathbf{T}$$

$$\phi_{\mathbf{R3}_2} = <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_1.\,\text{action\_textfield}(i)>\ \mathbf{T}$$

$$\begin{aligned}\phi_{\mathbf{R3}_3} = {} & <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}> \\ & <a_1.\,\text{action\_textfield}(i)><a_1.\,\text{action\_result}(i)>\ \mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_4} = {} & <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_2.\,\text{create}> \\ & <a_1.\,\text{action\_textfield}(i)\ //\ a_2.\,\text{action\_textfield}(j)> \\ & <a_1.\,\text{action\_result}(i+j)>\ \mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_5} = {} & <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_2.\,\text{create}> \\ & <a_1.\,\text{action\_textfield}(i)><a_2.\,\text{action\_textfield}(j)> \\ & <a_1.\,\text{action\_result}(i+j)>\ \mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_6} = {} & <\text{JVM . java(RandomRelayServer, [])}>\ ((<a_1.\,\text{create}><a_1.\,\text{action\_stop\_button}>) \wedge \\ & \neg(<a_1.\,\text{action\_stop\_button}><a_1.\,\text{create}>))\mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_7} = {} & <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_2.\,\text{create}> \\ & <a_1.\,\text{action\_textfield}(i)><a_1.\,\text{action\_stop\_button}><a_2.\,\text{action\_result}(i)>\ \mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_8} = {} & <\text{JVM . java(RandomRelayServer, [])}><a_1.\,\text{create}><a_2.\,\text{create}><a_3.\,\text{create}> \\ & <a_1.\,\text{action\_textfield}(i)\ //\ a_2.\,\text{action\_textfield}(j)> \\ & <a_2.\,\text{action\_result}(i)><a_1.\,\text{action\_result}(j)> \\ & <a_3.\,\text{action\_result}(i+j)>\ \mathbf{T}\end{aligned}$$

$$\begin{aligned}\phi_{\mathbf{R3}_9} = {} & <gr.\,\text{create}><gr.\,\text{put}(a)><gr.\,\text{put}(b)> \\ & (<gr.\,\text{get}(a)> \wedge\neg <gr.\,\text{get}(b)>)\mathbf{T}.\end{aligned}$$

Formulae $\phi_{\mathbf{R3}_1}$ to $\phi_{\mathbf{R3}_9}$ correspond to formulae $\phi_{\mathbf{R2}_1}$ to $\phi_{\mathbf{R2}_9}$. The only differences are the following: first `DSG` object is replaced by `JVM` object; second, methods of Class module `Applets` of refinement **R2** are replaced by methods of the form `action_textfied`, etc.

These formulae are actually properties of **R3**.

Definition 9.5.1 ***CR3***.
We define the following contractual CO-OPN/2 specification

$$\boldsymbol{CR3} = <\boldsymbol{R3}, \Phi_{\boldsymbol{R3}}>.$$

Refine Relation

Given **CR2**, **CR3** of Definitions 9.4.1 and 9.5.1 respectively, we define a CO-OPN/2 refine relation $\lambda_2 \subseteq \textsc{Elem}_{\mathbf{CR2}} \times \textsc{Elem}_{\mathbf{CR3}}$ in the following way:

$$
\begin{aligned}
\lambda_{2_{SA}} =& \{(\text{integer}, \text{integer})\}\\
\lambda_{2_{SC}} =& \{(\text{string}, \text{javastring}), (\text{arraystring}, \text{java-arraystring}),\\
& (\text{applet}, \text{dsgammaclientapp})(\text{globalrelay}, \text{globalrelay}),\\
& (\text{dsgamma-system2}, \text{jvm})\}\\
\lambda_{2_{FA}} =& \{(+_{\text{integer}}, +_{\text{integer}})\}\\
\lambda_{2_{FC}} =& \{(\text{new}_{\text{string}}, \text{new}_{\text{javastring}}), (\text{init}_{\text{string}}, \text{init}_{\text{javastring}}),\\
& (\text{new}_{\text{arraystring}}, \text{new}_{\text{java-arraystring}}), (\text{init}_{\text{arraystring}}, \text{init}_{\text{java-arraystring}}),\\
& (\text{new}_{\text{applet}}, \text{new}_{\text{dsgammaclientapp}}), (\text{init}_{\text{applet}}, \text{init}_{\text{dsgammaclientapp}}),\\
& (\text{new}_{\text{globalrelay}}, \text{new}_{\text{globalrelay}}), (\text{init}_{\text{globalrelay}}, \text{init}_{\text{globalrelay}}),\\
& (\text{new}_{\text{dsgamma-system2}}, \text{new}_{\text{jvm}}), (\text{init}_{\text{dsgamma-system2}}, \text{init}_{\text{jvm}})\}\\
\lambda_{2_M} =& \{(\text{init}_{\text{dsgamma-system2,string,arraystring}}, \text{java}_{\text{jvm,javastring,java-arraystring}}),\\
& (\text{insert}_{\text{applet,integer}}, \text{action\_textfield}_{\text{dsgammaclientapp,integer}}),\\
& (\text{result}_{\text{applet,integer}}, \text{action\_result}_{\text{dsgammaclientapp,integer}}),\\
& (\text{exit}_{\text{dsgamma-system2}}, \text{action\_stop\_button}_{\text{dsgammaclientapp}}),\\
& (\text{put}_{\text{globalrelay,integer}}, \text{put}_{\text{globalrelay,integer}}), (\text{get}_{\text{globalrelay,integer}}, \text{get}_{\text{globalrelay,integer}})\}\\
\lambda_{2_O} =& \{(\text{DSG}_{\text{dsgamma-system}}, \text{JVM}_{\text{jvm}}\}\\
\lambda_{2_X} =& \{(a_1, a_1), (a_2, a_2), (a_3, a_3), (i, i), (j, j), (a, b), (b, b), (gr, gr)\}.
\end{aligned}
$$

Refine relation λ_2 maps elements of Class module DSGammaSystem2 to elements of Class module JVM; elements of Class module Applets to elements of Class module DSGammaClientApp; and elements of Class module GlobalRelay of **R2** to elements of Class module GlobalRelayof **R3**.

Formula Refinement

Similarly to refine relation λ_1, refine relation λ_2 is essentially a renaming of methods of Class module DSGammaSystem and Applets to methods of Class module JVM and DSGammaClientApp. Formula refinement Λ_2 is simply a renaming as well. Thus, we have actually $\Lambda_2(\Phi_{\mathbf{R2}}) \subseteq \Phi_{\mathbf{R3}}$.

9.6 Implementation: The Java Program

The Java program has exactly the same classes than refinement **R3** with exactly the same behaviour.

Implementation process

The only differences with refinement **R3** are the following: first, a CO-OPN/2 transition is firable as soon as its pre-condition is fulfilled, this naturally specifies polling. In the Java program, the four thread classes: `TakeoffGlobal`, `TakeoffLocal`, `InputRelay`, `OutputRelay` use `wait`, `notify` methods in order to avoid polling. Second, CO-OPN/2 specifications of Java GUI are treated in a special way, in order to be able to specify the capture of events occurring in the GUI. Therefore, the Java source code of the applet slightly differs from CO-OPN/2 Class module `DSGammaClientApp` of refinement **R3**.

Figure 9.13 shows a snapshot of the graphical user interface provided by the applets. A user may enter several integers in the textfield, he sees the evolution of his local multiset in the textarea, he can request to see an integer by pressing the result button, and he can exit the system by pressing the exit button.

Part (a) of Figure 9.14 shows a system with a single user who has entered integers $1, 2, 3, 4$. They are firstly stored in his local multiset (maintained by the applet), and then randomly removed. Progressively sums are performed and inserted into the local multiset. Finally, the result 10 is obtained.

Part (b) of Figure 9.14 shows the arrival of a new user who does not enter any integer. The result 10, previously computed, jumps from one applet to the other (due to the timeout). Part (c) depicts the case where the second user enters integers $5, 6, 7, 8$. As for the first user, they are inserted in his local multiset, and randomly removed. Since two applets are running, some sums are computed in one applet, and some others in the other applet. Finally, the result 36 is computed.

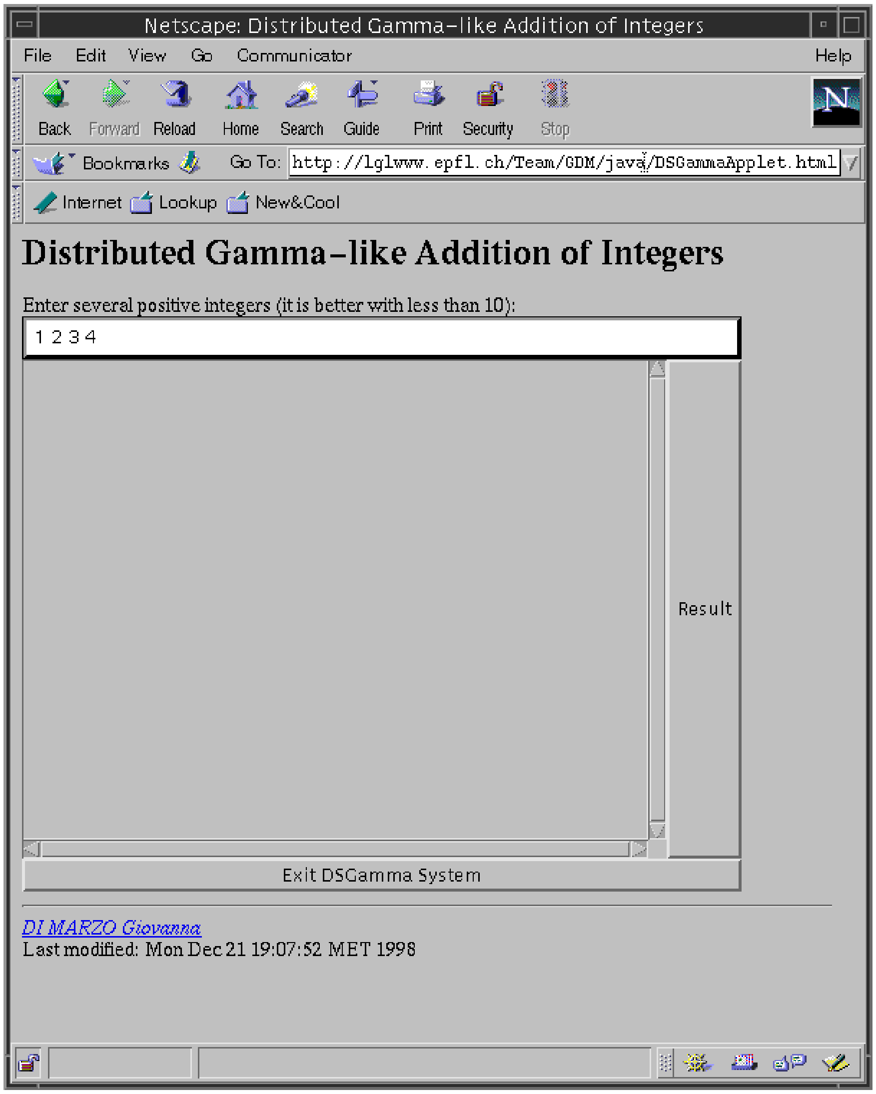
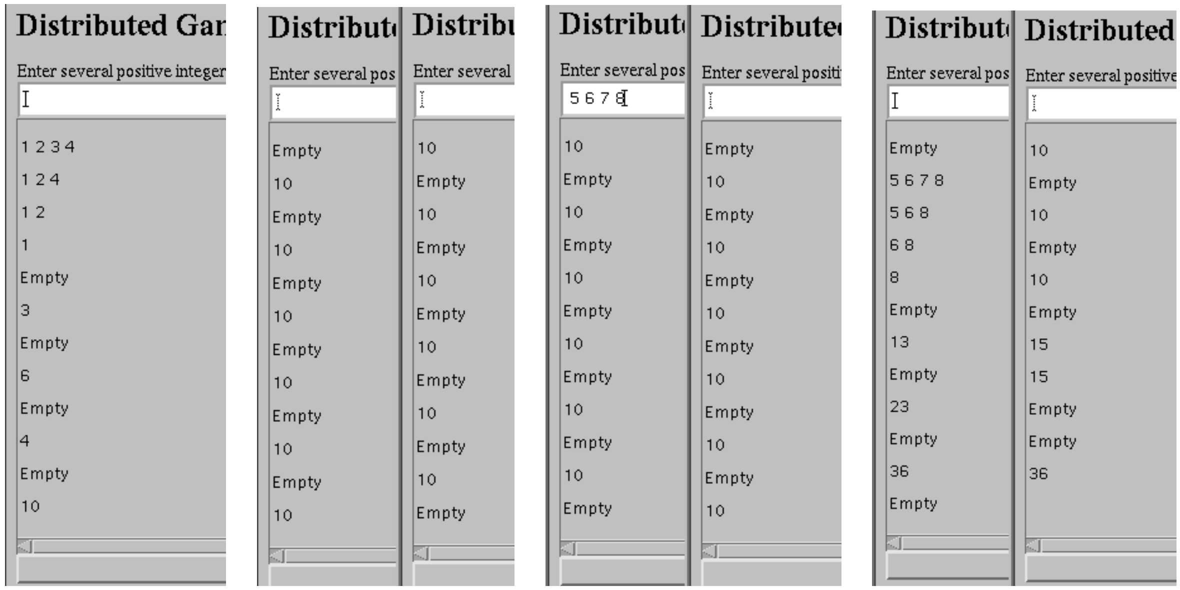


Figure 9.13: DSGamma GUI

Program

Program **Prog** is given by

$$
\begin{aligned}
\mathbf{Prog} = \{ & (Md^{\mathsf{A}}_{\mathbf{Prog}})_{\mathrm{int}}, (Md^{\mathsf{A}}_{\mathbf{Prog}})_{\mathrm{byte}}, (Md^{\mathsf{A}}_{\mathbf{Prog}})_{\mathrm{boolean}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Object}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{TextField}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{TextArea}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Button}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Event}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Thread}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Applet}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Vector}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Socket}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{ServerSocket}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{ArrayBytes}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{InputStream}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{OutputStream}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{DataInputStream}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{DataOutputStream}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{TakeoffGlobal}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{TakeoffLocal}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{DSGammaClientApp}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{GlobalRelay}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{OutputRelay}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{InputRelay}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{RandomRelayServer}}, \\
& (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{Strings}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{ArrayStrings}}, (Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{JVM}} \}.
\end{aligned}
$$



(a) (b) (c) (d)

Figure 9.14: DSGamma Application

Prog contains less ADT modules than **R3**, because **R3** needs extra ADT modules necessary to specify the internal behaviour of the Java Virtual Machine. This behaviour is not visible in a Java program source. **Prog** is made of Java classes corresponding to all CO-OPN/2 Class modules of refinement **R3** specifying Java classes. Finally, **Prog** contains `JVM` class which stands for the Java Virtual Machine itself.

Contract

Given **Prog**, and the set of variables $Y = \{a_1, a_2, a_3\}_{\text{DSGammaClientApp}} \cup \{i, j, a, b\}_{\text{int}} \cup \{gr\}_{\text{GlobalRelay}}$. Formulae ψ_1, to ψ_9 below form a contract $\Psi = \{\psi_1, \ldots, \psi_9\}$:

$$\psi_1 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])><a_1.\,\text{DSGammaClientApp}> \\ <a_2.\,\text{DSGammaClientApp}>\ \mathbf{T}$$

$$\psi_2 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])><a_1.\,\text{DSGammaClientApp}> \\ <a_1.\,\text{action\_textfield}(i)>\ \mathbf{T}$$

$$\psi_3 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])><a_1.\,\text{DSGammaClientApp}> \\ <a_1.\,\text{action\_textfield}(i)><a_1.\,\text{action\_result}(i)>\ \mathbf{T}$$

$$\psi_4 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])><a_1.\,\text{DSGammaClientApp}> \\ <a_2.\,\text{DSGammaClientApp}><a_1.\,\text{action\_textfield}(i)\ //\ a_2.\,\text{action\_textfield}(j)> \\ <a_1.\,\text{action\_result}(i+j)>\ \mathbf{T}$$

$$\psi_5 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])><a_1.\,\text{DSGammaClientApp}> \\ <a_2.\,\text{DSGammaClientApp}><a_1.\,\text{action\_textfield}(i)><a_2.\,\text{action\_textfield}(j)> \\ <a_1.\,\text{action\_result}(i+j)>\ \mathbf{T}$$

$$\psi_6 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])> \\ ((<a_1.\,\text{DSGammaClientApp}><a_1.\,\text{action\_stop\_button}>) \wedge \\ \neg(<a_1.\,\text{action\_stop\_button}><a_1.\,\text{DSGammaClientApp}>))\mathbf{T}$$

$$\psi_7 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])> \\ <a_1.\,\text{DSGammaClientApp}><a_2.\,\text{DSGammaClientApp}> \\ <a_1.\,\text{action\_textfield}(i)><a_1.\,\text{action\_stop\_button}><a_2.\,\text{action\_result}(i)>\ \mathbf{T}$$

$$\psi_8 = <\text{JVM}\,.\,\text{java}(\text{RandomRelayServer}, [\,])> \\ <a_1.\,\text{DSGammaClientApp}><a_2.\,\text{DSGammaClientApp}><a_3.\,\text{DSGammaClientApp}> \\ <a_1.\,\text{action\_textfield}(i)\ //\ a_2.\,\text{action\_textfield}(j)> \\ <a_2.\,\text{action\_result}(i)><a_1.\,\text{action\_result}(j)> \\ <a_3.\,\text{action\_result}(i+j)>\ \mathbf{T}$$

$$\psi_9 = <gr.\,\text{GlobaRelay}><gr.\,\text{put}(a)> \\ <gr.\,\text{put}(b)>\ (<gr.\,\text{get}(a)> \wedge\neg <gr.\,\text{get}(b)>)\mathbf{T}.$$

These formulae correspond to formulae $\phi_{\mathbf{R3}_1}$ to $\phi_{\mathbf{R3}_9}$. They have the same syntax, except for the `create` constructors which are replaced by the corresponding Java class names.

These formulae are satisfied by the execution of the program. Thus, we consider Ψ to be actually a contract of **Prog**. Use of testing method, as described in Chapter 8, would help to formally verify that Ψ is a contract.

Definition 9.6.1 ***CProg.***
We define the following contractual program

$$\boldsymbol{CProg} =< \boldsymbol{Prog}, \Psi > .$$

Implement Relation

Given **CR3**, **CProg** of Definitions 9.5.1 and 9.6.1 respectively, we define a CO-OPN/2 implement relation $\lambda^I \subseteq \text{ELEM}_{\mathbf{CR3}} \times \text{ELEM}_{\mathbf{CProg}}$ in the following way:

$\lambda^I_{S^A} = \{(\text{integer}, \text{int}), (\text{byte}, \text{byte}), (\text{boolean}, \text{boolean})\}$

$\lambda^I_{S^C} = \{(\text{javaobject}, \text{Object}), (\text{javatextfield}, \text{TextField}),$
$(\text{javatextarea}, \text{TextArea}), (\text{javabutton}, \text{Button}), (\text{javaevent}, \text{Event}),$
$(\text{javathread}, \text{Thread}), (\text{javaapplet}, \text{Applet}), (\text{javavector}, \text{Vector}),$
$(\text{javasocket}, \text{Socket}), (\text{javaserversocket}, \text{JavaServerSocket}),$
$(\text{java-arraybyte}, \text{ArrayBytes}),$
$(\text{javainputstream}, \text{InputStream}), (\text{javaoutputstream}, \text{OutputStream}),$
$(\text{javadatainputstream}, \text{DataInputStream}),$
$(\text{javadataoutputstream}, \text{DataOutputStream}),$
$(\text{takeoffglobal}, \text{TakeOffGlobal}), (\text{takeofflocal}, \text{TakeOffLocal}),$
$(\text{dsgammaclientapp}, \text{DSGammaClientApp}),$
$(\text{globalrelay}, \text{GlobalRelay}), (\text{outputrelay}, \text{OutputRelay}),$
$(\text{inputrelay}, \text{InputRelay}), (\text{randomrelayserver}, \text{RandomRelayServer}),$
$(\text{javastring}, \text{String}), (\text{java-arraystring}, \text{ArrayString}), (\text{jvm}, \text{JVM})\}$

$\lambda^I_{F^A} = \{(+_{\text{integer}}, +_{\text{integer}})\}$

$\lambda^I_{F^C} = \{(\text{new}_{\text{javaobject}}, \text{new}_{\text{Object}}), (\text{init}_{\text{javaobject}}, \text{init}_{\text{Object}}), \ldots,$
$(\text{new}_{\text{javasocket}}, \text{new}_{\text{Socket}}), (\text{init}_{\text{javasocket}}, \text{init}_{\text{Socket}}), \ldots,$
$(\text{new}_{\text{javasocket}}, \text{new}_{\text{Socket}}), (\text{init}_{\text{javasocket}}, \text{init}_{\text{Socket}}), \ldots,$
$(\text{new}_{\text{dsgammaclientapp}}, \text{new}_{\text{DSGammaClientApp}}), (\text{init}_{\text{dsgammaclientapp}}, \text{init}_{\text{dsgammaclientapp}}), \ldots$
$(\text{new}_{\text{randomrelayserver}}, \text{new}_{\text{RandomRelayServer}}), (\text{init}_{\text{randomrelayserver}}, \text{init}_{\text{RandomRelayServer}}), \ldots$
$(\text{new}_{\text{jvm}}, \text{new}_{\text{JVM}}), (\text{init}_{\text{jvm}}, \text{init}_{\text{JVM}})\}$

$\lambda^I_M = \{(\text{wait}_{\text{javaobject}}, \text{wait}_{\text{Object}}), (\text{notify}_{\text{javaobject}}, \text{notify}_{\text{Object}}), \ldots$
$(\text{action}_{\text{dsgammaclientapp,javaevent,javaobject,boolean}}, \text{action}_{\text{DSGammaClientApp,Event,Object,boolean}}),$
$(\text{action\_textfield}_{\text{dsgammaclientapp,integer}}, \text{action\_textfield}_{\text{DSGammaClientApp,int}}),$
$(\text{action\_result}_{\text{dsgammaclientapp,integer}}, \text{action\_result}_{\text{DSGammaClientApp,int}}),$
$(\text{action\_stop\_button}_{\text{dsgammaclientapp}}, \text{action\_stop\_button}_{\text{DSGammaClientApp}}), \ldots$
$(\text{new-RandomRelayServer}_{\text{randomrelayserver,integer}},$
$\text{RandomRelayServer}_{\text{RandomRelayServer,int}}), \ldots$
$(\text{put}_{\text{globalrelay,integer}}, \text{put}_{\text{GlobalRelay,int}}), (\text{get}_{\text{globalrelay,integer}}, \text{get}_{\text{GlobalRelay,int}}), \ldots$
$(\text{java}_{\text{jvm}}, \text{java}_{\text{JVM}})\}$

$\lambda^I_O = \{(\text{Object}_{\text{javaobject}}, \text{Object}_{\text{Object}}), \ldots$
$(\text{DSGammaClientApp}_{\text{dsgammaclientapp}}, \text{DSGammaClientApp}_{\text{DSGammaClientApp}}), \ldots$
$(\text{JVM}_{\text{jvm}}, \text{JVM}_{\text{JVM}})$

$\lambda^I_X = \{(a_1, a_1), (a_2, a_2), (a_3, a_3), (i, i), (j, j), (a, a), (b, b), (gr, gr)\}.$

Since **CR3** is very close to **CProg** every element (type name, method, Class object) of **CR3** is trivially mapped to its corresponding element in **CProg**. It is worth noting the following:

- Refine relation λ^I is defined on methods `action_result(i)`, `action_textfield(i)`, and `action_stop_button`. Indeed, $(Md^{\mathsf{C}}_{\mathbf{Prog}})_{\mathrm{DSGammaClientApp}}$ defines these methods even though they are not actually in the Java source.
- CO-OPN/2 non-default Constructor `new-RandomRelayServer(port)` is related to non-default Java creation method `RandomRelayServer(port)`.

Formula Implementation

Implement relation λ^I maps elements of **CR3** to elements of **CProg** having the same name; and CO-OPN/2 `create` constructors to Java constructors having the name of the Java class. We see easily that $\Lambda^I(\Phi_{\mathbf{R3}}) = \Psi$.

Summary

The refinement process described above is directed by the idea of implementing the system by the means of the Java programming language, and with an architecture using Java Applets. It starts with contractual CO-OPN/2 specification **CI** and ends with contractual Java program **CProg**:

- **CI** gives a centralised view of the application to develop. It deals with the problem of correctly computing the sums;
- **CR1** gives a view of the application with a distributed multiset of integers. It has to resolve the problem of correctly computing the result even though a user leaves the system;
- **CR2** gives a client/server view of the application. It solves the problem of deadlock occurring when the number of integers present in the system is less than the number of users. Therefore it introduces a timeout.
- **CR3** gives the complete CO-OPN/2 specification of the Java program. It integrates the use of sockets, and uses a two-phase protocol to correctly perform the sum when users leave the system.
- **CProg** is the Java program, close to **CR3**, and providing a graphical user interface.

Appendix B gives the CO-OPN/2 specifications **I**, **R1**, **R2**, **R3**, and the Java program **Prog**.

The refinement process integrates progressively more and more details, and enables the specifier to concentrate separately on different problems (the computing of the sum first, the quitting protocol, the deadlock, and finally the sockets). Therefore, we think that schema a development proposed here (**CI** to **CProg**) is well suited for the development of distributed Java applications.

Other Refinement Process

Starting with the same requirements and initial contractual specification **CI**, another refinement process has been realised. It is guided by the concern of satisfying certain non-functional requirements, such as making the system tolerating to certain breakdowns, as well as by constraints of design integrating the concept of a certain kind of multi-threaded transactions, called Coordinated Atomic Actions (CAAs) [62].

Reports [30, 31] contain the complete CO-OPN/2 specifications of the DSGamma system designed using CAAs.

Chapter 10

Conclusion

Model-oriented formal specifications languages allow to easily describe a model of a system to be developped, but are not well-suited for explicitly expressing properties of the system. Conversely, logical languages easily express properties, but describe a model with more difficulty. The *two languages framework*, described among others by Pnueli in [54], consists of using a logical language for expressing requirements and a model-oriented language for describing models or implementations.

Meyer [50] advocates that in order to address the correctness issue, i.e., the ability of a software to perform according to its specification, it is necessary to develop software with *built-in* features for dealing with correctness, in order to "write correct software and know it".

This thesis is based on the two languages framework as described by Pnueli, and integrates built-in features for addressing the correctness issue as proposed by Meyer. Indeed, this thesis advocates the joint use of a specifications language and a logical language, in order to perform the stepwise refinement of model-oriented specifications. The logical language enables to express a *contract* on a model-oriented system specification, i.e., a set of logical formulae, satisfied by the model of the specification. The contract has a dual function: first it semantically determines correct refinement steps; and second, it is the key for verifying the correctness of the refinement process.

10.1 Summary

This thesis defines a theoretical framework for the stepwise refinement and implementation of specifications using a two languages framework. Due to the use of two specific languages, we derive methodological results that allow to deal with the correctness issue during the whole development process. Finally, the application of the theoretical results to the CO-OPN/2 specifications language and the Hennessy-Milner logic is a first step towards a development methodology in the framework of CO-OPN/2.

Theoretical Framework

The theoretical framework necessary to define a stepwise refinement and implementation based on contracts is made of the following elements:

- *A Formal Model-Oriented Specifications Language*
  It is used to give a complete and mathematical solution (*how*) that represents to system to be developped. At each step of the refinement process it takes into account refinement choices;

- *A Logic on the Formal Specifications Language*
  It is used to express the contracts on the specifications. The contracts are sets of formulae that express the essential requirements and refinement choices (*what*) that must be kept till the implementation. A contractual specification is a pair given by a specification and a contract, such that the model of the specification part satisfies the contract;

- *A Refine Relation, A Formula Refinement, A Refinement Relation*
  The refine relation is a relation on syntactical elements of contractual specification. It expresses the syntactical changes that occur to the specifications during a refinement process.

  Given a refine relation, the formula refinement is a function able to transform a high-level contract into lower-level formulae, according to modifications required by the refine relation on the elements constituting the formulae.

  The refinement relation conveys the semantical requirements defining a correct refinement step. It is a relation on contractual specifications, that simply requires that a lower-level contract contains the translation, provided by the formula refinement, of a higher-level contract. This ensures that the model of the lower-level specification satisfies the higher-level contract, and that the high-level contract is satisfied as well by subsequent correct refinement steps;

- *A Programming Language*
  The programming language, different from the specifications language, is the language chosen for the software implementation. The choice of the programming language may affect refinement choices performed during the refinement process;

- *A Logic on the Programming Language*
  It is used to express the contract of the program. This logic is certainly different from that used for the formal specifications language, since the programming language and the formal specifications language are different;

- *An Implement Relation, A Formula Implementation, An Implementation Relation*
  The implement relation is a relation on elements of contractual specifications and elements of contractual programs. It explains the syntactical links between a contractual specification and a contractual program.

The formula implementation transforms a specification contract into formulae expressed on a program.

The implementation relation on contractual specifications and contractual programs simply requires that the program contract contain the translation of the specification contract. Therefore, the program satisfies the contract of every contractual specification obtained during the refinement process.

Methodological Results

The use of two distinct languages during a refinement process leads to the following methodological results:

- *A General Theory of Stepwise Refinement and Implementation Based on Contracts*
  It advocates the joint use of a model-oriented formal specification, and a set of logical formulae, called a contract, satisfied by the model of the specification. Correctness of a refinement step is obtained by preservation of contracts. Implementation is similarly treated;

- *Correctness as a Built-In Feature*
  The use of explicit contracts during a development process allows the specifier to recognise essential properties to preserve during a refinement step; and let the verification process be easier since the contract explicitly identifies the properties that have to be checked.

CO-OPN/2 Development Framework

The application of the general theory of refinement and implementation to the CO-OPN/2 specifications languages brings some elements useful for defining a whole development framework for CO-OPN/2:

- *A Theory of Stepwise Refinement and Implementation Based on Contracts*
  The CO-OPN/2 language expresses the system specifications, while the Hennessy-Milner logic expresses the contracts. The choice of this logic is motivated by the fact that is used in the CO-OPN/2 framework for generating test cases. The refine relation is an injective, partial function, that is total on elements of the contract; it is essentially a renaming that maintains the part of the structure of the high-level specification which is concerned by the contract. The formula refinement is a simple rewriting of the formulae based on the renaming given by the refine relation.

  The implementation is considered towards object-oriented programming languages. The implement relation and the formula implementation are defined in a similar way as the refine relation and the formula refinement;

- *Implementation of CO-OPN/2 Specifications in Java*
  Advices are given for performing a stepwise refinement based on contracts, followed by an implementation using the Java programming language. Among others, the most concrete contractual CO-OPN/2 specification reached at the end of the refinement process should specify every instruction of the program, and should convey the semantics of the Java programming language. We show how to obtain a CO-OPN/2 specification which specifies a Java program and reflects the Java semantics.

  Through a concrete case study, a whole refinement process has been realised and has lead to the development of a Java program having a client/server architecture distributed across the Web using Java applets. Guidelines for such a development process have been identified: an initial specification is provided which describes the system in a centralised manner; a first refinement step leads to a view of the system with distributed data; a second refinement step introduces the client/server architecture; and finally, a last refinement step takes into account the socket layer - necessary to communicate through a network - as well as the Java semantics;

- *Verification Using Generated Tests*
  A way of verifying the refinement steps and the implementation phase using generated tests is proposed for the CO-OPN/2 language. It consists mainly of generating test cases that are representative of the contract;

- *Towards a Methodology of Development*
  The three points above constitute starting elements for establishing a development methodology with formal proofs for the CO-OPN/2 framework (design, implementation, verification). Indeed, the work presented in this thesis can be combined with current other works (test, direct implementation of CO-OPN/2 specifications in Java, axiomatic semantics) occurring in the framework of the CO-OPN/2 language, in order to form a complete methodology of development using CO-OPN/2 specifications.

10.2 Future Works

As we have seen above, this thesis brings some elements useful for the establishment of a methodology of development in the framework of the CO-OPN/2 language. In order to actually reach this aim both theoretically and practically, the following works should be undertaken:

- *Assessment of the General Theory*
  Chapter 3 presents a general theory of refinement and implementation based on contracts, which can be applied to any model-oriented specifications language, and any logic well-suited for expressing properties on these specifications. Even though this general theory is presented independently of any specifications and logical languages, some fundamental definitions, such as the one of the refine relation, and

the formula refinement, take their motivation by the application of the theory to the CO-OPN/2 specifications language, and the Hennessy-Milner logic. In order to assess the foundation of the general theory it is necessary to confront it with other specifications and logical languages;

- *Industrial Case Studies*
  The case study described in Chapter 9 is rather an academic application. In order to identify problems that could occur during the development of more complex applications, it is necessary to put the CO-OPN/2 theory of refinement to the test with well-known examples of refinement, and with industrial case studies;

- *Enhancement of HML*
  Currently any invariant property that must be satisfied at each state (or at least at an infinite number of states) of a transition system, needs an infinite number of HML formulae to be expressed. In order to be of practical use for a specifier the current version of HML, described in this thesis, should be enhanced with some temporal operators and variables quantifiers. In that manner, a single enhanced HML formula could represent an infinite number of simple HML formulae;

- *Development of Tools*
  In order to make the work of the specifier easier, a series of tools, integrated into a homogeneous toolkit, would be very useful: (1) a tool for generating contracts by deriving simple HML formulae from enhanced HML formulae; (2) a tool for graphically editing high-level and low-level contractual specifications; for helping the specifier to build the refine relation; and for constructing the formula refinement from the refine relation; (3) a tool for proving: that the models of the specifications satisfy their contract (horizontal verification); that a low-level contract contains the translated high-level contract (vertical verification); and that the models of the program satisfy their contracts (program verification). This last tool should be related to the **Co-opnTest** tool, which automatically generates test cases;

- *Weaker Refine Relation*
  Chapter 5 defines a strong refine relation; it is functional, injective, and do not allow that a high-level Class module or ADT module is split over several lower-level Class modules of ADT modules respectively. However, in some cases, it could facilitate the refinement process, if splitting Class modules is allowed;

- *Towards an Axiomatic Verification*
  Once the axiomatic semantics for CO-OPN/2, currently studied by Buchs and Vachon [59], is established, it will be possible to propose an axiomatic verification of the correctness of the refinement process and the implementation step;

- *Another Compositional Refinement*
  This thesis proposes a hierarchical operator for composing CO-OPN/2 specifications, and a compositional refinement based on this hierarchical operator. Buffo and Buchs [23] propose a compositional semantics for CO-OPN/2 specifications. It

could be worth studying another compositional refinement, which would be based on this new compositional semantics.

The work presented in this thesis provides a theoretical basis for a development methodology using the CO-OPN/2 language. We are confident that the development of tools proposed above will considerably help a specifier, using the CO-OPN/2 language, to practically build reliable software.

Appendix A

Swiss Chocolate Factory

A.1 CO-OPN/2 Textual Specifications

Here are the CO-OPN/2 textual specifications used for running examples of Chapters 4 and 5.

```
Class PackagingUnit;
Interface
  Type packaging-unit;
  Method take;
Body
  Use Chocolate, ConveyorBelt, Packaging, PralineContainer;
  Transitions
    filling, store;
  Place
    work-bench _ : packaging;
  Axioms
    take with the-conveyor-belt.get box ::
      -> work-bench box;
    filling with
      the-praline-container. get choc .. box.fill choc ::
      work-bench box -> work-bench box;
    store with box.full-praline choc ::
      work-bench box -> ;
    where
      box: packaging;
      choc: chocolate;
End PackagingUnit;

Class PackagingProducer;
Interface
  Use  Packaging;
  Type packaging-procuder;
  Object the-packaging-producer;
Method
  produce;
Body
  Axiom
    produce with
```

```
      box.create-packaging .. the-conveyor-belt.put box :: -> ;
    where
      box: packaging;
End PackagingProducer;

Class PralineContainer;
Interface
  Use  Chocolate;
  Type praline-container;
  Object the-praline-container;
  Method get _ : praline;
Body
  Use Natural, Capacity;
  Place
    amount > : natural;
  Initial
    amount container-capcity;
  Axiom
    get p :: amount n -> amount (n-1);
    Where
      p : praline;
      n : natural;
End PralineContainer;

Class Heap;
Interface
  Use   Packaging;
  Type  heap;
  Object the-heap;
  Methods put _, get _ : packaging;
Body
  Place storage _ : packaging;
  Axioms
    put box :: -> storage box;
    get box :: storage box -> ;
  Where
    box : packaging;
End Heap;

Class ConveyorBelt;
Interface
  Use    Packaging;
  Type   conveyor-belt;
  Object the-conveyor-belt;
  Methods put _, get _ : packaging;
Body
  Use    FifoPackaging;
  Place  belt _ : fifo-packaging;
  Initial belt [];
  Axioms
    put box ::
      (size f)>conveyor-capacity = true =>
      belt f -> belt (insert box f);
    get (first f') ::
      belt f' -> belt (extract f');
    where
      f   : fifo-packaging;
      f'  : ne-fifo-packaging;
      box : packaging;
```

```
End ConveyorBelt;

Class Packaging;
Interface
  Use Chocolate;
  Type packaging;
  Methods
    fill _ : chocolate;
    full-praline;
  Creation
    create-packaging;
Body
  Use Naturals, Capacity;
  Place
    #square-holes _ : natural;
  Initial
    #square-holes praline-capacity;
  Axioms
    fill P :: #square-holes n -> #square-holes (n-1);
    full-praline :: #square-holes 0 -> #square-holes 0;
    where n: nz-natural;
End Packaging;

Class DeluxePackaging;
Inherit Packaging;
  Rename packaging -> deluxe-packaging;
Interface
  Use Packaging;
  Subtype deluxe-packaging < packaging;
  Method
    full-truffle;
  Creation
    create-packaging;
Body
  Place
    #round-holes _ : natural;
  Initial
    #square-holes praline-capacity;
    #round-holes  truffle-capacity;
  Axioms
    fill T :: #round-holes n -> #round-holes (n-1);
    full-truffle :: #round-holes 0 -> #round-holes 0;
    create-packaging :: ->
    where n : nz-natural;
End DeluxePackaging;

Adt FifoPackaging;
Interface
  Use Naturals, Packaging;
  Sorts ne-fifo-packaging, fifo-packaging;
  Subsort ne-fifo-packaging < fifo-packaging;
  Generators
    [] : -> fifo-packaging;
    insert _ _ : packaging fifo-packaging ->
                 ne-fifo-packaging;
  Operations
    first   _ : ne-fifo-packaging -> packaging;
    extract _ : ne-fifo-packaging -> fifo-packaging;
    size    _ : ne-fifo-packaging -> natural;
```

```
Body
  Axioms
    first (insert box [])  = box;
    first (insert box f)   = first f;

    extract (insert box []) = [];
    extract (insert box f)  =
                insert box (extract f);

    size [] = 0;
    size (insert box f) = 1 + (size f);

    where
      box : packaging;
      f   : ne-fifo-packaging;
End FifoPackaging;

Adt Chocolate;
Interface
  Sorts chocolate, praline, truffle;
  Subsort
    praline < chocolate;
    truffle  < chocolate;
  Generators
    P : praline;
    T : truffle;
End Chocolate;

Adt Capacity;
Interface
  Use Naturals;
  Operations
    praline-capacity  : -> natural;
    truffle-capacity  : -> natural;
    conveyor-capacity : -> natural;
Body
  Axioms
    praline-capacity = 16;
    truffle-capacity = 8;
    conveyor-capacity = 50;
End Capacity;

Adt Naturals;
Interface
  Use  Booleans;
  Sort natural;
  Generators
    0       : -> natural;
    succ _  : natural -> natural;
  Operations
    _ + _   ,
    _ - _   ,
    _ * _   ,
    _ / _   ,
    _ % _   : natural natural -> natural;
    _ = _   ,
    _ <= _  ,
    _ < _   ,
    _ > _   ,
```

```
    _ >= _  : natural natural -> boolean;
    max _ _ : natural natural -> natural;
    min _ _ : natural natural -> natural;
    even _  : natural -> boolean;
    2** _   ,
    _ ** 2  : natural -> natural;

    ;; constants
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 : -> natural;
Body
  Axioms
    0+natVar1 = natVar1;
    (succ natVar1)+natVar2 = succ (natVar1+natVar2);

    ;; substraction, if natVar2 > natVar1 then natVar1-natVar2 = 0
    0-natVar1 = 0;
    (succ natVar2)-0 = succ natVar2;
    (succ natVar2)-succ natVar1 = natVar2-natVar1;

    0*natVar1 = 0;
    (succ natVar1)*natVar2 = (natVar1*natVar2)+natVar2;

    ;; division, if natVar2 = 0 then div natVar1 natVar2 = 0
    natVar1/0 = 0;
    natVar1<natVar2  = true => natVar1/natVar2 = 0;
    natVar1>=natVar2 = true => natVar1/natVar2 =
                               succ ((natVar1-natVar2)/natVar2);

    ;; modulo, if natVar2 = 0 then mod natVar1 natVar2 = 0
    natVar1%natVar2 = natVar1-(natVar2*(natVar1/natVar2));

    0=0             = true;
    0=succ natVar1        = false;
    succ natVar1=0        = false;
    (succ natVar1)=succ natVar2 = natVar1=natVar2;

    natVar1<=natVar2 = not natVar2<natVar1;

    0<0                       = false;
    0<succ natVar1        = true;
    succ natVar1 < 0      = false;
    succ natVar1 < succ natVar2 = natVar1<natVar2;

    natVar1>natVar2 = not natVar1<=natVar2;

    natVar1>=natVar2 = not natVar1<natVar2;

    even 0 = true;
    even succ natVar1 = not even natVar1;

    2**0 = succ 0;
    2**succ natVar1 = (succ succ 0)*(2**natVar1) ;

    (natVar1<=natVar2)=true  => max natVar1 natVar2 = natVar2 ;
    (natVar1<=natVar2)=false => max natVar1 natVar2 = natVar1 ;
    (natVar1<=natVar2)=true  => min natVar1 natVar2 = natVar1 ;
    (natVar1<=natVar2)=false => min natVar1 natVar2 = natVar2 ;

    natVar1**2 = natVar1*natVar1;
```

```

    1  = succ 0;   2  = succ 1;   3  = succ 2;   4  = succ 3;
    5  = succ 4;   6  = succ 5;   7  = succ 6;   8  = succ 7;
    9  = succ 8;   10 = succ 9;   11 = succ 10;  12 = succ 11;
    13 = succ 12;  14 = succ 13;  15 = succ 14;  16 = succ 15;
    17 = succ 16;  18 = succ 17;  19 = succ 18;  20 = succ 19;

  Theorems

    ;; various properties for division and modulo
    0 / natVar1 = 0;
    (natVar1 % natVar2) / natVar2 = 0;
    0 % natVar1 = 0;
    (natVar1 % natVar2) % natVar2 = natVar1 % natVar2;


  Where
    natVar1, natVar2: natural;

Inherit EquivalenceRelation;                ;; "=" is an equivalence
 Rename
    theSort -> natural;

Inherit TotalOrderRelation;                ;; "<=" is a total order
  Rename
    theSort -> natural;

Inherit TotalOrderRelation;                ;; ">=" is a total order
  Rename
    theSort -> natural;
    _ <= _ -> _ >= _;
    max _ _ -> min _ _;
    min _ _ -> max _ _;

Inherit StrictTotalOrderRelation;        ;; "<" is a strict total order
  Rename
    theSort -> natural;

Inherit StrictTotalOrderRelation;        ;; ">" is a strict total order
  Rename
    theSort -> natural;
    _ < _ -> _ > _;

Inherit AssociativityCommutativity;
  Rename      ;; "+" is associative and commutative
    theSort -> natural;
    _ theOp _ -> _ + _;
Inherit NeutralElement;        ;; "+" has "0" as neutral element
  Rename
    theSort -> natural;
    1 -> 0;
  Undefine 1;

Inherit AssociativityCommutativity;
  Rename      ;; "*" is associative and commutative
    theSort -> natural;
    _ theOp _ -> _ * _;
Inherit NeutralElement;        ;; "*" has "1" as neutral element
  Rename theSort -> natural;
```

```
Inherit ZeroElement;                 ;; "*" has "0" as zero element
  Rename theSort -> natural;
  Undefine 0;

Inherit AssociativityCommutativity;
  Rename       ;; "max" is associative and commutative
    theSort -> natural;
    _ theOp _ -> max _ _;
End Naturals;

Adt Booleans;
Interface
  Sort boolean;
  Generators
    true    : -> boolean;
    false   : -> boolean;
  Operations
    not   _ : boolean -> boolean;
    _ and _ : boolean boolean -> boolean;
    _ or  _ : boolean boolean -> boolean;
    _ xor _ : boolean boolean -> boolean;
    _ = _   : boolean boolean -> boolean;
Body
  Axioms
    not true      = false;
    not false     = true;

    true  and booleanVar1  = booleanVar1;
    false and booleanVar1  = false;

    true  or booleanVar1   = true;
    false or booleanVar1   = booleanVar1;

    false xor booleanVar1  = booleanVar1;
    true  xor booleanVar1  = not booleanVar1;

    (true=true)   = true;
    (true=false)  = false;
    (false=true)  = false;
    (false=false) = true;

  Theorems
    ;; reflexivity
    (booleanVar1 = booleanVar1) = true;

    ;; symetry
    (booleanVar1 = booleanVar2) = true =>
        (booleanVar2 = booleanVar1) = true;

    ;; transitivity
    (booleanVar1 = booleanVar2) = true &
        (booleanVar2 = booleanVar3) = true =>
        (booleanVar1 = booleanVar3) = true;

  Where
    booleanVar1, booleanVar2, booleanVar3 : boolean;

Inherit AssociativityCommutativity;
```

```
  Rename        ;; "and" is associative and commutative
    theSort -> boolean;
    _ theOp _ -> _ and _;

Inherit AssociativityCommutativity;
  Rename        ;; "or" is associative and commutative
    theSort -> boolean;
    _ theOp _ -> _ or _;

Inherit AssociativityCommutativity;
  Rename        ;; "xor" is associative and commutative
    theSort -> boolean;
    _ theOp _ -> _ xor _;
End Booleans;
```

A.2 CO-OPN/2 Abstract Specifications

This section presents the mathematical definitions of CO-OPN/2 specifications of running examples of Chapters 4 and 5.

Example 4.1.24: *Spec*

The CO-OPN/2 specification of *Spec* of Example 4.1.24 is given by:

$$\begin{aligned} Spec = \{ & (Md^{\mathsf{A}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{Chocolate}}, (Md^{\mathsf{A}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{Capacity}}, (Md^{\mathsf{A}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{Booleans}}, \\ & (Md^{\mathsf{A}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{C}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{Packaging}}, (Md^{\mathsf{C}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{ConveyorBelt}}, \\ & (Md^{\mathsf{C}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{PralineContainer}}, (Md^{\mathsf{C}}_{\boldsymbol{\Sigma},\boldsymbol{\Omega}})_{\text{PackagingUnit}}\}. \end{aligned}$$

The global signature of *Spec* is given by:

$$\begin{aligned} \Sigma = \Big\langle & \{\text{chocolate}, \text{praline}, \text{truffle}, \text{boolean}, \text{natural}\} \ \cup \\ & \{\text{packaging}, \text{conveyor-belt}, \text{praline-container}, \text{packaging-unit}\}, \\ & \{((\text{praline}, \text{chocolate}),\ (\text{truffle}, \text{chocolate}))^*\}, \\ & \{\text{P}_{\text{praline}}, \text{T}_{\text{truffle}}, \text{conveyor-capacity}, \text{praline-capacity}, \text{truffle-capacity}, \\ & \ \text{true}, \text{false}, \text{not}, \text{and}, \text{or}, \text{xor}, =, 0, \text{succ}, +, \ldots, 1, \ldots, 20\} \ \cup \\ & \{\text{init}_{\text{packaging}}, \text{new}_{\text{packaging}}, \\ & \ \text{init}_{\text{conveyor-belt}}, \text{new}_{\text{conveyor-belt}}, \text{the-conveyor-belt}_{\text{conveyor-belt}}, \\ & \ \text{init}_{\text{praline-container}}, \text{new}_{\text{praline-container}}, \text{the-praline-container}_{\text{praline-container}}, \\ & \ \text{init}_{\text{packaging-unit}}, \text{new}_{\text{packaging-unit}}\} \Big\rangle. \end{aligned}$$

The global interface of $Spec$ is given by:

$$\Omega = \Big\langle \{\,\text{packaging}, \text{conveyor-belt}, \text{praline-container}, \text{packaging-unit}\}, \varnothing,$$
$$\{\,\text{fill}_{\text{packaging,chocolate}}, \text{full-praline}_{\text{packaging}},$$
$$\text{put}_{\text{conveyor-belt,packaging}}, \text{get}_{\text{conveyor-belt,packaging}},$$
$$\text{take}_{\text{packaging-unit}}, \text{get}_{\text{praline-container,praline}}\},$$
$$\{\,\text{the-conveyor-belt}_{\text{conveyor-belt}}, \text{the-praline-container}_{\text{praline-container}}, \}\Big\rangle.$$

Example 5.1.2: $Spec_0$

The CO-OPN/2 specification of $Spec_0$ of Example 5.1.2 is given by:

$$Spec_0 = \{(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Chocolate}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Capacity}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Booleans}},$$
$$(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Naturals}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Packaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{Heap}}\}.$$

The global signature of $Spec_0$ is given by:

$$\Sigma_0 = \Big\langle \{\,\text{chocolate}, \text{praline}, \text{truffle}, \text{boolean}, \text{natural}\} \cup \{\text{packaging}, \text{heap}\},$$
$$\{((\text{praline}, \text{chocolate}),\ (\text{truffle}, \text{chocolate}))^*\},$$
$$\{\,\text{P}_{\text{praline}}, \text{T}_{\text{truffle}}, \text{conveyor-capacity}, \text{praline-capacity}, \text{truffle-capacity},$$
$$\text{true}, \text{false}, \text{not}, \text{and}, \text{or}, \text{xor}, =, 0, \text{succ}, +, \ldots, 1, \ldots, 20\} \cup$$
$$\{\,\text{init}_{\text{packaging}}, \text{new}_{\text{packaging}}, \text{init}_{\text{heap}}, \text{new}_{\text{heap}}\}\Big\rangle.$$

The global interface of $Spec_0$ is given by:

$$\Omega_0 = \Big\langle \{\,\text{packaging}, \text{heap}\}, \varnothing,$$
$$\{\,\text{fill}_{\text{packaging,chocolate}}, \text{full-praline}_{\text{packaging}},$$
$$\text{put}_{\text{heap,packaging}}, \text{get}_{\text{heap,packaging}}, \},$$
$$\{\,\text{the-heap}_{\text{heap}}\}\Big\rangle.$$

Example 5.2.14: $Spec_1$

The CO-OPN/2 specification $Spec_1$ of Example 5.2.14 is given by:

$$\begin{aligned}Spec_1 = \{&(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{Chocolate}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{Capacity}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{Booleans}},\\ &(Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{Naturals}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{Packaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{DeluxePackaging}},\\ &(Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{FifoPackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\mathrm{ConveyorBelt}}\}.\end{aligned}$$

The global signature of $Spec_1$ is given by:

$$\begin{aligned}\Sigma_1 = \Big\langle &\{\mathrm{chocolate}, \mathrm{praline}, \mathrm{truffle},\\ &\ \mathrm{boolean}, \mathrm{natural}, \text{fifo-packaging}\} \cup \{\mathrm{packaging}, \text{deluxe-packaging}, \text{conveyor-belt}\},\\ &\{((\mathrm{praline}, \mathrm{chocolate}), (\mathrm{truffle}, \mathrm{chocolate}), (\text{deluxe-packaging}, \mathrm{packaging}))^*, \},\\ &\{\mathrm{P}, \mathrm{T}, \text{conveyor-capacity}, \text{praline-capacity}, \text{truffle-capacity},\\ &\ [], \mathrm{insert}, \mathrm{first}, \mathrm{extract}, \mathrm{size},\\ &\ \mathrm{true}, \mathrm{false}, \mathrm{not}, \mathrm{and}, \mathrm{or}, \mathrm{xor}, =, 0, \mathrm{succ}, +, \dots, 1, \dots, 20\} \cup\\ &\{\mathrm{init}_{\mathrm{packaging}}, \mathrm{new}_{\mathrm{packaging}}, \mathrm{init}_{\mathrm{heap}}, \mathrm{new}_{\mathrm{heap}}\}\Big\rangle.\end{aligned}$$

The global interface of $Spec_1$ is given by:

$$\begin{aligned}\Omega_1 = \Big\langle &\{\mathrm{packaging}, \text{deluxe-packaging}, \text{conveyor-belt}\},\\ &\{(\text{deluxe-packaging}, \mathrm{packaging}))^*\},\\ &\{\mathrm{fill}_{\mathrm{packaging},\mathrm{chocolate}}, \text{full-praline}_{\mathrm{packaging}},\\ &\ \mathrm{fill}_{\text{deluxe-packaging},\mathrm{chocolate}}, \text{full-praline}_{\text{deluxe-packaging}},\\ &\ \text{full-truffle}_{\text{deluxe-packaging}},\\ &\ \mathrm{put}_{\text{conveyor-belt},\mathrm{packaging}}, \mathrm{get}_{\text{conveyor-belt},\mathrm{packaging}}, \},\\ &\{\text{the-conveyor-belt}_{\text{conveyor-belt}}\}\Big\rangle.\end{aligned}$$

A.3 Java Source Classes

The Java source classes of Examples 6.1.8 and 6.1.24 are given below:

```
package ChocFactory;

import java.util.*;
import java.lang.*;

public class ChocFactory {
  public static void main(String argv[]){
```

```
    JavaPackaging elem;

    // Test of Class JavaHeap
    System.out.println("Test Heap");
    // Inserts 10 packaging into theheap
    for (int i=0; i<10;i++){
      elem = new JavaPackaging();
      // fills the packaging with 3 "praline"
      elem.fill(true);elem.fill(true);elem.fill(true);
      JavaHeap.theheap.insertElement(elem);
      System.out.println(elem);
    }
    // Removes 10 packaging from theheap:
    // the order of extraction is different from that of insertion
    for (int i=0; i<10;i++){
      elem = JavaHeap.theheap.removeElement();
      System.out.println(elem);
    }

    // Test of Class JavaConveyorBelt
    JavaDeluxePackaging elem2;
    System.out.println("Test ConveyorBelt");
    // Inserts 5 deluxepackaging and 5 packagings into theconveyorbelt
    for (int i=0; i<5;i++){
      elem2 = new JavaDeluxePackaging();
      // fills deluxepackaging with 1 "praline", 2 "truffle"
      elem2.fill(true);elem2.fill(false);elem2.fill(false);
      // inserts deluxepackaging
      JavaConveyorBelt.theconveyorbelt.insertElement(elem2);
      System.out.println(elem2);

      elem = new JavaPackaging();
      // fills packaging with 1 "praline"
      elem.fill(true);
      // inserts packaging
      JavaConveyorBelt.theconveyorbelt.insertElement(elem);
      System.out.println(elem);
    }
    // Removes 10 packaging from theconveyorbelt:
    // the order of extraction must be the same as the order of insertion
    for (int i=0; i<10;i++){
      elem = JavaConveyorBelt.theconveyorbelt.removeElement();
      System.out.println(elem);
    }
}
}

  class JavaHeap extends Vector{
  // Public Static Variables
  public static JavaHeap theheap = new JavaHeap();

  // Inserts a Packaging box at the end of theheap
  public static void insertElement(JavaPackaging box){
    theheap.insertElementAt(box,theheap.size());
```

```
  }

  // Removes a Packaging box at a Random Position
  public static JavaPackaging removeElement(){
    JavaPackaging elem;
    int i;
    i = (int) (Math.random() * theheap.size()) % theheap.size();
    elem = (JavaPackaging) theheap.elementAt(i);
    theheap.removeElementAt(i);
    return elem;
  }
}

 class JavaPackaging extends Object {
// Simulates the Insertion of a Praline into a Packaging box
  public void fill(boolean P){
    if (P == true) {
      System.out.println("One more Praline");}
  }
}

  class JavaConveyorBelt extends Vector{
  // Public Static Variables
  public static JavaConveyorBelt theconveyorbelt = new JavaConveyorBelt();

  // Inserts Packaging box at the end of theconveyorbelt
  public static void insertElement(JavaPackaging box){
   // Limited size
    if (theconveyorbelt.size() < 51) {
      theconveyorbelt.insertElementAt(box,theconveyorbelt.size());}
  }

  // Removes Packaging box at the beginning of theconveyorbelt
  public static JavaPackaging removeElement(){
    JavaPackaging elem;
    elem = (JavaPackaging) theconveyorbelt.elementAt(0);
    theconveyorbelt.removeElementAt(0);
    return elem;
  }
}

 class JavaDeluxePackaging extends JavaPackaging {
  // Simulates the insertion of a Praline and a Truffle
  // into DeluxePackaging box
  public void fill(boolean P){
    if (P == true) { // Praline
      super.fill(P);}
    else // Truffle
      System.out.println("One more Truffle");
  }
}
```

A.4 Java Abstract Programs

Here are the mathematical definitions of Java programs presented in Chapter 6.

Example 6.1.8: $Prog_0$

The abstract definition of program $Prog_0$ of Example 6.1.8 is given by:

$$\begin{aligned} Prog_0 = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{boolean}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{int}}, \\ & (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaPackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaHeap}}\}. \end{aligned}$$

The global signature of $Prog_0$ is given by:

$$\begin{aligned} \Sigma_{Prog_0} = \Big\langle & \{\text{boolean}, \text{int}\} \cup \{\text{JavaPackaging}, \text{JavaHeap}\}, \varnothing, \\ & \{\text{true}_{\text{boolean}}, \text{false}_{\text{boolean}}, !_{\text{boolean}}, \&_{\text{boolean}}, \&\&_{\text{boolean}}, |_{\text{boolean}}, ||_{\text{boolean}}, \dots, \\ & \quad \dots, -2_{\text{int}}, -1_{\text{int}}, 0_{\text{int}}, 1_{\text{int}}, 2_{\text{int}}, +_{\text{int}}, -_{\text{int}}, \dots, \\ & \{\text{init}_{\text{JavaPackaging}}, \text{new}_{\text{JavaPackaging}}, \text{init}_{\text{JavaHeap}}, \text{new}_{\text{JavaHeap}}\} \Big\rangle. \end{aligned}$$

The global interface of $Prog_0$ is given by:

$$\begin{aligned} \Omega_{Prog_0} = \Big\langle & \{\text{JavaPackaging}, \text{JavaHeap}\}, \varnothing, \\ & \{\text{fill}_{\text{JavaPackaging,boolean}}, \text{notify}_{\text{JavaPackaging}}, \dots, \\ & \quad \text{insertElement}_{\text{JavaHeap,JavaPackaging}}, \text{removeElement}_{\text{JavaHeap,JavaPackaging}}, \\ & \quad \text{insertElementAt}_{\text{JavaHeap,Object}}, \text{removeElementAt}_{\text{JavaHeap,Object}}, \\ & \quad \text{size}_{\text{JavaHeap,int}}, \text{notify}_{\text{JavaHeap}}, \dots\}, \\ & \{\text{theheap}_{\text{JavaHeap}}\} \Big\rangle. \end{aligned}$$

Example 6.1.8: $Prog_1$

The abstract definition of program $Prog_1$ of Example 6.1.8 is given by:

$$\begin{aligned} Prog_1 = \{ & (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{boolean}}, (Md^{\mathsf{A}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{int}}, \\ & (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaPackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaDeluxePackaging}}, (Md^{\mathsf{C}}_{\mathbf{\Sigma},\mathbf{\Omega}})_{\text{JavaConveyorBelt}}\}. \end{aligned}$$

The global signature of $Prog_1$ is given by:

$$\Sigma_{Prog_1} = \Big\langle \{\text{boolean}, \text{int}\} \cup \{\text{JavaPackaging}, \text{JavaDeluxePackaging}, \text{JavaConveyorBelt}\},$$
$$\varnothing$$
$$\{\text{true}_{\text{boolean}}, \text{false}_{\text{boolean}}, !_{\text{boolean}}, \&_{\text{boolean}}, \&\&_{\text{boolean}}, |_{\text{boolean}}, ||_{\text{boolean}}, \dots,$$
$$\dots, -2_{\text{int}}, -1_{\text{int}}, 0_{\text{int}}, 1_{\text{int}}, 2_{\text{int}}, +_{\text{int}}, -_{\text{int}}, \dots,$$
$$\{\text{init}_{\text{JavaPackaging}}, \text{new}_{\text{JavaPackaging}},$$
$$\text{init}_{\text{JavaDeluxePackaging}}, \text{new}_{\text{JavaDeluxePackaging}},$$
$$\text{init}_{\text{JavaConveyorBelt}}, \text{new}_{\text{JavaConveyorBelt}}\} \Big\rangle.$$

The global interface of $Prog_1$ is given by:

$$\Omega_{Prog_1} = \Big\langle \{\text{JavaPackaging}, \text{JavaDeluxePackaging}, \text{JavaConveyorBelt}\}, \varnothing,$$
$$\{\text{fill}_{\text{JavaPackaging,boolean}}, \text{fill}_{\text{JavaDeluxePackaging,boolean}},$$
$$\text{notify}_{\text{JavaPackaging}}, \text{notify}_{\text{JavaDeluxePackaging}}, \dots,$$
$$\text{insertElement}_{\text{JavaConveyorBelt,JavaPackaging}}, \text{removeElement}_{\text{JavaConveyorBelt,JavaPackaging}},$$
$$\text{insertElementAt}_{\text{JavaConveyorBelt,Object}}, \text{removeElementAt}_{\text{JavaConveyorBelt,Object}},$$
$$\text{size}_{\text{JavaConveyorBelt,int}}, \text{notify}_{\text{JavaConveyorBelt}}, \dots\},$$
$$\{\text{theconveyorbelt}_{\text{JavaConveyorBelt}}\} \Big\rangle.$$

A.5 A Program Execution

This is the program execution corresponding to a possible execution of $Prog_0$ and $Prog_1$ as requested by Class ChocFactory. We observe that the first test leads to an extraction order of the packaging that is different from the insertion order, while the second test the insertion and extraction orders are the same.

```
Test Heap
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc607a9
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc607e4
One more Praline
One more Praline
```

```
One more Praline
ChocFactory.JavaPackaging@1dc607d5
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc607c6
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc6080c
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc607fd
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc60843
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc60834
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc60825
One more Praline
One more Praline
One more Praline
ChocFactory.JavaPackaging@1dc6086b
ChocFactory.JavaPackaging@1dc60834
ChocFactory.JavaPackaging@1dc607d5
ChocFactory.JavaPackaging@1dc60843
ChocFactory.JavaPackaging@1dc607a9
ChocFactory.JavaPackaging@1dc607c6
ChocFactory.JavaPackaging@1dc607fd
ChocFactory.JavaPackaging@1dc6080c
ChocFactory.JavaPackaging@1dc607e4
ChocFactory.JavaPackaging@1dc60825
ChocFactory.JavaPackaging@1dc6086b
Test ConveyorBelt
One more Praline
One more Truffle
One more Truffle
ChocFactory.JavaDeluxePackaging@1dc608af
One more Praline
ChocFactory.JavaPackaging@1dc608ed
One more Praline
One more Truffle
One more Truffle
ChocFactory.JavaDeluxePackaging@1dc608e2
One more Praline
ChocFactory.JavaPackaging@1dc608d3
One more Praline
```

```
One more Truffle
One more Truffle
ChocFactory.JavaDeluxePackaging@1dc608c8
One more Praline
ChocFactory.JavaPackaging@1dc6090e
One more Praline
One more Truffle
One more Truffle
ChocFactory.JavaDeluxePackaging@1dc60903
One more Praline
ChocFactory.JavaPackaging@1dc608f4
One more Praline
One more Truffle
One more Truffle
ChocFactory.JavaDeluxePackaging@1dc6093d
One more Praline
ChocFactory.JavaPackaging@1dc6092e
ChocFactory.JavaDeluxePackaging@1dc608af
ChocFactory.JavaPackaging@1dc608ed
ChocFactory.JavaDeluxePackaging@1dc608e2
ChocFactory.JavaPackaging@1dc608d3
ChocFactory.JavaDeluxePackaging@1dc608c8
ChocFactory.JavaPackaging@1dc6090e
ChocFactory.JavaDeluxePackaging@1dc60903
ChocFactory.JavaPackaging@1dc608f4
ChocFactory.JavaDeluxePackaging@1dc6093d
ChocFactory.JavaPackaging@1dc6092e
```

Appendix B

DSGamma System

B.1 Initial Specification: I

Here is the CO-OPN/2 specification **I** described in Section 9.2.

```
Class Users;
Interface
  Use Integers;
  Methods
    insert _ : integer;
    result _ : integer;
    exit;
  Type user;
Body
  Use DSGammaSystem, BlackTockens;
  Place
    Init _ : blacktocken;
  Initial
    Init @;
  Transitions
    init;
  Axioms
    init With DSG.new-user(Self)
        :: Init @ -> ;
    insert(i) With DSG.user-action(i,Self):: -> ;
    result(i) With DSG.result(i,Self) :: ->;
    exit With DSG.user-exit(Self) :: -> ;
    Where
        i : integer;
End Users;

Class DSGammaSystem;
Interface
  Use Integers, Users, String, ArrayStrings;
  Methods
    init        _ _ : string  arraystring;
    new-user      _ : user;
    user-action _ _ : integer, user;
    result      _ _ : integer, user;
```

```
      user-exit     _ : user;
    Object  DSG: dsgamma-system;
    Type dsgamma-system;
  Body
    Use BlackTockens;
    Places
      init  _ : blacktocken;
      MSInt _ : integer;
      users _ : user;
    Transition
      ChemicalReaction;
    Axioms
      init(D'(S'(G'(a'(m'(m'(a'[]))))))),par)
          :: -> init @;
      new-user(usr)
          :: init @ -> init @, users usr;
      user-action(i,usr)
          :: users usr -> users usr, MSInt i;
      result(i,usr)
          :: users usr, MSInt i -> users usr, MSInt i;
      user-exit(usr)
          :: users usr -> ;
          ;; All the possible Chemical Reactions
      ChemicalReaction
          :: MSInt i, MSInt j -> MSInt i+j;
      Where
        i, j : integer;
        usr  : user;
        par  : arraystring;
  End DSGammaSystem;

  Adt ArrayStrings As Array(String);
  Morphism elem -> string;
  Rename   array -> arraystring;
  End ArrayStrings;

  Adt BlackTockens;
  Interface
    Generator
      @ : -> blacktocken;
    Sort
      blacktocken;
  End BlackTockens;
```

B.2 First Refinement: R1

Here is the CO-OPN/2 specification **R1** described in Section 9.3.

```
  Class DSGammaSystem1;
  Interface
    Use Integers, Users, String, ArrayStrings;
    Methods
      init        _ _ : string arraystring;
      new-user      _ : user;
      user-action _ _ : integer user;
```

```
    result          _ _ : integer user;
    user-exit         _ : user;
  Object DSG : dsgamma-system1;
  Type dsgamma-system1;
Body
  Use BagIntegers, PairUserBags, BlackTockens;
  Places
    init            _ : blacktocken;
    UsrToExit       _ : user;
    MSInt           _ : pairuserbag;
    MSIntToEmpty _ : pairuserbag;
  Transition
    CR1, CR2, CR3, CR4, CR5, CR6, CR7, CR8;
    exit;
  Axioms
    init(D’(S’(G’(a’(m’(m’(a’[]))))))),par)
        :: -> init @;
    new-user(usr)
        :: init @ -> init @, MSInt <usr {}>;
    user-action(i,usr)
        :: MSInt <usr bag> -> MSInt <usr bag ’ i>;
    result(i,usr)
        :: MSInt <usr {}’i> -> MSInt <usr {}’i>;
    user-exit(usr)
        ::  -> UsrToExit usr;
        ;; All possible Chemical Reactions
    CR1 :: MSInt <usr (bag ’ i) ’ j>
        -> MSInt <usr bag ’(i+j)>;
    CR2 :: MSInt <usr1 bag1 ’ i>, MSInt <usr2 bag2 ’ j>
        -> MSInt <usr1 bag1 ’ (i+j)>, MSInt <usr2 bag2>;
    CR3 :: MSInt <usr1 (bag1 ’ i) ’ j>, MSInt <usr2 bag2>
        -> MSInt <usr1 bag1>, MSInt <usr2 bag2 ’ (i+j)>;
    CR4 :: MSInt <usr1 bag1 ’ i>, MSInt <usr2 bag2 ’ j>,
           MSInt <usr3 bag3>
        -> MSInt <usr1 bag1>, MSInt <usr2 bag2>,
           MSInt <usr3 bag3 ’ (i+j)>;
   exit :: UsrToExit usr, MSInt <usr bag>
        -> MSIntToEmpty <usr bag>;
        ;; do not add integers in MSIntToEmpty
    CR5 :: MSInt <usr1 bag1>, MSIntToEmpty <usr2 (bag2 ’ i) ’ j>
        -> MSInt <usr1 bag1 ’ (i+j)>, MSIntToEmpty <usr2 bag2>;
    CR6 :: MSInt <usr1 bag1 ’ i>, MSIntToEmpty <usr2 bag2 ’ j>
        -> MSInt <usr1 bag1 ’ (i+j)>, MSIntToEmpty <usr2 bag2>;
    CR7 :: MSInt <usr1 bag1 ’ i>, MSInt <usr2 bag2>,
           MSIntToEmpty <usr3 bag3 ’ j>
        -> MSInt <usr1 bag1>, MSInt <usr2 (bag2 ’ i) ’ j>,
           MSIntToEmpty <usr3  bag3>;
    CR8 :: MSInt <usr1 bag1>, MSIntToEmpty <usr2 bag2 ’ i>,
           MSIntToEmpty <usr3  bag3 ’ j>
        -> MSInt <usr1 bag1 ’ (i+j)>, MSIntToEmpty <usr2 bag2>
           MSIntToEmpty <usr3  bag3>;
    Where
        bag, bag1, bag2, bag3 : baginteger;
        usr, usr1, usr2, usr3 : user;
        i, j : integer;
        par  : arraystring;
End DSGammaSystem1;
```

```
Adt BagIntegers As Bag(Integers);
Morphism
    elem -> integer;
Rename
    bag -> baginteger;
End BagIntegers;

Adt PairUserBags As Pair(Users,BagIntegers);
Morphism
    elem  -> user;
    elem2 -> baginteger;
Rename
    pair -> pairuserbag;
End PairUserBags;
```

B.3 Second Refinement: R2

Here is the CO-OPN/2 specification **R2** described in Section 9.4.

```
Class DSGammaSystem2;
Interface
  Use String, ArrayStrings, GlobalRelays;
  Methods
    init        _ _ : string arraystring;
    get-server    _ : globalrelay;
  Object DSG : dsgamma-system2;
  Type dsgamma-system2;
Body
  Places
    GR  _ : globalrelay;
  Axioms
        ;; create globarelay gr at initialization
    init(D'(S'(G'(a'(m'(m'(a'[])))))),par) With gr.Create
        :: -> GR gr;
    get-server(gr)
        :: GR gr -> GR gr;
    Where
      gr  : globalrelay;
      par : arraystring;
End DSGammaSystem2;

Class GlobalRelays;
Interface
  Use Integers;
  Methods
    put _ : integer;
    get _ : integer;
  Type globalrelay;
Body
  Use FifoIntegers;
  Places
    buffer _ : fifointeger;
  Initial
    buffer [];  ;; empty-fifo
  Axioms
```

```
    put(i) :: buffer b -> buffer b ’ i;
    get(next of (b’i)) :: buffer b ’ i -> buffer (remove from(b’i));
  Where
    i : integer;
End GlobalRelays;

Class Applets;
Interface
  Use DSGammaSystem2, Integers, GlobalRelays;
  Methods
    insert  _ : integer;
    result  _ : integer;
    exit;
  Type applet;
Body
  Use Booleans, Random, Clock, BlackTockens;
  Places
    Init         _ : blacktocken;
    store-gr     _ : globalrelay;
    MSInt, first _ : integer;
    endp         _ : boolean;
    beginning    _ : boolean;
    timeout      _ : integer;
  Transitions
    getfirst, getsecond, tik, put, init;
  Initial
    endp            false;
    beginning       true;
    Init            @;
  Axioms
        ;; retrieve gr
    init With DSG.get-server(gr)
        :: Init @ -> store-gr gr;

        ;; add new integer to MSInt
    insert(i)
        :: endp false -> endp false, MSInt i;
        ;; change flag
     exit
        :: endp false -> endp true;
            ;; get result taken from place first
    result(i)
        :: endp false, first i
        -> endp false, first i;
        ;; receives a first integer from system
        ;; provided the user has not exit
    getfirst With
        (gr.get(i) // R.random(millis) // C.clock(hour))
        :: endp false, beginning true, store-gr gr
        -> endp false, store-gr gr,
           first i, timeout (hour + millis);
        ;; user has performed an exit
    getfirst
        :: endp true, beginning true
        -> ;

        ;; receive a second integer, adds it to first and
        ;; inserts into MSInt
```

```
    getsecond With gr.get(j)
        :: first i, timeout d, store-gr gr
        -> beginning true, MSInt i+j, store-gr gr;
        ;; to prevent deadlock when no sufficient integers in the
        ;; system, add only first integer to MSInt.
    tik With C.clock(hour)
        :: (hour > d) = true
        => timeout d, first i
        -> beginning true, MSInt i;
        ;; removes integer from MSInt until no more integer
    put With gr.put(i)
        :: store-gr gr, MSInt i
        -> store-gr gr;
    Where
      gr              : globalrelay;
      i, j            : integer;
      hour, millis, d : integer;
End Applets;

Adt FifoIntegers;
Interface
  Use     Integers, Naturals;
  Sort    fifointeger, ne-fifointeger;
  Subsort ne-fifointeger -> fifointeger;
  Generators
    [] : -> fifointeger;
    _ ’ _ : integer, fifointeger -> ne-fifointeger;
  Operations
    insert _ to _ : integer, fifointeger
                -> ne-fifointeger;
    next of _ : ne-fifointeger -> integer;
    remove from _ : ne-fifointeger -> fifointeger;
Body
  Axioms
    insert i to fifo = i ’ fifo;

    next of (i ’ []) = i;
    next of (i ’ j ’ fifoVar1)
              = next of (j ’ fifoVar1);

    remove from (i ’ []) = [];
    remove from (i ’ j ’ fifoVar1)
              = i ’ (remove from (j ’ fifoVar1));
    Where
     fifo : fifo;
     i, j : elem;
End FifoIntegers;

Class Random;
Interface
  Use Integers;
  Methods
    random _ : integer;
  Object
    R : random;
  Type random;
End Random;
```

```
Class Clock;
Interface
  Use Integers;
  Methods
    clock _ : integer;
  Object
    C : clock;
  Type clock;
End Clock;
```

B.4 Third Refinement: R3

Here is the CO-OPN/2 specification **R3** described in Section 9.5.

Server Side

```
;; RandomRelayServer class
;; ----------------------------
Class RandomRelayServer;
Inherit JavaThreads;
Rename
  Thread -> RandomRelayServer;
  javathread -> randomrelayserver;
Interface
  Use JavaThreads, Integers,
      JavaArrayStrings, RegisterParameters;
  Subtype randomrelayserver -> javathread;
  Methods
    run;
    main        _ : java-arraystring;
    register    _ : registerparameter;
    getregister _ : registerparameter;
  Creation
    new-RandomRelayServer _ : integer;
Body
  Use JavaServerSockets, GlobalRelay, JavaSockets,
      InputRelay, OutputRelay, Defaults,
      ThreadIdentity,
      PairJavaSocketThreadIdentity,
      PairOutputRelayThreadIdentity,
      PairInputRelayThreadIdentity;
  Methods
    start-run     _ : threadidentity;
    start-main  _ _ : java-arraystring threadidentity;
    End-main      _ : threadidentity;
    start-new-RandomRelayServer _ _ : integer threadidentity;
    End-new-RandomRelayServer     _ : threadidentity;
  Places
        ;; Global Variables
    port          _ : integer;
    listen-socket _ : javaserversocket;
    globalrelay   _ : globalrelay;
```

```
        ;; Local Variables
    client-socket _ : pair-javasocketthreadidentity;
    outputrelay   _ : pair-outputrelaythreadidentity;
    inputrelay    _ : pair-inputrelaythreadidentity;
    id _ : registerparameter;
    p1 _ , p2 _ , p3 _ ,
    p11 _ , p12 _ , p13 _ , p14 _ , p15 _ , p16 _ , p17 _,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ : threadidentity;
  Axioms
        ;; Method register: put call into id place
    register(regpar)
        :: -> id regpar;
        ;; Remove call from id (for dynamic creations only)
    getregister(regpar)
        :: id (regpar) -> ;
        ;; Method main(): look for a call to main and
        ;; actually start the main method
    main(args) With Self.start-main(args,<cnt t>) ..
           Self.End-main(<cnt t>)
        :: id (args,main,<cnt t>) -> ;
        ;; handles input parameters and local variables
    start-main([],<cnt t>)
        ::
        -> x (<[] <cnt t>), local (<PORT <cnt t>>),
           p1 <cnt t>;
        ;; creation of an instance
    next With Counter.get(cnt') ..
           RandomRelayServer.register(
           <PORT new-RandomRelayServer <cnt' t>>)
        :: p1 <cnt t> , local (<PORT <cnt t>>)
        -> p2 <cnt t> , local (<PORT <cnt t>>);
    next With o.new-RandomRelayserver(PORT)
        :: p2 <cnt t>, local (<PORT <cnt t>>)
        -> p3 <cnt t>, local (<PORT <cnt t>>);
    End-main(<cnt t>)
        :: p3 <cnt t>, local (<PORT <cnt t>>),
           x (<[] <cnt t>>)
        -> ;
        ;; Method new-RandomRelayServer
    new-RandomRelayServer(port) ;;with
           RandomRelayServer.getregister(
           <port new-RandomRelayServer <cnt t>>) ..
           Self.start-new-RandomRelayServer(port, <cnt t>) ..
           Self.End-new-RandomRelayServer(<cnt t>)
        ::  -> ;
        ;; replaces a non precised port with default port
    start-new-RandomRelayServer(port, <cnt t>)
        :: (port = zero) = true
        =>
        -> p11 <cnt t>, port PORT;
        ;; stores the given port
    start-new-RandomRelayServer(port, <cnt t>)
        :: (port = zero) = false
         =>
        -> p11 <cnt t>, port port;
        ;; Creation of a JavaServerSocket instance
    next With Counter.get(cnt1) ;; ..
           JavaServerSocket.register(
```

```
           <port new-JavaServerSocket <cnt1 t>>)
        :: p11 <cnt t>, port port
        -> p12 <cnt t>, port port;
    next With ls.new-JavaServerSocket(port)
        :: p12 <cnt t>
        -> p13 <cnt t>, listen-socket ls;
        ;; Creation of a GlobalRelay instance
    next With Counter.get(cnt1)  ..
           GlobalRelay.register(
           <[] new-GlobalRelay <cnt1  t>>)
        :: p13 <cnt t>
        -> p14 <cnt t>;
    next With gr.new-GlobalRelay
        :: p14 <cnt t>
        -> p15 <cnt t>, globalrelay gr;
        ;; Activates its own method start (=> run)
    next With Counter.get(cnt1) ..
           Self.register(<[] start <cnt1  t>>)
        :: p15 <cnt t>
        -> p16 <cnt t>;
    next With Self.start
        :: p16 <cnt t>
        -> p17 <cnt t>;
    End-new-RandomRelayServer(<cnt t>)
        :: p17 <cnt t> -> ;
        ;; Method run()
    run With Self.start-run(<cnt t>)
        :: id <[] run <cnt t>> -> ;
    start-run(<cnt t>)
        ::
        -> p21 <cnt t>;
        ;; accepts a client connection and stores
        ;; socket
    next With Counter.get(cnt1) ..
           ls.register(<[] accept <cnt1 t>>)
        :: p21 <cnt t>, listen-socket ls,
        -> p22 <cnt t>, listen-socket ls
    next With ls.accept(cs)
        :: p22 <cnt t>, listen-socket ls
        -> p23 <cnt t>, listen-socket ls,
           client-socket <cs <cnt t>>;
        ;; Creation of an OutputRelay instance
    next With Counter.get(cnt1)  ..
           OutputRelay.register(
           <[cs,gr,STOP-TRANSMIT] new-OutputRelay <cnt1 t>>)
        :: p23 <cnt t>, client-socket <cs <cnt t>>,
           globalrelay gr
        -> p24 <cnt t>, client-socket <cs <cnt t>>,
           globalrelay <gr <cnt t>>;
    next With or.new-OutputRelay(cs,gr,STOP-TRANSMIT)
        :: p24 <cnt t>, client-socket <cs <cnt t>>,
           globalrelay gr
        -> p25 <cnt t>,  client-socket <cs <cnt t>>,
           globalrelay gr,
           outputrelay <or <cnt t>>;
        ;; Creation of an InputRelay instance
   next With Counter.get(cnt1) ..
           InputRelay.register(
```

```
          <[cs,gr,or,STOP-TRANSMIT,STOP-CONNECTION]
          new-InputRelay <cnt1 t>>)
       :: p25 <cnt t>, client-socket <cs <cnt t>>,
          globalrelay gr, outputrelay <or  <cnt t>>
       -> p26 <cnt t>, client-socket <cs <cnt t>>,
          globalrelay gr,  outputrelay <or <cnt t>>;
    next With ir.new-InputRelay(
          cs,gr,or,STOP-TRANSMIT,STOP-CONNECTION)
       :: p26 <cnt t>, client-socket <cs <cnt t>>,
          globalrelay gr, outputrelay <or <cnt t>>
       -> p21 <cnt t>, client-socket <cs <cnt t>>,
          globalrelay gr, outputrelay <or <cnt t>>,
          inputrelay <ir <cnt t>>;

       ;; this thread loops infinitely !
    next
       :: p21 <cnt t> -> ;
    Where
      port      : integer;
      ls        : javaserversocket;
      cs        : javasocket;
      gr        : globalrelay;
      ir        : inputrelay;
      or        : outputrelay;
      t                : javathread;
      args      : java-arraystring;
      cnt, cnt1, cnt': integer;
End RandomRelayServer;

;; Defaults Used for Connection
;; ---------------------------
Adt Defaults;
Interface
  Sort default;
  Generators
    PORT, REMOTE-HOST, STOP-TRANSMIT, STOP-CONNECTION
       -> default;
Body
End Defaults;

;; InputRelay class
;; ------------------------------
Class InputRelay;
Inherit  JavaThreads;
Rename
  Thread -> InputRelay;
  javathread -> inputrelay;
Interface
  Use  JavaThreads, JavaSockets, GlobalRelay,
       OutputRelay, Integers;
  Subtype inputrelay -> javathread;
  Methods
    run;
  Creation
    new-InputRelay _ _ _ _ _ : javasocket globalrelay outputrelay
                               integer integer;
Body
  Use JavaDataInputStreams, Booleans, ThreadIdentity,
```

```
      PairIntegerThreadIdentity;
  Methods
    start-run     _ : threadidentity;
    start-new-InputRelay _ _ _ _ _ _ : javasocket globalrelay
                      outputrelay integer integer threadidentity;
    End-new-InputRelay _ : threadidentity;
  Places
        ;; Global Variables
    clientsocket    _ : javasocket;
    globalrelay     _ : globalrelay;
    outputrelay     _ : outputrelay;
    stop-transmit   _ : integer;
    stop-connection _ : integer;
    datainputstream _ : javadatainputstream;
    inputstream     _ : javainputstream;
        ;; Local Variables
    elem            _ : pair-integerthreadidentity;
    p11 _ , p12 _ , p13 _ , p14 _ , p15 _ , p16 _ , p17 _ ,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ , p26 _ , p27 _ ,
    p28 _ , p29 _,  p210 _ : threadidentity;
  Axioms
        ;; Method new-InputRelay
    new-InputRelay(cs,gr,or,stop-transmit,stop-connection)  With
           InputRelay.getregister(
            <[cs,gr,or,stop-transmit,stop-connection]
            new-InputRelay <cnt t>>) ..
           Self.start-new-InputRelay(
            cs,gr,or,stop-transmit,stop-connection,<cnt t>) ..
           Self.End-new-InputRelay(<cnt t>)
        ::  -> ;
    start-new-InputRelay(cs, gr, or,
           stop-transmit, stop-connection <cnt t>)
        ::
        -> clientsocket cs, globalrelay gr,
           outputrelay or,  stop-transmit stop-transmit,
           stop-connection stop-connection,
           p11 <cnt t>;
        ;; get inputstream from socket
     next With Counter.get(cnt1)  ..
          cs.register(<[] getInputStream <cnt1 t>>)
        :: p11 <cnt t>, clientsocket cs
        -> p12 <cnt t>, clientsocket cs
     next With cs.getInputStream(In)
        :: p12 <cnt t>, clientsocket cs
        -> p13 <cnt t>, clientsocket cs,
           inputstream In;
         ;; create an instance of JavaDataInputStream using inputstream
     next With Counter.get(cnt1)  ..
           JavaDataInputStream.register(<In Create <cnt1 t>>)
        :: p13 <cnt t>, inputstream In
        -> p14 <cnt t>, inputstream In;
    next With datain.Create(In)
        :: p14 <cnt t>, inputstream In
        -> p15 <cnt t>, inputstream In,
           datainputstream datain;
        ;; starts itself
    next With Counter.get(cnt1)  ..
           Self.register(<[] start <cnt1 t>>)
```

```
        :: p15 <cnt t>
        -> p16 <cnt t>;
    next With Self.start
        :: p16 <cnt t>
        -> p17 <cnt t>;
    End-new-InputRelay(<cnt t>)
        :: p17 <cnt t> -> ;
        ;; Method run()
    run With Self.start-run(<cnt t>)
        :: id <[] run <cnt t>> -> ;
    start-run(<cnt t>)
        ::
        -> p21 <cnt t>;
        ;; waits for an integer from datain.
    next With Counter.get(cnt1)  ..
           datain.register(<[] readInt <cnt1 t>>)
        :: p21 <cnt t>, datainputstream datain,
        -> p22 <cnt t>, datainputstream datain;
    next With datain.readInt(elem)
        :: p22 <cnt t>, datainputstream datain
        -> p23 <cnt t>, datainputstream datain,
           elem <elem <cnt t>>;
        ;; if the received integer is the stop-connection
        ;; signal then stops
    next With Counter.get(cnt1)  ..
           Self.register(<[] stop <cnt1 t>>)
        :: (elem = stop-connection) = true
        => p23 <cnt t>, elem <elem <cnt t>>,
           stop-connection stop-connection
        -> p24 <cnt t>, elem <elem <cnt t>>,
           stop-connection stop-connection;
     next With Self.stop
        :: p24 <cnt t>
        -> p25 <cnt t>;
        ;; if the received integer is the stop-transmit signal
        ;; then forwards the signal to outputrelay
     next With Counter.get(cnt1)  ..
            or.register(<true setnotify-End-sending <cnt1 t>>)
        :: (elem = stop-transmit) = true
        => p23 <cnt t>, elem <elem <cnt t>>,
           stop-transmit stop-transmit, outputrelay or
        -> p26 <cnt t>, elem <elem <cnt t>>,
           stop-transmit stop-transmit, outputrelay or;
     next With or.End-setnotify-End-sending(true)
        :: p26 <cnt t>, outputrelay or
        -> p21 <cnt t>, outputrelay or;
        ;; the received integer is not a stop signal,
        ;; then forward it to globalrelay
     next With Counter.get(cnt1)  ..
            gr.register(<elem put <cnt1 t>>)
        :: ((elem = stop-transmit) = false ) and
           ((elem = stop-connection) = false ) and
        => p23 <cnt t>, elem <elem <cnt t>>,
           stop-transmit stop-transmit,
           stop-connection stop-connection,
           globalrelay gr
        -> p27 <cnt t>, elem <elem <cnt t>>,
           stop-transmit stop-transmit,
```

```
            stop-connection stop-connection,
            globalrelay gr;
      next With gr.put(elem)
         :: p27 <cnt t>, globalrelay gr
         -> p21 <cnt t>, globalrelay gr;
         ;; close socket
      next With Counter.get(cnt1)  ..
           cs.register(<[] close <cnt1 t>>)
         :: p25 <cnt t>, clientsocket cs
         -> p28 <cnt t>, clientsocket cs;
      next With cs.close
         :: p28 <cnt t>, clientsocket cs
         -> p29 <cnt t>, clientsocket cs
      next With Counter.get(cnt1)  ..
            Self.register(<[] stop <cnt1 t>>)
         :: p29 <cnt t>
         -> p210 <cnt t>;
      next With Self.stop
         :: p210 <cnt t>
         -> ;
     Where
       cs        : javasocket;
       gr        : globalrelay;
       or        : outputrelay;
       datain    : javadatainputstream;
       In        : javainputstream;
       elem      : integer;
       t         : javathread;
       cnt1, cnt : integer;
       stop-transmit, stop-connection : integer;
End InputRelay;

;; GlobalRelay class
;; ----------------------------
Class GlobalRelay;
Inherit JavaThreads;
Rename
  Thread -> GlobalRelay;
  javathread -> globalrelay;
Interface
  Use  JavaThreads, Integers;
  Methods
    put _ : integer;
    get _ : integer;
  Creation
    new-GlobalRelay;
Body
  Use ThreadIdentity, JavaVectors, PairIntegerThreadIdentity;
  Methods
    start-put           _ _ : integer threadidentity;
    End-put               _ : threadidentity;
    start-get             _ : threadidentity;
    End-get             _ _ : integer threadidentity;
    start-new-GlobalRelay _ : threadidentity;
    End-new-GlobalRelay   _ : threadidentity;
  Places
        ;; Global Variables
    buffer        _ : javavector;
        ;; Local Variables
```

```
    input-elem    _ : pair-integerthreadidentity;
    elem-to-relay _ : pair-integerthreadidentity;
    p11 _ , p12 _ , p13 _ ,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _
    p31 _ , p32 _ , p33 _ , p34 _ , p35 _  : threadidentity;
  Axioms
        ;; Method new-GlobalRelay
    new-GlobalRelay With
           GlobalRelay.getregister(<cnt t>) ..
           Self.start-new-GlobalRelay(<cnt t>) ..
           Self.End-new-GlobalRelay(<cnt t>)
        :: -> ;
    start-new-GlobalRelay(<cnt t>) ::
        -> p11 <cnt t>;
        ;; create an instance of JavaVector
    next With Counter.get(cnt1)  ..
           JavaVector.register(<[] Create <cnt1 t>>)
        :: p11 <cnt t>
        -> p12 <cnt t>;
    next With b.Create
        :: p12 <cnt t>
        -> p13 <cnt t>, buffer b;
    End-new-GlobalRelay(<cnt t>)
        :: p13 <cnt t> -> ;

        ;; Method put(i)
    put(input-elem) With
           Self.start-put(input-elem,<cnt t>) ..
           Self.End-put(<cnt t>)
        :: id <input-elem put <cnt t>> -> ;
        ;; put is synchronized !!!
    start-put(input-elem <cnt t>)
        :: -> p21 <cnt t>, input-elem <input-elem <cnt t>>;
        ;; acquires the lock
    next With Self.lock(t)
        :: p21 <cnt t>
        -> p22 <cnt t>;
        ;; add input-elem at the end of b
    next With Counter.get(cnt1)  ..
           b.register(<input-elem addElement <cnt1 t>>)
        :: p22 <cnt t>, buffer b,
           input-elem <input-elem <cnt t>>
        -> p23 <cnt t>, buffer b,
           input-elem <input-elem <cnt t>>;
    next With b.addElement(input-elem)
        :: p23 <cnt t>, buffer b,
           input-elem <input-elem <cnt t>>
        -> p24  <cnt t>, buffer b;
        ;; releases the lock
    next With Self.unlock(t)
        :: p24 <cnt t>
        -> p25 <cnt t>;
    End-put(<cnt t>)
        :: p25 <cnt t> -> ;
        ;; Method get(i)
    get(elem-to-relay) With
           Self.start-get(<cnt t>) ..
           Self.End-get(elem-to-relay,<cnt t>)
```

```
        :: id <[] get <cnt t>> -> ;
            ;; get is synchronized !!!
    start-get(<cnt t>)
        ::
        -> p31 <cnt t>;
        ;; acquires the lock
    next With Self.lock(t)
        :: p31 <cnt t>
        -> p32 <cnt t>;
        ;; get first integer from b
    next With Counter.get(cnt1)  ..
           b.register(<0  elementAt <cnt1 t>>)
        :: p32 <cnt t>, buffer b
        -> p33 <cnt t>, buffer b;
    next With b.elementAt(0,elem-to-relay,<cnt1 t>))
        :: p33 <cnt t>, buffer b
        -> p34 <cnt t>, elem-to-relay <elem-to-relay <cnt t>> ;
        ;; releases the lock
    next With Self.unlock(t)
        :: p34 <cnt t>
        -> p35 <cnt t>;
    End-get(elem-to-relay, <cnt t>)
        :: p35 <cnt t>,
           elem-to-relay <elem-to-relay <cnt t>>
        -> ;
    Where
      b             : javavector;
      input-elem    : integer;
      elem-to-relay : integer;
      t             : javathread;
      cnt, cnt1     : integer;
End GlobalRelay;

;; OutputRelay class
;; -----------------------
Class OutputRelay;
Inherit  JavaThreads;
Rename
  Thread -> OutputRelay;
  javathread -> outputrelay;
Interface
  Use  JavaThreads, JavaSockets, GlobalRelay,
       Booleans, Integers;
  Methods
    run;
    setnotify-End-sending  _ : boolean;
  Creation
    new-OutputRelay   _ _ _  : javasocket globalrelay
                                 integer;
Body
  Use JavaDataOutputStream, ThreadIdentity,
      PairIntegerThreadIdentity;
  Methods
    start-run                        _ : threadidentity;
    start-setnotify-End-sending _ _ : boolean threadidentity
    End-setnotify-End-sending      _ : threadidentity;
    start-new-OutputRelay   _ _ _ _ : javasocket globalrelay integer
                                      threadidentity;
```

```
    End-new-OutputRelay           _ : threadidentity;
 Places
            ;; Global Variables
    client            _ : javasocket;
    globalrelay       _ : globalrelay;
    stop-transmit     _ : integer;
    End-sending       _ : boolean;
    dataoutputstream  _ : javadataoutputstream;
    outputstream      _ : javaoutputstream;
        ;; Local Variables
    elem              _ : pair-integerthreadidentity;
    p11 _ , p12 _ , p13 _ , p14 _ , p15 _ , p16 _ , p17 _,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ ,
    p31  _  : threadidentity;
  Initial
    End-sending false;
  Axioms
        ;; Method new-OutputRelay
    new-OutputRelay(cs,gr,stop-transmit) With
           OutputRelay.getregister(
            <[cs,gr,stop-transmit] new-OutputRelay <cnt t>>) ..
           Self.start-new-OutputRelay(
            cs,gr,stop-transmit,<cnt t>) ..
           Self.End-new-OutputRelay(<cnt t>)
        ::  -> ;
    start-new-OutputRelay(cs,gr,stop-transmit,<cnt t>)
        ::
        -> p11 <cnt t>, client cs, globalrelay gr,
           stop-transmit stop-transmit;
        ;; get outputstream from socket
     next With Counter.get(cnt1)  ..
          cs.register(<[] getOutputStream <cnt1 t>>)
        :: p11 <cnt t>, client cs
        -> p12 <cnt t>, client cs
     next With cs.getOutputStream(out)
        :: p12 <cnt t>, client cs
        -> p13 <cnt t>, client cs,
           outputstream out;
        ;; create an instance of DataOutputStream
    next With Counter.get(cnt1)  ..
           JavaDataOutputStream.register(<out Create<cnt1 t>>)
        :: p13 <cnt t>, outputstream out
        -> p14 <cnt t>, outputstream out
    next With dataout.Create(out)
        :: p14 <cnt t>, outputstream out
        -> p15 <cnt t>, outputstream out,
           dataoutputstream dataout;
        ;; starts itself
    next With Counter.get(cnt1)  ..
           Self.register(<[] start <cnt1 t>>)
        :: p15 <cnt t>
        -> p16 <cnt t>;
    next With Self.start
        :: p16 <cnt t>
        -> p17 <cnt t>;
    End-new-InputRelay(<cnt t>)
        :: p17 <cnt t> -> ;

        ;; Method run()
```

```
    run With Self.start-run(<cnt t>)
        :: id <[] run <cnt t>> -> ;
    start-run(<cnt t>)
      ::
        -> p21 <cnt t>;
        ;; if stop-transmit then write it on dataout and stop
    next With Counter.get(cnt1)  ..
           dataout.register(<stop-transmit writeInt <cnt1 t>>)
        :: p21 <cnt t>, End-sending true, dataoutputstream dataout,
           stop-transmit stop-transmit
        -> p22 <cnt t>, End-sending true, dataoutputstream dataout
           stop-transmit stop-transmit;
    next With dataout.writeInt(stop-transmit)
        :: p22 <cnt t>, dataoutputstream dataout,
           stop-transmit stop-transmit
        -> p23 <cnt t>, dataoutputstream dataout,
           stop-transmit stop-transmit;
    next With Counter.get(cnt1)  ..
           Self.register(<[] stop <cnt1 t>>)
        :: p23 <cnt t>
        -> p24 <cnt t>;
    next With Self.stop
        :: p24 <cnt t>
        -> ;

        ;; if not stop-transmit, then take integer from
        ;; globalrelay and loop (go to p21)
    next With Counter.get(cnt1)  ..
            gr.register(<[] get <cnt1 t>>)
        :: p21 <cnt t>, End-sending false,
           globalrelay gr
        -> p25 <cnt t>, End-sending false,
           globalrelay gr;
    next With gr.get(elem)
        :: p25 <cnt t>, globalrelay gr
        -> p21 <cnt t>, globalrelay gr,
           elem <elem <cnt t>>;
        ;; Method setnotify-end-sending()
    setnotify-End-sending(value) With
           Self.start-setnotify-End-sending(value, <cnt t>) ..
           Self.End-setnotify-End-sending(<cnt t>)
        :: id <value setnotify-End-sending <cnt t>> -> ;
    start-setnotify-End-sending(value, <cnt t>)
        :: End-sending old-value
        -> p31 <cnt t>, End-sending value;
    End-setnotify-End-sending(<cnt t>)
        :: p31 <cnt t> -> ;
    Where
      cs                 : javasocket;
      gr                 : globalrelay;
      stop-transmit      : integer;
      out                : javaoutputstream;
      dataout            : javadataoutputstream;
      value, old-value   : noolean;
      t                  : javathread;
      cnt1, cnt          : integer;
End OutputRelay;
```

Client Side

```
;; DSGammaClientApp Class
;; ------------------------
Class DSGammaClientApp;
Inherit JavaApplets;
Rename
  Applet -> DSGammaClientApp;
  javaapplet -> dsgammaclientapp;
Interface
  Use JavaApplets, Integers, JavaEvents, Booleans;
  Methods
    action           _ : javaevent javaobject boolean;
        ;; extra methods
    action-textfield _ : integer;
    action-result    _ : integer;
    action-stop-button;
Body
  Use Defaults, TakeoffGlobal, TakeoffLocal,
      JavaSockets, JavaDataInputStreams, JavaDataOutputStreams,
      JavaInputStreams, JavaOutputStreams,
      JavaVectors, ThreadIdentity,
      PairIntegerThreadIdentity;
  Methods
    start-action      _ _ : javaevent javaobject threadidentity;
    End-action          _ : boolean threadidentity;
  Places
        ;; Global Variables
    socket           _ : javasocket;
    datainputstream  _ : javadatainputstream;
    dataoutputstream _ : javadataoutputstream;
    inputstream      _ : javainputstream;
    outputstream     _ : javaoutputstream;
    MSInt            _ : javavector;
    takeofflocal     _ : takeofflocal;
    takeoffglobal    _ : takeoffglobal;

    port             _ : integer;
    host             _ : javastring;
    stop-transmit    _ : integer;
    stop-connection  _ : integer;
        ;; Local Variables
    entering-int     _ : pair-integerthreadidentity;
    result           _ : pair-integerthreadidentity;
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ , p26 _ , p27 _ , p28 _ ,
    p29 _ , p210 _ , p211 _ , p212 _ , p213 _ , p214 _ ,
    p215 _ , p216 _ ,
    p31 _ , p32 _ , p33 _ , p34 _ , p35 _ ,
    p41 _ , p42 _ , p43 _ ,
    p51 _ , p52 _ , p53 _ , p54 _ ,
    p61 _ , p62 _ , p63 _ : threadidentity;
  Initial
    port            PORT;
    stop-transmit   STOP-TRANSMIT;
    stop-connection STOP-CONNECTION;
    host            REMOTE-HOST;
  Axioms
        ;; respecify JavaApplet.init
```

```
    init With Self.start-init(<cnt t>) ..
           Self.End-init(<cnt t>)
        :: id <[] <cnt t>>
        -> ;
        ;; respecify JavaApplet.start-init
    start-init(<cnt t>)
        ::
        -> p21 <cnt t>;
        ;; creates a socket
    next With Counter.get(cnt1)  ..
           JavaSocket.register(<[host,port] Create <cnt1 t>>)
        :: p21 <cnt t>,
           host host, port port
        -> p22 <cnt t>,
           host host, port port;
    next With s.Create(host,port)
        :: p22 <cnt t>,
           host host, port port
        -> p22 <cnt t>,
           host host, port port, socket s;
        ;; gets JavaInputStream associated to the socket
    next With Counter.get(cnt1)  ..
            s.register(<[] getInputStream <cnt1 t>>)
        :: p23 <cnt t>, socket s
        -> p24 <cnt t>, socket s;
    next With s.getInputStream(In)
        :: p24 <cnt t>, socket s
        -> p25 <cnt t>, socket s, inputstream In;
        ;; creates an instance of JavaDataInputStream
    next With Counter.get(cnt1)  ..
            JavaDataInputStream.register(<In Create <cnt1 t>>)
        :: p25 <cnt t>, inputstream In
        -> p26 <cnt t>, inputstream In;
    next With datain.Create(In)
        :: p26 <cnt t>, inputstream In
        -> p27 <cnt t>, inputstream In,
           datainputstream datain;
         ;; get JavaOutputStream associated to the socket
    next With Counter.get(cnt1)  ..
           s.register(<[] getOutputStream <cnt1 t>>)
        :: p27 <cnt t>, socket s
        -> p28 <cnt t>, socket s;
    next With s.getOutputStream(out)
        :: p28 <cnt t>, socket s
        -> p29 <cnt t>, socket s, outputstream out;
        ;; creates an instance of JavaDataOutputStream
    next With Counter.get(cnt1)  ..
           JavaDataOutputStream.register(<out Create <cnt1 t>>)
        :: p29 <cnt t>,  outputstream out
        -> p210 <cnt t>, outputstream out;
    next With dataout.Create(out)
        :: p210 <cnt t>, outputstream out
        -> p211 <cnt t>, outputstream out,
           dataoutputstream dataout;

              ;; Creates an instance of JavaVector
    next With Counter.get(cnt1)  ..
```

```
            JavaVector.register(<[] Create <cnt1 t>>)
         :: p211 <cnt t>
         -> p212 <cnt t>;
     next With MSInt.Create
         :: p212 <cnt t>
         -> p213 <cnt t>, MSInt MSInt;
         ;; ... Creates an instance of JavaTextField,
         ;; JavaTextArea, and two instances of JavaButton
         ;; Creates an instance of  TakeoffLocal
     next With Counter.get(cnt1)  ..
            TakeoffLocal.register(
             <[dataout,MSInt,textarea,stop-connection]
              new-TakeoffLocal <cnt1 t>>)
         :: p212 <cnt t>, dataoutputstream dataout,
            MSInt MSInt, textarea textarea,
            stop-connection stop-connection
         -> p213 <cnt t>, dataoutputstream dataout,
            MSInt MSInt, textarea textarea,
            stop-connection stop-connection;
     next With takeofflocal.new-TakeoffLocal(
            dataout,MSInt,textarea,stop-connection)
         :: p213 <cnt t>, dataoutputstream dataout,
            MSInt MSInt, textarea textarea,
            stop-connection stop-connection
         -> p214 <cnt t>, dataoutputstream dataout,
            MSInt MSInt, textarea textarea,
            stop-connection stop-connection,
            takeofflocal takeofflocal;
         ;; Creates an instance of  TakeoffGlobal
     next With Counter.get(cnt1)  ..
            TakeoffGlobal.register(
            <[datain,MSInt,textarea,takeofflocal,stop-transmit]
             new-TakeoffGlobal <cnt1 t>>)
         :: p214 <cnt t>, datainputstream datain,
            MSInt MSInt, textarea textarea, takeofflocal takeofflocal,
            stop-transmit stop-transmit
         -> p215 <cnt t>, datainputstream datain,
            MSInt MSInt, textarea textarea, takeofflocal takeofflocal,
            stop-transmit stop-transmit;
    next With takeoffglobal.new-TakeoffGlobal(
            datain,MSInt,textarea,takeofflocal,stop-transmit)
         :: p215 <cnt t>, datainputstream datain,
            MSInt MSInt, textarea textarea, takeofflocal takeofflocal,
            stop-transmit stop-transmit
         -> p216 <cnt t>, datainputstream datain,
            MSInt MSInt, textarea textarea, takeofflocal takeofflocal,
            stop-transmit stop-transmit,
            takeoffglobal takeoffglobal;
         ;; respecify JavaApplet.end-init
     End-init(<cnt t>)
         :: p216 <cnt t>
         -> ;
         ;; respecify JavaApplet.start-stop
     start-stop(<cnt t>)
         :: -> p31 <cnt t>;
         ;; close datainputstream
     next With Counter.get(cnt1) ..
```

```
          datain.register(<[] close <cnt1 t>>)
       :: p31 <cnt t>, datainputstream datain
       -> p32 <cnt t>, datainputstream datain;
   next With datain.close
       :: p32 <cnt t>, datainputstream datain
       -> p33 <cnt t>;
       ;; close dataoutputstream
   next With Counter.get(cnt1)  ..
          dataout.register(<[] close <cnt1 t>>)
       :: p33 <cnt t>, dataoutputstream dataout
       -> p34 <cnt t>, dataoutputstream dataout;
   next With dataout.close
       :: p34 <cnt t>, dataoutputstream dataout
       -> p35 <cnt t>;
       ;; close socket
   next With Counter.get(cnt1)  ..
          s.register(<[] close <cnt1  t>>)
       :: p33 <cnt t>, socket s
       -> p34 <cnt t>, socket s;
   next With s.close
       :: p34 <cnt t>, socket s
       -> p35 <cnt t>;

       ;; respecify JavaApplet.end-stop
   End-stop(<cnt t>)
       :: p35 <cnt t> -> ;
       ;; Method action-textfield
   action-textfield(i) With Counter.get(cnt1)  ..
          Self.register(<[event-textfield,textfield]
                        action <cnt1 Self>>) ..
          Self.action(event-textfield,textfield,b)
       :: -> entering-int <i <cnt1,Self>>;
       ;; Method action-stop-button
   action-stop-button With Counter.get(cnt1)  ..
          Self.register(<[event-stop-button,
            stop-button] action <cnt1 Self>>) ..
          Self.action(event-stop-button,stop-button,b)
       :: -> ;
       ;; Method action-result
   action-result(i) With Counter.get(cnt1) ..
          Self.register(<[event-result-button,
            result-button] action <cnt1 Self>>) ..
          Self.action(event-result-button,result-button,b)
       :: result <i<cnt1 Self>> -> ;
       ;; Method action
   action(e,o,b) With
       Self.start-action(e,o,<cnt t>) ..
       Self.End-action(b,<cnt t>)
       :: id <[e,o] action <cnt t>>
       -> ;
       ;; event coming from textfield: user enters an integer
   start-action(event-textfield,textfield,<cnt t>)
       ::
       -> p41 <cnt t>;
       ;; add new integer to MSInt
   next With Counter.get(cnt1) ..
```

```
            MSInt.register(<i addElement <cnt1, t>))
         :: p41 <cnt t>, entering-int <i <cnt t>>,
            MSInt MSInt
         -> p42 <cnt t>, entering-int <i <cnt t>>,
            MSInt MSInt;
     next With MSInt.addElement(i)
         :: p42 <cnt t>, entering-int <i <cnt t>>,
            MSInt MSInt;
         -> p43 <cnt t>, MSInt MSInt;
     End-action(true,<cnt t>)
         :: p43 <cnt t> -> ;
         ;; event coming from stop-button: user wants to exit
     start-action(event-stop-button,stop-button,<cnt t>)
         :: -> p61 <cnt t>;
         ;; send stop-transmit signal to server
     next With Counter.get(cnt1)  ..
            dataout.register(<stop-transmit writeInt <cnt1 t>>)
         :: p61 <cnt t>, stop-transmit stop-transmit,
            dataoutputstream dataout
         -> p62 <cnt t>, stop-transmit stop-transmit,
            dataoutputstream dataout;
     next With dataout.writeInt(stop-transmit)
         :: p62 <cnt t>, stop-transmit stop-transmit,
            dataoutputstream dataout
         -> p63 <cnt t>, stop-transmit stop-transmit,
            dataoutputstream dataout;
     End-action(true,<cnt t>)
         :: p63 <cnt t>  ->;
         ;; event coming from result-button: user wants to see result
     start-action(event-result-button,result-button,<cnt t>)
         :: -> p51 <cnt t>;
         ;; reads an integer in MSInt
     next With Counter.get(cnt1)  ..
            MSInt.register(<0 elementAt <cnt1 t>>)
         :: p52 <cnt t>, MSInt MSInt
         -> p53 <cnt t>, MSInt MSInt;
     next With MSInt.elementAt(0,i)
         :: p53 <cnt t>, MSInt MSInt
         -> p54 <cnt t>, MSInt MSInt, result <i <cnt t>>;
     End-action(true,<cnt t>)
         :: p54 <cnt t> ->;
     Where
       t            : javathread;
       s            : javasocket;
       In           : javainputstream;
       out          : javaoutputstream;
       datain       : javadatainputstream;
       dataout      : javadataoutputstream;
       takeofflocal : takeofflocal;
       takeoffglobal: takeoffglobal;
       MSInt        : javavector;
       cnt, cnt1    : integer;
       i            : integer;
       host         : javastring;
       port         : integer;
       b            : boolean;
 End DSGammaClientApp;
```

```
;; TakeoffLocal class
;; ------------------------
Class TakeoffLocal;
Inherit JavaThreads;
Rename
  Thread -> TakeoffLocal;
  javathread -> takeofflocal;
Interface
  Use   JavaThreads, Integers, JavaDataOutputStreams,
        JavaVectors, JavaTextAreas, Booleans;
  Methods
    run;
    set-End-reception _ : boolean;
  Creation
    new-TakeoffLocal _ _ _ _ : javadataoutputstream javavector
                                 javatextarea integer;
Body
  Use Random, PairIntegerThreadIdentity, ThreadIdentity;
  Methods
    start-run                    _ : threadidentity;
    start-set-End-reception    _ _ : boolean threadidentity;
    End-set-End-reception        _ : threadidentity;
    start-new-TakeoffLocal _ _ _ - : javadataoutputstream javavector
                        javatextarea integer threadidentity;
    End_new-TakeoffLocal         _ : threadidentity;
  Places
           ;; Global Variables
    End-reception        _ : boolean;
    dataoutputstream     _ : javadataoutputstream;
    MSInt                _ : javavector;
    textarea             _ : javatextarea;
    stop-connection      _ : integer;
        ;; Local Variables
    random, elem-to-send _ : pair-integerthreadidentity;
    p11 _ , p12 _ , p13 _ ,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ , p26 _ , p27 _ , p28 _ ,
    p29 _ , p210 _ , p211 _ , p212 _ , p213 _ , p214 _ ,
    p31  __ : threadidentity;
  Initial
    End-reception  false;
  Axioms
        ;; Method new-TakeoffLocal
    new-TakeoffLocal(dataout, MSInt, textarea, stop-connection) With
          TakeoffLocal.getregister(
           <[dataout, MSInt, textarea, stop-connection]
            new-TakeoffLocal <cnt t>>) ..
          Self.start-new-TakeoffLocal(dataout, MSInt, textarea,
              stop-connection,<cnt t>) ..
          Self.End-new-TakeoffLocal(<cnt t>)
        :: -> ;
    start-new-TakeoffLocal(dataout,MSInt,
                   textarea,stop-connection,<cnt t>)
        ::
        -> p11 <cnt t>,
           dataoutputstream dataout, MSInt MSInt,
           textarea textarea, stop-connection stop-connection;
        ;; starts itself
    next With Counter.get(cnt1)  ..
```

```
           Self.register(<[] start <cnt1 t>>)
        :: p11 <cnt t>
        -> p12 <cnt t>;
    next With Self.start(<cnt1 t>)
        :: p12 <cnt t>
        -> p13 <cnt t>;
    End-new-TakeoffLocal(<cnt t>)
        :: p13 <cnt t>
        -> ;
        ;; Method run()
    run With Self.start-run(<cnt t>)
        :: id <[] run <cnt t>> -> ;
    start-run(<cnt t>)
        ::
        -> p21 <cnt t>, p29 <cnt t>;
        ;; the stop signal has been received,
        ;; then check if MSInt is empty
    next With Counter.get(cnt1)  ..
           MSInt.register(<[] isEmpty <cnt1 t>>)
        :: p21 <cnt t>
           End-reception true, MSInt MSInt
        -> p22 <cnt t>, MSInt MSInt;
        ;; MSInt is empty
    next With MSInt.isEmpty(true)
        :: p22 <cnt t>, MSInt MSInt
        -> p23 <cnt t>, MSInt MSInt;
        ;; loops until MSInt is empty
    next With MSInt.isEmpty(false)
        :: p22 <cnt t>, MSInt MSInt
        -> p21 <cnt t>, MSInt MSInt;

        ;; stop signal has been received and MSInt is empty
        ;; then send the stop signal to server and ...
    next With Counter.get(cnt1)  ..
              dataout.register(<stop-connection writeInt <cnt1 t>>)
        :: p23 <cnt t>, stop-connection stop-connection,
           dataoutputstream dataout
        -> p24 <cnt t>, stop-connection stop-connection,
           dataoutputstream dataout;
    next With dataout.writeInt(stop-connection)
        :: p24 <cnt t>, dataoutputstream dataout,
           stop-connection stop-connection
        -> p25 <cnt t>, dataoutputstream dataout,
           stop-connection stop-connection;
        ;; .. and flush dataout ...
     next With Counter.get(cnt1)  ..
           dataout.register(<[] flush <cnt1 t>>)
        :: p25 <cnt t>,
           dataoutputstream dataout
        -> p26 <cnt t>,
           dataoutputstream dataout;
    next With dataout.flush
        :: p26 <cnt t>, dataoutputstream dataout
        -> p27 <cnt t>, dataoutputstream dataout;

        ;; ... and stops itself
    next With Counter.get(cnt1)  ..
           Self.register(<[] stop <cnt1 t>>)
```

```
        :: p27 <cnt t>
        -> p28 <cnt t>;
    next With Self.stop
        :: p28  <cnt t>
        -> ;

        ;; MSInt has to be emptied
        ;; gets an integer from MSInt (random position)
    next With Random.get(random) ..
           Counter.get(cnt1)  ..
           MSInt.register(<random elementAt <cnt1 t>>)
        :: p29 <cnt t>, MSInt MSInt
        -> p210 <cnt t>
           MSInt MSInt, random <random <cnt t>>;
    next With MSInt.elementAt(random,i)
        :: p210 <cnt t>, MSInt MSInt, random <random <cnt t>>
        -> p211 <cnt t>, MSInt MSInt, random <random <cnt t>>,
           elem-to-send <i <cnt t>>;
    next With Counter.get(cnt1)  ..
           MSInt.register(<random removeElementAt <cnt1 t>>)
        :: p211 <cnt t>, MSInt MSInt, random <random <cnt t>>
        -> p212 <cnt t>, MSInt MSInt, random <random <cnt t>>;
    next With MSInt.removeElementAt(random)
        :: p212 <cnt t>, MSInt MSInt, random <random <cnt t>>
        -> p213 <cnt t>, MSInt MSInt;
        ;; sends integer to server and loops until MSInt is empty
    next With Counter.get(cnt1)  ..
           dataout.register(<i writeInt <cnt1 t>>)
        :: p213 <cnt t>, elem-to-send <i <cnt t>>,
           dataoutputstream dataout
        -> p214 <cnt t>, elem-to-send <i <cnt t>>,
           dataoutputstream dataout;
    next With dataout.writeInt(i)
        :: p214 <cnt t>, elem-to-send <i <cnt t>>,
          dataoutputstream dataout
        -> p29 <cnt t>, dataoutputstream dataout;
            ;; Method set-end-reception
    set-End-reception(value) With
           Self.start-set-End-reception(value, <cnt t>) ..
           Self.set-End-reception(<cnt t>)
        :: id <value set-End-reception <cnt t>> -> ;
    start-set-End-reception(value,<cnt t>)
        :: End-reception old-value
        -> p31 <cnt t>, End-reception value;
    End-set-End-reception(<cnt t>)
        :: p31 <cnt t> -> ;
    Where
      value, old-value  : boolean;
      stop-connection   : integer;
      dataout           : javadataoutputstream;
      MSInt             : javavector;
      textarea                : javatextarea;
      t                 : javathread;
      cnt, cnt1         : integer;
      random            : integer;
End TakeoffLocal;

;; TakeoffGlobal class
;;------------------------
```

```
Class TakeoffGlobal
Inherit JavaThreads;
Rename
  Thread -> TakeoffGlobal;
  javathread -> takeoffglobal;
Interface
  Use   JavaThreads, Integers, JavaDataInputStreams, JavaVectors,
        JavaTextAreas, TakeoffLocal;
  Methods
    run;
  Creation
    new-TakeoffGlobal _ _ _ _ _ : javadatainputstream javavector
                            javatextarea takeofflocal integer;
Body
  Use Booleans, Random, Clock, PairIntegerThreadIdentity,
      ThreadIdentity;
  Methods
    start-run     _ : threadidentity;
    start-new-TakeoffGlobal  _ _ _ _ _ _ : javadatainputstream
                javavector javatextarea takeofflocal
                integer threadidentity;
    End-new-TakeoffGlobal _ : threadidentity;
  Transitions
    tik;
  Places
        ;; Global Variables
    datainputstream   _ : javadatainputstream;
    MSInt             _ : javavector;
    textarea          _ : javatextarea;
    takeofflocal      _ : takeofflocal;
    stop-transmit     _ : integer;
    timeout           _ : integer;
        ;; Local Variables
    first, second,
    result            _ : pair-integerthreadidentity;

    p11 _ , p12 _ , p13 _ , p14 _ ,
    p21 _ , p22 _ , p23 _ , p24 _ , p25 _ , p26 _ , p27 _ , p28 _ ,
    p29 _ , p210 _ , p211 _ , p212 _ , p213 _ ,
    p214 _ , p215 _ : threadidentity;
  Axioms
        ;; Method new-TakeoffGlobal
    new-TakeoffGlobal(datain, MSInt, textarea,
           tl,stop-transmit) With
           TakeoffGlobal.getregister(
           <[datain,MSInt,textarea,tl,stop-transmit]
            new-TakeoffGlobal <cnt t>>) ..
          Self.start-new-TakeoffGlobal(datain, MSInt, textarea,
              tl,stop-transmit,<cnt t>) ..
          Self.End-new-TakeoffGlobal(<cnt t>)
        :: -> ;
    start-new-TakeoffGlobal(datain, MSInt, textarea, tl,
                stop-transmit, <cnt t>)
        ::
        -> p11 <cnt t>, datainputstream datain, MSInt MSInt,
           textarea textarea, takeofflocal tl,
           stop-transmit stop-transmit;
        ;; starts itself
    next With Counter.get(cnt1)  ..
```

```
           Self.register(<[] start <cnt1 t>>)
        :: p11 <cnt t>
        -> p12 <cnt t>;
    next With Self.start(<cnt1 t>)
        :: p12 <cnt t>
        -> p13 <cnt t>;
    End-new-TakeoffGlobal(<cnt t>)
        :: p13 <cnt t>
        -> ;
        ;; Method run()
    run With Self.start-run(<cnt t>)
        :: id <[] run <cnt t>> -> ;
    start-run(<cnt t>)
        ::
        -> p21 <cnt t>;

        ;; get the first integer
    next With Counter.get(cnt1)  ..
           datain.register(<[] readInt <cnt1 t>>)
        :: p21 <cnt t>, datainputstream datain
        -> p22 <cnt t>, datainputstream datain;
        ;; first integer is not a stop signal
    next With (datain.readInt(first) ..
               Random.get(millis) // C.clock(hour))
        :: (first = stop-transmit) = false
        => p22 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit
        -> p23 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit,
           first <first <cnt t>>, timeout (hour + millis);
        ;; first integer is a stop signal
    next With (datain.readInt(first)
        :: (first = stop-transmit) = true
        => p22 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit
        -> p210 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit,
           first <first <cnt t>>;
        ;; get the second integer
    next With Counter.get(cnt1)  ..
           datain.register(<[] readInt <cnt1 t>>)
        :: p23 <cnt t>, datainputstream datain, timeout d
        -> p24 <cnt t>, datainputstream datain;
        ;; second integer is not a stop signal
    next With datain.readInt(second)
        :: (second = stop-transmit) = false
        => p24 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit
        -> p25 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit,
           second <second <cnt t>>;
        ;; add first+second to MSInt
    next With Counter.get(cnt1)  ..
           MSInt.register(<first + second addElement <cnt1 t>>)
         :: p25 <cnt t>, MSInt MSInt,
           first <first <cnt t>>,
           second <second <cnt t>>
```

```
        -> p26 <cnt t>,  MSInt MSInt,
           first <first <cnt t>>,
           second <second <cnt t>>;
    next With MSInt.addElement(first + second)
        :: p26 <cnt t>, MSInt MSInt,
           first <first <cnt t>>,
           second <second <cnt t>>
        -> p27 <cnt t>, MSInt MSInt;
        ;; second integer is a stop signal
    next With datain.readInt(second)
        :: (second = stop-transmit) = true
        => p24 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit
        -> p28 <cnt t>, datainputstream datain,
           stop-transmit stop-transmit;
        ;; add only first integer to MSInt
    next With Counter.get(cnt1) ..
           MSInt.register(<first addElement <cnt1 t>>)
         :: p28 <cnt t>, MSInt MSInt,
           first <first <cnt t>>
        -> p29  <cnt t>, MSInt MSInt,
           first <first <cnt t>>;
    next With MSInt.addElement(first)
        :: p29  <cnt t>, MSInt MSInt,
           first <first <cnt t>>
        -> p210  <cnt t>, MSInt MSInt;
        ;; prevent deadlock when no sufficient integers.
        ;; tik adds only first to MSInt and loops for new integers.
    tik With C.clock(hour)
        :: (hour > d) = true
        => p23 <cnt t>, timeout d
        -> p214 <cnt t>;
        ;; adds only first to MSInt
    next With Counter.get(cnt1)  ..
           MSInt.register(<first addElement <cnt1 t>>)
         :: p214 <cnt t>, MSInt MSInt,
           first <first <cnt t>>
        -> p215  <cnt t>, MSInt MSInt,
           first <first <cnt t>>;
    next With MSInt.addElement(first)
        :: p215  <cnt t>, MSInt MSInt,
           first <first <cnt t>>
        -> p21  <cnt t>, MSInt MSInt;

        ;; a stop signal has been received, then
        ;; forward it to tl ...
    next With Counter.get(cnt1)  ..
           tl.register(<true set-End-reception <cnt1 t>>)
        :: p210 <cnt t>, takeofflocal tl
        -> p211 <cnt t>, takeofflocal tl;
    next With tl.set-End-reception(true)
        :: p211 <cnt t>, takeofflocal tl
        -> p212 <cnt t>, takeofflocal tl, ;

        ;;; ... and stops
    next With Counter.get(cnt1)  ..
           Self.register(<[] stop <cnt1 t>>)
        :: p212 <cnt t>
```

```
        -> p213 <cnt t>;
    next With Self.stop
        :: p213  <cnt t>
        -> ;
    Where
        datain          : javadatainputstream;
        MSInt           : javavector;
        textarea        : javatextarea;
        tl              : takeofflocal;
        t               : javathread;
        cnt1, cnt       : integer
        stop-transmit   : integer;
        first, second   : integer;
        hour, millis, d : integer;
End TakeoffGlobal;
```

B.5 CO-OPN/2 Specifications of Java Basics Classes

```
;; JVM Class
:: ------------
Class JVM;
Interface
  Use JavaStrings, JavaArrayStrings;
  Method
    java _ _ : javastring java-arraystring;
  Object JVM : jvm;
  Type jvm;
Body
  Use JavaObject,
      PairJavaObjectArrayString, Counter;
  Place
    Store _ : pair-javaobjectarraystring;
  Transition
    begin;
  Axioms
    java(ClassName,args)
        :: -> Store <ClassName args>;
    begin with Counter.get(cnt) ..
        ClassName.register(<args main <cntClassName>>) ..
        ClassName.main(args)
        :: Store <ClassName args> -> ;
    Where
      cnt : integer;
      args: java-arraystring
End JVM;

;; Java Object Class
;; ----------------------------------
Class JavaObject;
Interface
  Use Integers, RegisterParameters;
  Type javaobject;
  Methods
    wait, notify;
    register    _ : registerparameter;
```

```
    getregister _ : registerparameter;
  Object JavaObject: javaobject;
Body
  Use ThreadIdentity, BlackTockens, Counter,
      PairLockIdentity, PairThreadInteger;
  Methods
    start-notify    _ : threadidentity;
    end-notify      _ : threadidentity;
    start-wait      _ : threadidentity;
    end-wait        _ : threadidentity;
    lock _ , unlock _ : javathread;
  Transitions
    next;
  Places
        ;; Global Variables
        ;; set of threads waiting on the current object
    wait-set    _ : pairlockidentity;
        ;; set of threads resumed by a notify
    resumed-set _ : pairlockidentity;
        ;; the Thread who is currently possessing
        ;; the object's lock, together with
        ;; the number of current Integer locks it
        ;; possesses on the object.
    locker      _ : pairthreadinteger;
    locked      _ : blacktocken;

        ;; stores the method calls
    id          _ : registerparameter;
        ;; execution flow
    p11 _ , p12 _ , p13 _ ,
    p21 _ , p22 _ , p23 _ : threadidentity;
  Axioms
        ;; Method register: put call into id place
    register(regpar)
        :: -> id regpar;
        ;; Remove call from id (for dynamic creations only)
    getregister(regpar)
        :: id (regpar) -> ;

        ;; Method wait
     wait with self.start-wait(<cnt t>) ..
           self.end-wait(<cnt t>)
        :: id <[] wait <cnt t>>
        -> ;

    start-wait(<cnt t>)
        ::
        -> p11 <cnt t>;
        ;; it is necessary to have a lock on the
        ;; object in order to continue and
        ;; to release all the locks
    next
        :: p11 <cnt t>, locker <t i>
        -> p12 <cnt t>, locked @, wait-set <<t i> <cnt t>>;
        ;; reacquires all the locks on the object
    next with self.lock(t)
        :: p12 <cnt t>, resumed-set <<t j+1> <cnt t>>
        -> p12 <cnt t>, resumed-set <<t j><cnt t>>
    end-wait(<cnt t>)
```

```
        :: p12 <cnt t>, resumed-set <<t0><cnt t>>
        -> ;


        ;; Method notify
    notify with self.start-notify(<cnt t>) ..
           self.end-notify(<cnt t>)
        :: id <[] notify <cnt t>>
        -> ;
        ;;
    start-notify(<cnt1  t1>)
        ::
        -> p21 <cnt1 t1>;
        ;; it is necessary to have a lock on the
        ;; object in order to continue
    next
        :: p21 <cnt1 t1>, locker <t1 i>
        -> p22 <cnt1 t1>, locker <t1 i>
        ;; resume a thread that is in the wait-set
    next
        :: p22 <cnt1 t1>, wait-set <<t i> ,<cnt t>>
        -> p23 <cnt1 t1>, resumed-set <<t i> <cnt t>>
    end-notify(<cnt1 t1>)
        :: p23 <cnt1 t1>
        -> ;

        ;; Method lock
        ;; the current locker increments the lock
    lock(t)
        :: locker <t i>
        -> locker <t i+1>;
        ;; no current locker, acquisition of the lock
    lock(t)
        :: locked @;
        -> locker <t 1>;

        ;; Method unlock
        ;; the current locker decrements the lock
    unlock(t)
        :: locker <t i+1>
        -> locker <t i>;
        ;; the current locker releases the lock
    unlock(t)
        :: locker <t 1>
        -> locked @;

    Where
      t, t1     : javaobject;
      cnt1, cnt : integer;
      i         : integer;
      regpar    : registerparameter;
End JavaObject;

Class Counter;
Interface
  Use Integer;
  Type counter;
  Methods
     get _ : integer;
```

```
  Object Counter;
Body
  Places
    counters : integer;
  Initial
    counters  1;
  Axioms
    get(cnt) :: counters cnt -> counters succ (cnt);
    Where
      cnt : integer;
End Counter;

;; Java Thread class
;; ------------------------
Class JavaThreads;
Inherit JavaObject;
Rename
  JavaObject -> Thread;
  javaobject -> javathread;
Interface
  Use JavaObject;
  Subtype javathread -> javaobject;
  Methods
    run, start;
Body
  Use ThreadIdentity;
  Methods
    start-run     _ : threadidentity;
    start-start   _ : threadidentity;
    end-start     _ : threadidentity;

    p11 _ , p12 _ , p13 _ , p14 _ , p15 _ : threadidentity;
  Axioms
        ;; Method run
    run with start-run(<cnt t>)
        :: id <[] run <cnt t>>
        -> ;
        ;; empty (to be redefined by sub-classes)
    start-run(<cnt t>)
        :: -> ;

        ;; Method start
    start with start-start(<cnt t>) ..
           end-start(<cnt t>)
        :: id <[] start <cnt t>>
        -> ;
        ;; start is a synchronized method
    start-start(<cnt t>)
        ::
        -> p11 <cnt t>;
    next with self.lock(t)
        :: p11 <cnt t>
        -> p12 <cnt t>;
        ;; start  causes run
    next with Counter.get(cnt1) ..
           self.register(<[] run <cnt1 self>>)
        :: p12 <cnt t>
        -> p13 <cnt t>;
    next with self.run
```

```
        :: p13 <cnt t>
        -> p14 <cnt t>;
        ;; it is a new execution flow, thus there is no need
        ;; to wait for the end of the run method
    next with self.unlock(t)
        :: p14 <cnt t>
        -> p15 <cnt t>;
    end-start(<cnt t>)
        :: p15 <cnt t>
        -> ;
    Where
      t         : javathread;
      cnt, cnt1 : integer;
End JavaThreads;
```

B.6 Implementation: The Java Program

Here is the Java program described in Section 9.6.

Server Side

```
package RelayServer;
import java.io.*;
import java.net.*;
import java.util.*;

/**Create several socket connections with several clients.
  Act as a random relay between all the clients.
  Data sent along the socket must be of type int.
  */
public class RandomRelayServer extends Thread {
  /** default value for the server port is 6090 */
  public final static int DEFAULT_PORT=6090;
  public final static int STOP_TRANSMIT=-2;
  public final static int STOP_CONNECTION=-1;

  int port;

  ServerSocket listen_socket;
  GlobalRelay globalrelay;

  /** Create a ServerSocket to listen for connections on a given port.
    Initialize the thread GlobalRelay which will realize the random relay
    between all clients. <br>
    Starts itself.
    */
  public RandomRelayServer(int port){
    if (port == 0) port=DEFAULT_PORT;
    this.port = port;
    try { listen_socket = new ServerSocket(port); }
    catch(IOException e) {
      System.out.println("Exception creating server socket"+e);
    }
    System.out.println("RandomRelayServer: listening on port "+port);
    globalrelay = new GlobalRelay();
    this.start();
  }
```

```

  /**Body of the server thread. Loop forever, listening for and
    accepting connections from clients. For each connection, initialize two threads
    InputRelay, and OutputRelay, handling respectively incoming and outgoing
    communication from/to clients.
    */
  public void run(){
     try{
       while(true){
         Socket client_socket = listen_socket.accept();
         System.out.println("A client wants a connection\n");
         OutputRelay outputrelay = new OutputRelay(client_socket,globalrelay,
                                                   STOP_TRANSMIT);
         InputRelay inputrelay = new InputRelay(client_socket, globalrelay,
                                                outputrelay, STOP_TRANSMIT,
                                            STOP_CONNECTION);

       }
     }
     catch(IOException e) {
       System.out.println("Exception while listening for connections "+e);
     }
   }

  /**Start the server up, listening on an optionally specified port. <br>
    Default port is 6090.
   */
  public static void main(String[] args){
    int port =0;
    if (args.length == 1){
      try {port = Integer.parseInt(args[0]);}
      catch (NumberFormatException e) {port = 0;}
    }
    new RandomRelayServer(port);
  }

} //end of RandomRelayServer class

//-------------------

/** Handle all incoming communication from a dedicated client using Socket.
  Relay this data to GlobalRelay. Notifies OutputRelay if the stop_transmit signal
  is received from client. Stops itself it the stop_connection signal is received
  from client.
  */
class InputRelay  extends Thread{
  Socket client;
  GlobalRelay globalrelay;
  DataInputStream in;
  OutputRelay outputrelay;
  int stop_transmit;
  int stop_connection;


  /** Initialize DataInputStream and starts itself
   */
  public InputRelay(Socket client_socket, GlobalRelay globalrelay,
                    OutputRelay outputrelay, int stop_transmit,
                    int stop_connection){
    this.client = client_socket;
    this.globalrelay = globalrelay;
    this.outputrelay = outputrelay;
    this.stop_transmit = stop_transmit;
    this.stop_connection = stop_connection;

    try{in = new DataInputStream(client_socket.getInputStream());}
    catch(IOException e){
      try {client.close();}
```

```
      catch(IOException e2){
        System.out.println("Exception while getting socket streams:"+e2);};
      System.out.println("Exception while getting socket streams:"+e);
      return; //to RandomRelayServer
    }
    this.start();
  }


  /**Body of InputRelay.<br>
    Read data from DataInputStream of client.
    Put data to GlobalRelay
    */
  public void run(){
    int elem;

    try{
      for(;;){
        // Read a data from DataInputStream of client
        try{
          elem = in.readInt();
          if (elem == stop_connection) {
            //client has no more to send
            System.out.println("InputRelay "+this.getName()+
                               " Exit DSGamma done.\n");
              break; // to finally
          }
          if (elem == stop_transmit) {
            //client wants no more on its input (our output)
            System.out.println("InputRelay "+this.getName()+
                               " Exit DSGamma: stop sending
                                 received from client\n");
            outputrelay.setnotify_end_sending(true);
          }
          else {
            // Relay data to GlobalRelay thread.
            System.out.println("InputRelay before put"+this.getName()+" "+elem);
            globalrelay.put(elem);
            System.out.println("InputRelay after put"+this.getName()+" "+elem);
          }
        }
        catch(IOException e) {
          System.out.println("Input Relay "+this.getName()+
                             " not possible: "+e+"\n");

          break; // to finally
        }
      }
    }
    finally {
      try {client.close();client = null; }
      catch(IOException e2) {
        System.out.println("Exception closing client: "+e2+"\n");
      }
      finally{stop();}
    }
  }

} // end of InputRelay

//-------------------

/** Handle all outgoing communication to a dedicated client.
  Relay a data from GlobalRelay thread to the dedicated client.
  */
class OutputRelay  extends Thread{
  Socket client;
  GlobalRelay globalrelay;
```

```
DataOutputStream out;
int stop_transmit;

boolean end_sending= false;

/** Initialize DataOutputStream and starts itself
 */
public OutputRelay(Socket client_socket, GlobalRelay globalrelay,
                   int stop_transmit){

  this.client = client_socket;
  this.globalrelay = globalrelay;
  this.stop_transmit = stop_transmit;

  try{out = new DataOutputStream(client_socket.getOutputStream());}
  catch(IOException e){
    try {client.close();}
    catch(IOException e2){
      System.out.println("Exception while getting socket streams:"+e2);
    }
    System.out.println("Exception while getting socket streams:"+e);
    return; //to RandomRelayServer
  }
  this.start();
}

/** Body of OutputRelay.<br>
  Get data from GlobalRelay <br>
  Relay data to DataOutputStream of client.
  */
public void run(){
  int elem;

  try{
    for(;;){
      if (end_sending){
          System.out.println("OutputRelay "+this.getName()+
                             " Exit DSGamma: stop sending received\n");
          //notifies the client that the stop_transmit signal has been received
          try{
            out.writeInt(stop_transmit);
            out.flush();
          }
          catch(IOException e) {
            System.out.println("OutputRelay "+this.getName()+" not possible\n"+e);
          }
          finally{break;} //to stop
      }


      // Wait for  data from GlobalRelay
      System.out.println("OutputRelay before get"+this.getName());
      elem = globalrelay.get();
      System.out.println("OutputRelay after get"+this.getName()+" "+elem);

      // Relay data to DataOutputStream of client.
      try{
        out.writeInt(elem);
        out.flush();
        System.out.println("OutputRelay "+this.getName()+" "+elem);
      }
      catch(IOException e) {
        System.out.println("OutputRelay "+this.getName()+" not possible\n"+e);

        //Save value
        globalrelay.put(elem);
        break;  // to finally
      }
```

```
      }
    }
    finally{
      System.out.println("Output relay "+this.getName()+
                         " Exit DSGamma: stop sending done\n");
      stop();
    }

  }

  /** Set end_sending to value <br>
    It is used as an asynchronous flag to notify OutputRelay to stop sending
    data to a client.
   */
  public  void setnotify_end_sending(boolean value){
    end_sending = value;
  }
} // end of OutputRelay


//-------------------

/** Act as a FIFO buffer.
  */
class GlobalRelay extends Thread{
  Vector buffer;

  /** Initializes the FIFO buffer to empty and Starts itself */
  public GlobalRelay(){
    buffer = new Vector();
    this.start();
  }

  /**Incoming data  is stored at the end of the FIFO buffer
   */
  synchronized public void put(int input_elem){

    //prevent two consecutive put, without intermediary get
    System.out.println("GlobalRelay rcvd "+input_elem);
    buffer.addElement(new Integer(input_elem));
    notify();
  }

  /**First data stored in buffer is returned and removed from the FIFO buffer.
    This method blocks  until a data to relay is available.
    */
  synchronized public int get(){
    int elem_to_relay;

    while (buffer.isEmpty()) {
      try {wait();}
      catch (InterruptedException e) {
        System.out.println("Error while get GlobalRelay is waiting "+e);
      }
    }
    elem_to_relay = ((Integer) buffer.elementAt(0)).intValue();
    System.out.println("GlobalRelay has relayed "+elem_to_relay);
    buffer.removeElementAt(0);
    return elem_to_relay;
  }
} //end of GlobalRelay
```

Client Side

```
package Gamma;
```

```java

import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import MyUtils.*;

/*
   Distributed Gamma-like addition of integers
   */

/** Distributed Gamma-like addition of integers
  DSGammaClientApp Applets allows a user to enter several integers.
  This local multiset (Vector MSInt) of integers will be part of a global distributed
  multiset of integers, that obtained by the union of all the other local multisets of
  integers provided by all the other users using the same applet.
  DSGammaClientApp is responsible for: <br>
  a) establishing connection with a server,  <br>
  b) entering the DSGamma system (the set of all these applets running),  <br>
  c) managing integers entered by user and those received by the server,  <br>
  d) properly quitting the DSGamma system (empty the local
  MSInt of integers, stop the threads and closing socket)
  */
public class DSGammaClientApp extends Applet{
  public final static int PORT=6090;
  public final static int STOP_TRANSMIT=-2;
  public final static int STOP_CONNECTION=-1;

  Socket s;
  DataInputStream in;
  DataOutputStream out;
  TextField textfield;
  TextArea textarea;
  Button stop_button;
  Button result_button;

  TakeoffGlobal takeoffglobal;
  TakeoffLocal takeofflocal;

  Vector MSInt;

  /** Create a socket to communicate with a server on port 6090
    of the host that the applet's code is on. Create streams to use
    with the socket. Then create a TextField for user input, a TextArea
    for output, and a Button for exitting the DSGamma system.
    MSInt stores the integers entered by the local users, and those received by the
    server. Finally, create two threads for interaction with
    the server.
    */
  public void init(){

    try{
      s=new Socket(this.getCodeBase().getHost(),PORT);
      in=new DataInputStream(s.getInputStream());
      out=new DataOutputStream(s.getOutputStream());

      textfield = new TextField();
      textarea = new TextArea();
      stop_button = new Button("Exit DSGamma System");
      result_button = new Button("Result");
      textarea.setEditable(false);
      MSInt = new Vector();


      setLayout(new BorderLayout());
      add("North",textfield);
      add("Center",textarea);
```

```
      add("South",stop_button);
      add("East",result_button);

      //Initializes takeofflocal and takeoffglobal threads
      takeofflocal = new TakeoffLocal(out, MSInt, textarea, STOP_CONNECTION);
      takeoffglobal = new TakeoffGlobal(in, MSInt, textarea,takeofflocal,
                                         STOP_TRANSMIT);

      showStatus("Connected to "
                      + s.getInetAddress().getHostName()
                      + ":" + s.getPort()+"\n");
    }
    catch (IOException e) {
      showStatus("Exception while creating socket: "+e);
      try{if (s!=null) {s.close();}}
      catch (IOException e2) {
        showStatus("Exception while closing socket: "+e2);
      }
      stop();
    }

  }
   /** Close the socket and the input, output streams
   */
  public void stop(){

    try{
      if (in!=null)  {in.close(); in = null;}
      if (out!=null) {out.close(); out = null;}
      if (s!=null) {s.close(); s = null;}
    }
    catch (IOException e2) {
     showStatus("Exception while closing socket: "+e2);
    }

    showStatus("ByeBye\n");
  }

  /** Capture events on the TextField or Button Components of the interface
   */
  public boolean action(Event event, Object what){

    //User types a line in textfield, convert it to a Vector of Integer
    if (event.target == textfield){
      // Convert String into Vector of Integers (MSInt)
      Convert.StringtoInteger(textfield.getText(),MSInt);
      textfield.setText("");
      showStatus("User entered some integers\n");

      //Notifies takeofflocal, because MSInt is no more empty
      synchronized (takeofflocal) {takeofflocal.notify();}

      return true;
    }


    //User wants to exit the DSGamma system
    if (event.target == stop_button){
      //Notifies the server that the user wants to stop
      try{
        out.writeInt(STOP_TRANSMIT);
        textarea.appendText("Exit DSGamma requested\n");
      }
      catch(IOException e) {
        System.out.println("Client can't write on socket: "+e);
      }
      return true;
    }
```

```

    //User wants to see a result
    if (event.target == result_button){
       textarea.appendText("Result:"+ MSInt.elementAt(0).toString());
      return true;
    }
    return false;
  }
} // end of DSGammaClientApp


// ------------------------------------
/**  Randomly removes one integer from local multiset (Vector MSInt) of integers.
 */
class TakeoffLocal extends Thread{
  DataOutputStream out;
  Vector MSInt;
  TextArea textarea;
  int stop_connection;

  boolean end_reception = false;

  public TakeoffLocal(DataOutputStream out, Vector MSInt, TextArea textarea,
                      int stop_connection){
    this.out = out;
    this.MSInt = MSInt;
    this.textarea = textarea;
    this.stop_connection = stop_connection;
    this.start();
  }

  /**Body of TakeoffLocal.
    Wait for MSInt to be not empty, the send the content of MSInt to server.
    If no more integers will be received from server, MSInt is emptied a last time,
    before stopping.
    */
  public synchronized void run(){
    for (;;) {

      //Check if  TakeoffGlobal has finished received integers (end_reception = true).
      //In this case no more integers will be added in MSInt, and TakeoffLocal empties
      //MSInt a last time and stops.
      if (end_reception) {
        textarea.appendText("Emptying local multiset for the last time\n");

        //free MSInt
        doReactions();

        //send stop_connection signal to server
        try{
          out.writeInt (stop_connection);
          out.flush();
        }
        catch(IOException e) {
        System.out.println("Client can't write on socket: "+e);
        }
        //TakeoffLocal can stop
        finally{break;} //to stop()
      }


      //TakeoffGlobal is still receiving integers from server.
      //TakeoffLocal waits for user or for TakeoffGlobal to enter integer numbers
      //i.e. wait for MSInt to be not empty.
      try{wait();}
      catch(InterruptedException e) {
        textarea.appendText("Exception while waiting: "+e);
      }
```

```
      finally{
        //Free MSInt
        doReactions();
        }
    }

    textarea.appendText("Exit DSGamma system done\n");
    stop();
  }

  /**Randomly chooses one integer in Vector MSInt, and sends it to the Server,
    till MSInt is not empty.
    */
  public  void  doReactions(){
    int i;

    while (!MSInt.isEmpty()){

      //Show the user the new state of Vector
      textarea.appendText("\n");
      for (i=0; i<MSInt.size();i++){
        textarea.appendText(MSInt.elementAt(i).toString()+" ");
      }
      textarea.appendText("\n");

      //Choose an index
      i = (int) (Math.random() * MSInt.size()) % MSInt.size() ;

      //Send the chosen integer to the server
      try{
        out.writeInt(((Integer) MSInt.elementAt(i)).intValue());
        //Remove the integer from Vector MSInt
        MSInt.removeElementAt(i);
      }
      catch (IOException e) {
        System.out.println("Client can't write on socket: "+e);
        stop();
      }
    }
    //Ensure the sending of integers to server
    try{
      out.flush();
      textarea.appendText("\nEmpty\n");
    }
    catch(IOException e) {
        System.out.println("Client can't write on socket: "+e);
        stop();}

  }

  //TakeoffGlobal set end_reception to true when it has finished receiving
  //integers from server.
  /**Set variable end_reception to value. <br>
    It is used as an asyncronous flag to notify TakeoffLocal that nothing
    more will be received from server.
   */
  public  void set_end_reception(boolean value){
    end_reception = value;
  }


}// end of TakeoffLocal


// -------------------------------------
/** Wait for output (2 integers) from the server on the specified stream, adds
  them and puts the result in its local Vector of integers.
  */
```

```
class TakeoffGlobal extends Thread{
  DataInputStream in;
  TextArea textarea;
  Vector MSInt;
  TakeoffLocal takeofflocal;
  int stop_transmit;

  int result;


  public TakeoffGlobal(DataInputStream in, Vector MSInt, TextArea textarea,
                       TakeoffLocal takeofflocal, int stop_transmit){
    this.in = in;
    this.textarea = textarea;
    this.MSInt = MSInt;
    this.takeofflocal = takeofflocal;
    this.stop_transmit = stop_transmit;
    this.start();
  }

  //Body of TakeoffGlobal.
  public synchronized void run(){
    doReactions();
    takeofflocal.set_end_reception(true);
    synchronized (takeofflocal) {takeofflocal.notify();}
    textarea.appendText("Exit DSGamma: stop receiving integers\n");
    stop();
  }


  /** Read two integers from server and add their sum to MSInt
    If the second integer does not come sufficiently soon, the first one
    is added to MSInt. This is useful when the number of integers in the global multiset
    is less than the number of current users.
    If the second integer is the STOP_TRANSMIT signal, then TakeoffGlobal adds
    the first one to MSInt and then stops.
    If the first integer is the STOP_TRANSMIT signal, then doReactions
    returns immediately.
    */
  public void doReactions(){
    int tmp = 0;
    int i;

    //Wait for two integer, add their sum to MSInt
    //until the stop signal arrives
    while(tmp != stop_transmit) {
      try{
        result = in.readInt();
        if (result == stop_transmit) {
          //the first integer is the stop signal, it is time to return
          break; //to return
        }
        if (in.available() > 0) {
          //A second integer is available, check for the stop signal and
          //add them if necessary
          tmp = in.readInt();
          if (tmp != stop_transmit) {
            result +=tmp;
          }
        }
        // A second integer is not available immediately
        else {
          // Sleep a random amount of millis before checking a second time
          // for available data from server.
          i = (int) (Math.random() *2000);
          try{sleep(i);}
          catch(InterruptedException e) {
            textarea.appendText("Exception while sleeping: "+e);return;
```

```
        }
        finally{
          if (in.available() > 0) {
            //A second integer is available after sleeping, check of the stop signal
            //and add the first and the second if necessary
            tmp = in.readInt();
            if (tmp!= stop_transmit) {
              result +=tmp;
            }
          }
           // if no second integer is availabe, the first one is reinjected
           // in Vector, instead of the sum.
        }
      }

      // Add either the first integer received from server, or the sum of
      // two integers received from server.
      MSInt.addElement(new Integer(result));

      // Notifies TakeoffLocal that a new integer has arrived in MSInt, this is
      // necessary if MSInt was empty.
      synchronized (takeofflocal) {takeofflocal.notify();}

     }
    catch(IOException e) {
      textarea.appendText("Connection closed by server");
      break; //to return
    }
  }
  return; //to run()
 }
}// end of TakeoffGlobal
```

Utils

```
package MyUtils;

import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;

/** Set of functions useful for some conversions.
 */
public final class Convert{

  /** Converts a String into a Vector of Integers.
    Ex: String (12 34) becomes Vector of two Integers 12 and 34.
    The current implementation does not consider ill-formed strings.
    */
  public static void StringtoInteger(String s, Vector v){

      int beginIndex =0;
      int endIndex;

      // extraction of substring from a string
   BI: while(beginIndex < s.length()){

        //search for a new integer
        while(Character.isSpace(s.charAt(beginIndex))) {
          beginIndex++;
          if (beginIndex == s.length()) break BI;
        }
        endIndex = beginIndex+1;
        if (endIndex < s.length()) {
```

```
          while(!Character.isSpace(s.charAt(endIndex))) {
            endIndex++;
            if (endIndex == s.length()) break;
          }
        }

        // add the new integer to the Vector
        v.addElement(Integer.valueOf(s.substring(beginIndex,endIndex)));
        beginIndex = endIndex;

   } // end of BI
  }// end of StringtoInteger
}
```

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, DEC Research Center, Palo Alto, CA, 1988.

[2] M. Abadi and L. Lamport. Open systems in TLA. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–90, 1994.

[3] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[4] J.-R. Abrial. A refinement case study (using the Abstract Machine Notation). In R.C. Morris and J.M. Shaw, editors, *4th Refinement Workshop. Proceedings of the 4th Refinement Workshop*, Workshops in Computing, pages 51–96, Berlin, Germany, jan 1991. Springer-Verlag.

[5] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[7] R.J.R. Back. Refinement Calculus, Part II: Parallel and Reactive Programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness - REX Workshop*, volume 430 of *LNCS*, Berlin, Germany, 1989. Springer-Verlag.

[8] R.J.R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness - REX Workshop*, volume 430 of *LNCS*, Berlin, Germany, 1989. Springer-Verlag.

[9] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR'94: concurrency theory*, volume 896 of *LNCS*, Berlin, Germany, 1994. Springer-Verlag.

[10] J.-P. Banâtre and D. Métayer. Gamma and the chemical reaction model. In IC Press, editor, *Proceedings of the Coordination'95 workshop*, 1995.

[11] S. Barbey. *Test Selection for Specification-Based Unit Testing of Object-Oriented Software Based on Formal Specifications.* PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1997.

[12] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy, October 1996*, LNCS (Lecture Notes in Computer Science) 1150, pages 303–320. Springer-Verlag, 1996. Also available as Technical Report (EPFL-DI No 96/163), Published in DeVa first year report (December 96).

[13] E. Best and T. Thielke. Refinement of coloured Petri nets. In B. S. Hlebus and L. Czaja, editors, *Fundamentals of computation theory: FCT'97*, volume 1279 of *Lecture Notes in Computer Science*, pages 105–116, Berlin, Germany, 1997. Springer-Verlag.

[14] O. Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva, Geneva, 1997. No 2919.

[15] O. Biberstein, D. Buchs, C. Buffard, M. Buffo, J. Flumet, J. Hulaas, G. Di Marzo, and P. Racloz. SANDS1.5/COOPN1.5, An overview of the language and its supporting tools. Tech. Report 95/133, Swiss Federal Institute of Technology (EPFL), Software Engineering Laboratory, Lausanne, Switzerland, June 1995.

[16] P. Borba. *Semantics and Refinement for a Concurrent Object Oriented Language*. PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, July 1995.

[17] P. Borba and J. Goguen. An operational semantics for FOOPS. In R. Wieringa and R. Feenstra, editors, *Working Papers of the International Workshop on Information Systems - Correctness and Reusabilility, IS-CORE'94. Technical Report IR-357*, Amsterdam, 1994. Vrije Universiteit.

[18] P. Borba and J. Goguen. Refinement of concurrent object oriented programs. Technical Report PRG-TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, 1995.

[19] W. Brauer, R. Gold, and W. Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *LNCS*, pages 1–46, Berlin, Germany, 1990. Springer-Verlag.

[20] D. Buchs, P. Racloz, M. Buffo, J. Flumet, and E. Urland. Deriving parallel programs using sands tools. *Transputer Communications*, 3(1):23–32, January 1996.

[21] Didier Buchs and Nicolas Guelfi. CO-OPN: A concurrent object oriented Petri nets model. Rapports de Recherche, LRI 616, University of Paris-Sud, 1990.

[22] M. Buffo. *Contextual Coordination: a coordination model for distributed object systems.* PhD thesis, University of Geneva, 1997.

[23] M. Buffo and D. Buchs. A distributed semantics for a design-level IWIM-based coordination language. In *submitted to COORDINATION'99*, 1999.

[24] M. Buffo and D. Buchs. Polymorphism and module-reuse mechanisms for algebraic Petri nets in CoopnTools. In *submitted to ICATPN'99*, 1999.

[25] A. Coen-Porisini, R. A. Kemmerer, and D. Mandrioli. A formal framework for ASTRAL inter-level proof obligations. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, number 989 in LNCS, pages 90–108, Berlin, Germany, 1995. Springer-Verlag.

[26] G. Denker. Reification - changing viewpoint but preserving truth. In M. Haveraan, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Types Specification, Proc. 11th Workshop on Specification of Abstract Data Types joint with the 8th General COMPASS Meeting. Selected Papers.*, volume 1130 of *LNCS*, pages 182–199, Berlin, Germany, 1996. Springer-Verlag.

[27] G. Denker. Semantic refinement of concurrent object systems based on serializability. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 105–126. Kluwer Academic Publisher, 1996.

[28] G. Denker and P. Hartel. TROLL - an object oriented formal method for distributed information system design: Syntax and pragmatics. Technical Report 97-03, Technische Universität Braunschweig, Braunschweig, Germany, 1997.

[29] R. Devillers, H. Klaudel, and R.-C. Riemann. General refinement for high level Petri nets. In S. Ramesh and G. Sivakumar, editors, *Foundations of software technology and theoretical computer science: proceedings*, volume 1346 of *Lecture Notes in Computer Science*, pages 297–311, Berlin, Germany, 1997. Springer-Verlag.

[30] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. F. Zorzo. Co-opn/2 specifications of the DSGamma system designed using coordinated atomic actions. Technical Report 641, University of Newcastle upon Tyne, june 1998.

[31] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. F. Zorzo. Formal development and validation of the DSGamma system based on CO-OPN/2 and Coordinated Atomic Actions. Technical Report 98/265, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[32] Giovanna Di Marzo Serugendo and Nicolas Guelfi. Formal Development of Java Programs. Technical Report 97/248, Software Engineering Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1997.

[33] M. Felder, A. Gargantini, and A. Morzenti. A theory of implementation and refinement in timed Petri nets. *Theoretical Computer Science*, 202(1–2):127–16, 1998.

[34] M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real-time systems through logical specifications and Petri net models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994.

[35] J. L. Fiadeiro. On the emergence of properties in component-based systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology. Proceedings - 1996*, volume 1101 of *LNCS*, pages 421–443, Berlin, Germany, 1996. Springer-Verlag.

[36] David Flanagan et al. *Java in a Nutshell.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, deluxe edition, 1997.

[37] C. Ghezzi and R. Kemmerer. ASTRAL: An assertion language for specifying realtime systems. In A. van Lamsweerde and A. Fugetta, editors, *Proceedings of the European Software Engineering Conference (ESEC'91)*, number 550 in LNCS, pages 122–146, Berlin, Germany, 1991. Springer-Verlag.

[38] C. Ghezzi and R. Kemmerer. Executing formal specifications: the ASTRAL to TRIO translation approach. In *Proceedings of TAV4: the Symposium on Testing, Analysis, and Verification*, pages 112–119, New-York, 1991. ACM Software Engineering Notes.

[39] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[40] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, Menlo Park, CA, 1988.

[41] A. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[42] M. Huhn, H. Wehrheim, and G. Denker. Action refinement in system specification: Comparing a process algebraic and an object-oriented approach. In U. Herzog and H. Hermanns, editors, *GI/ITG-Fachgespräch: Formale Beschreibungstechniken für verteilte Systeme, 29/9 in Arbeitsbericht des IMMD*, pages 77–88, Germany, 1996. Universität Erlangen. http://www.cs.tu-bs.de/idb/publications/huhwehden96_abs.html.

[43] J. Hulaas. *An Incremental Prototyping Methodology for Distributed Systems Based on Formal Specifications.* PhD thesis, no 1664, Swiss Federal Institute of Technology in Lausanne, 1997.

[44] J. Jacob. The varieties of refinements. In J.M. Morris and R.C. Shaw, editors, *4th Refinement Workshop*, Workshops in Computing, pages 441–455, Berlin, Germany, 1991. Springer-Verlag.

[45] A. Kiehn. Petri net systems and their closure properties. In G. Rozenberg, editor, *Advances in Petri Nets 1989*, volume 424 of *LNCS*, pages 306–328, Berlin, Germany, June 1989. Springer-Verlag.

[46] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[47] K. Lano. *Formal Object-Oriented Development*. Springer-Verlag, London, 1995.

[48] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley, 1997.

[49] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[50] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[51] J. Padberg. *Abstract Petri Nets: Uniform Approach and Rule-Based Refinement*. PhD thesis, Technical University of Berlin, 1996.

[52] C. Péraire. *Formal Testing of Object-Oriented Software: from the Method to the Tool*. PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1998.

[53] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[54] A. Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, number 652 in LNCS, pages 1–38, Berlin, Germany, 1992. Springer-Verlag.

[55] L. Pomello, G. Rozenberg, and C. Simone. A survey of equivalence notions for net based systems. In G. Rozenberg, editor, *Advances in Petri nets*, volume 609 of *LNCS*, pages 410–472, Berlin, 1992. Springer-Verlag.

[56] Wolfgang Reisig. Petri nets and algebraic specifications. In *Theoretical Computer Science*, volume 80, pages 1–34. Elsevier, 1991.

[57] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25(3):233–281, 1988.

[58] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, 1992.

[59] J. Vachon and D. Buchs. Towards a complete semantics with negation rules for CO-OPN/2. Technical Report 98/297, Swiss Federal Institute of Technology in Lausanne, Switzerland, 1998.

[60] W. Vogler. Failures semantics of Petri nets and the refinement of places and transitions. Technical Report TUM-I9003, Institut für Informatik, Universität München, 1990.

[61] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

[62] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing, IEEE CS Press, Pasadena, USA*, pages 450–457, 1995.

Curriculum Vitae

Giovanna A. Di Marzo Serugendo

Education	
1993	M.sc. Mathematics - University of Geneva Domain: Numerical Analysis
1994	M.sc. Computer Science - University of Geneva Domain: Software Engineering
Research	
1993 - 1996	Distributed Systems: Dynamically reconfigurable entities for building protocols; Mobile agents for implementing distributed systems; Specifications of systems built with mobile agents;
1994 - 1999	Software Engineering: Category theory; Stepwise refinement of formal specifications; Specifications of multi-threaded transactions;
Teaching	
1993 - 1998	Seminars on practical aspects of computer networks; Exercises on computer architecture; Tutorial on advanced calculus;
Other Activities	
1993 - 1996	Contribution to the definition and submission of several research projects related to agent technology;
1997 - 1998	Reviewer (ICATPN, INET, European Projects);
1999	Member of Program Committee of International Workshops (AA'99).