VRIJE UNIVERSITEIT

Modal Abstraction and Replication of Processes with Data

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 5 december 2005 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Miguel Ángel Valero Espada

geboren te Barcelona

promotor: prof.dr. W.J. Fokkink
copromotor: dr. J.C. van de Pol

Modal Abstraction and Replication of Processes with Data

Miguel Valero

Printed by Ponsen & Looijen
ISBN 90-64641-15-3
IPA dissertation series 2005-20

The research in this thesis has been carried out at the Centre for Mathematics and Computer Science (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). It has been financially supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research (NWO), the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009 "Formal Design, Tooling, and Prototype Implementation of a Real-Time Distributed Shared Dataspace".

"Arithmétique! algèbre! géométrie! trinité grandiose! triangle lumineux! Celui qui ne vous a pas connues est un insensé! Il mériterait l'épreuve des plus grands supplices; car, il y a du mépris aveugle dans son insouciance ignorante; mais, celui qui vous connaît et vous apprécie ne veut plus rien des biens de la terre; se contente de vos jouissances magiques; et, porté sur vos ailes sombres, ne désire plus que de s'élever, d'un vol léger, en construisant une hélice ascendante, vers la voûte sphérique des cieux. La terre ne lui montre que des illusions et des fantasmagories morales; mais vous, ô mathématiques concises, par l'enchaînement rigoureux de vos propositions tenaces et la constance de vos lois de fer, vous faites luire, aux yeux éblouis, un reflet puissant de cette vérité suprême dont on remarque l'empreinte dans l'ordre de l'univers. Mais, l'ordre qui vous entoure, représenté surtout par la régularité parfaite du carré, l'ami de Pythagore, est encore plus grand; car, le Tout-Puissant s'est révélé complètement, lui et ses attributs, dans ce travail mémorable qui consista à faire sortir, des entrailles du chaos, vos trésors de théorèmes et vos magnifiques splendeurs. Aux époques antiques et dans les temps modernes, plus d'une grande imagination humaine vit son génie, épouvanté, à la contemplation de vos figures symboliques tracées sur le papier brûlant, comme autant de signes mystérieux, vivants d'une haleine latente, que ne comprend pas le vulgaire profane et qui n'étaient que la révélation éclatante d'axiomes et d'hyéroglyphes éternels, qui ont existé avant l'univers et qui se maintiendront après lui. Elle se demande, penchée vers le précipice d'un point d'interrogation fatal, comment se fait-il que les mathématiques contiennent tant d'imposante grandeur et tant de vérité incontestable, tandis que, si elle les compare à l'homme, elle ne trouve en ce dernier que faux orgueil et mensonge."

Comte de Lautréamont
Les Chants de Maldoror, 1869

Acknowledgements

After more than four years this thesis is finally getting to the end. Now, I am writing what I feel is the most critical part of it: the acknowledgements. The problem is that I do not know how to express all my gratitude to the people that contributed in different ways to support my work and my every day life during this time.

No doubt, first I must begin being thankful to Jaco van de Pol. Without your daily supervision, your help, your comments, your critics and your inexhaustible enthusiasm and energy, I would not have been able to afford the completion of my work.

I cannot think of a better promotor than Wan Fokkink. He was always present during the process of my thesis, especially active at the end. Wan and Jaco gave me the opportunity to start my research career at CWI. They showed their confidence in my work from the very beginning (even before) until the end. Wan was the leader of the SEN2 group when I joined it in 2001, now Jaco has got his position. I have learned so much from their leadership, that I can just express my most sincere gratitude.

Apart from Jaco and Wan, I would also like to thank all the other members (and ex-members) of the SEN2 group. It was a great pleasure to share the working hours with you. I really appreciate your friendly faces and your clever minds.

I want to express my gratitude to the members of my reading committee. Arend Rensink, Jan Friso Groote, Jan Willem Klop and María del Mar Gallardo. Their careful reading of my thesis and their comments helped to improve the final result of my work. I thank also Jozef Hooman who kindly agreed to act as opponent in the defense of my thesis.

Special thanks go to the members of the user committee of the PROGRESS project CES. 5009 in which I worked these last four years. They closely followed the development of my work and contributed to it with many valuable ideas and comments.

Thanks to María del Mar and Pedro M., I could spend a couple of months on a research visit at the University of Málaga. The time was short but fruitful. Also, I thank Gabriel, Pedro d'A. and Nico. They brought me to the University of Córdoba, Argentina, giving me the possibility of teaching and of finishing my thesis in an unsurpassable environment.

Mis padres y mi hermano estuvieron siempre a mi lado cuando los necesite y cuando no, por eso va hacia ellos mi más sincero agradecimiento.

Personally, I am in debt with many more other people. These last years I have passed most part of the time in Amsterdam, where I changed my accommodation quite a lot of times. I have traveled a few times to many different countries: Belgium, France, Italy, U.K., Portugal, Germany, Spain, Austria, China,... There was a pleasant period in the warm Málaga. And nowadays I am living in the heart of Argentina. During this time, I have met a lot of people in different places and situations. With some of them I have just shared a glass of wine, a smile, a walk under the moon, a technical chat or a philosophical discussion, with others a have shared some of the most important parts of my life. I would like to give a list with all the names expressing how much I liked to meet each one, but I panic against the possibility of forgetting a single name, or not being able to find the adequate words to express my gratitude. Actually, I am not going to give it a try... I just want to conclude by being thankful to everyone that has been present in my mind and my heart.

Miguel
Buenos Aires, 2005

Contents

Introduction

From Practice to Theory... and Back

During the last decade many lines have been written trying to motivate the use of formal techniques in the industrial world (see, for example, [27, 111]). Despite the fact that there is now a common agreement about the benefits of formal methods for critical systems development in certain communities, we are still in a preliminary phase towards the full integration of formal techniques in the development process. Usually, real-life systems are far more complex than the case studies analysed by the scientific community. The challenge of formal methods is to make the theoretical results applicable to large scale systems.

This thesis is the result of the work done in a project with the title *Formal Design, Tooling, and Prototype Implementation of a Real-Time Distributed Shared Dataspace*, which was supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW (grant CES.5009) and carried out at the Centrum voor Wiskunde en Informatica (CWI). The main research goal of the project was to analyse the behaviour of applications built on the top of shared dataspaces. To this end, we can divide the research topics in two broad parts:

- To specify a formal model of the shared dataspace architecture, the analysis of applications built on top of it ant to verify requirements imposed to the system.
- To create a suitable verification framework able to handle realistic applications.

Real-time critical systems, such as air control management systems, have to deal with severe requirements of reliability, fault-tolerance, extensibility, timeliness, efficiency, availability, ... These requirements make the task of building such systems extremely hard. A way of managing these severe requirements is to use software architectures [113]. A software architecture is basically a description of how the different components of a system are to be composed. In general, it provides a set of interfaces and algorithms that allow external applications to communicate and coordinate. One of the main advantages of using architectures is reusability, components and the architecture itself can be used in different applications.

Shared dataspace architectures implement repositories for data elements in such a way that external components communicate by sharing information through the repositories instead of interacting directly between each other. This feature contributes to the conception of modular applications. The architecture is in charge of handling the concurrent access of the processes to the shared resources. External processes have a unified view of the shared space, although the repository may be distributed or centralised [17].

Both distributed and centralised shared dataspace architectures were studied inside the project. In distributed dataspaces the information is stored in different devices that are connected through a network. In general, several copies of data elements reside in different locations. When a new data element is produced, it is copied in one or several places. The distributed data storage causes that sometimes the copies of the data elements contain different values. The architecture has to be aware of the possible lack of data consistency. An example of this kind of architecture is SPLICE [16, 74] which has been used in command and control systems on several frigates of various nations and for the control of the metro of Amsterdam. The research work about this architecture does not form part of the thesis. Simona Orzan, who participated in the same project, dedicated some efforts to investigate about SPLICE. Her results can be found in [97].

In this thesis, we are going to study JavaSpaces [91], which is a Sun Microsystems, Inc. architecture based on the Linda coordination language [25]. Apart from implementing a centralised shared space, it provides other services such as leasing, transactions and an event notification mechanism. JavaSpaces interfaces are standard and there are publicly available implementations. These fact motivates us to choose JavaSpaces as the centralised shared dataspace architecture to be investigated. To have a running implementation was an important issue in order to compare the theoretical results with real executions.

The circumstances under which the architectures are used are extreme: channels are unreliable, and computers can fail or even die completely. So the project has to address the question of how to build applications that guarantee the specified services. In order to be able to design reliable applications, we have to provide a framework that allows to formally prove that a given system behaves as expected [37].

An illustrative example of a characteristic application that can be easily implemented using a shared dataspace architecture, such as JavaSpaces, is the following radar-monitor system (see Figure 1). The radar *produces* packets (A entries) of different measurements taken from an external moving agent. A transformer processes the measurements by computing predictions of future moves of the investigated agent. The monitor *consumes* the processed data (B entries) and displays the trajectories of the moving objects.

The example introduces another of the main topics of the thesis: *replication*. In the radar-monitor system, we would like to have several transformers

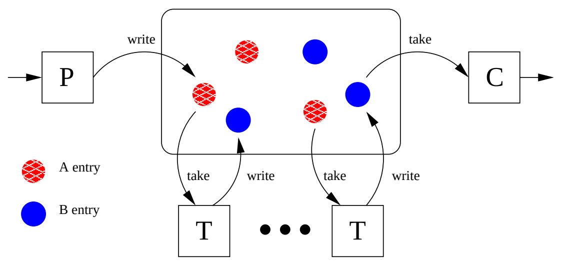


Figure 1: Producer-Transformer(s)-Consumer

making calculations at the same time in order to accelerate the display of the results. The problem then is how to transparently replicate certain processes in such a way that the behaviour of the system remains correct. Replication contributes to improve non-functional requirements such as efficiency, availability or robustness. In general replication is not straightforward, in fact it may introduce non-desired behaviours in the overall systems. For instance, if we replicate carelessly the transformer process in the system above, the change may cause duplication of the final results which will make the monitor display wrong trajectories. Some parts of the thesis are dedicated to investigating how to replicate processes and how to prove that the final application behaves as expected.

We have stressed that one of the main purposes of the analysis of shared dataspace architectures is to be able to verify the correctness of complex applications built on top of them. Formal verification includes different techniques which differ in the quality of the results and the human effort required to apply them. We can roughly distinguish three different techniques:

- *Testing.* It is maybe the most widely used verification technique. It consists of checking that some selected scenarios (*test* cases) behave as expected by means of simulation. In general, the selection of suitable *test* cases requires human expertise. The rest of the process can be fully automated. The main limitation of the technique is that in most of the cases it is not possible to cover all possible scenarios of the system and therefore to find all errors of the implementation. *Testing* becomes extremely hard for distributed systems in which execution of a test case may be non-deterministic due to, for example, the unpredictable latencies of the communication channels. In summary, *testing* only proves the correctness of some executions for some scenarios.

- *Theorem Proving.* This technique allows to prove that a complete system satisfies its specification, i.e., all behaviors of the implementation are correct. In general, one proceeds by systematically applying a set of inference rules to some formal description of the implementation of the system in order to prove that it satisfies its specification. For complex systems, the task is arduous and requires strong human interaction to find the proofs. *Theorem Proving* proves the correctness of all executions for all scenarios.

- *Model Checking.* It is a semi-automatic technique, that allows to prove the correctness of some scenarios against all possible behaviours of the system. To do so, a finite model of the system has to be built (see below). *Model Checking* proves the correctness of all executions for some scenarios.

The technique we have mainly focused on is the third one, which gives a good balance between completeness of the results and automation. In general, the application of model checking involves the following phases:

1. First, it is needed to formally specify the system to check. This specification has to be written in a formal language, such as logic or, in our case, process algebra.

2. The requirements of the system are also specified in the same language as the specification or in a different one. In general, temporal or modal logics can be used to write the properties that represent the requirements.

3. A model of the system is generated from the formal specification. It contains all possible behaviors. Some possible ways of representing the semantics of the specification are finite state automata or a labelled transition systems.

4. Finally, the satisfaction or refutation of the properties is checked against the model.

The formal specification of the system and the requirements requires human expertise. The checking of the satisfaction or refutation of the properties is automatically done by computers. The limit of automatic verification by model checking comes from the size of the models. In general, the parallel composition of the behaviors of processes generates large state spaces, due to the combinatorial growth, which can hardly be stored or manipulated by machines. Model checking in general can only handle systems with finite state space (the model checker *mucheck* [66], included in the μCRL toolset [12] is an exception that can handle infinite state systems) . During the last years many research has been conducted in order to attack the so-called *state space explosion* problem. Let us see some of the techniques dedicated to deal with the problem:

- *On-the-fly Model Checking* or *Bounded Model Checking.* The idea is to avoid the exploration of the full state space of a system [53]. The checking is done just in some parts. This technique allows to quickly find errors during early phases of the verification process. There is no benefit when one has to prove the correctness of the full system.

- *Symbolic Model Checking.* This technique consists of encoding systems using Binary Decision Diagrams (BDDs), a canonical form for boolean expressions [90]. These have traditionally been used as the underlying representation of the set of states. This compact form allows to handle systems with larger number of states than other standard ways of representing the semantics of a system. The efficiency of performing model checking on BDDs strongly depends on the order of the boolean variables. The main disadvantage is that a suitable order is in general hard to find and sometimes requires human intervention.

- *Partial Order Reduction.* In some cases the order of execution of some actions does not influence the correctness of the system [2]. Therefore, some sequences of actions can be eliminated from the state space without harm, reducing the complexity of the analysed system. A similar technique applied in the process algebraic setting consists on proving confluence of actions (see, for example [15]).

- *Distributed Model Checking.* This technique consists in using extra hardware resources to fight against the state explosion problem. The computation and the storage are done using different machines connected through a network. In the project, this technique was successfully explored by Simona Orzan, see for example [13, 14].

- *Abstract Interpretation.* It denotes a framework for program analysis. The seminal idea was to extract program approximations by removing uninteresting information [29, 30]. In order to obtain approximations, computations over concrete universes of data are performed over smaller abstract domains. A typical example of the technique is the so-called "rule of signs" used to determine the sign of arithmetic expressions by performing the computation only over the signs of the operators, i.e., the expression $-5 * 10$ is abstracted to $neg * pos$ which preserves the sign of the result.

In the first part of the thesis we focus on abstract interpretation (or simply, abstraction). The integration of this technique in the automatic verification framework allows to significantly reduce the complexity of the analyzed systems. This permits to apply model checking to large systems. The idea is to prove properties in the (small) abstract system and then to infer the satisfaction or refutation to the (large) concrete one. Abstract model checking integrates the following steps:

1. We depart from a concrete specification whose state space is too large or infinite and it cannot be handled by regular model checking techniques.

2. An abstract specification is built from the concrete, interpreting the concrete one over a smaller data domain (as we did for the "rule of signs"). This step usually requires some human intervention in order to select

suitable abstractions. In principle, the state space corresponding to the abstraction is significantly reduced.

3. We apply model checking techniques on the abstract system. The results of the abstract model checking can be inferred to the concrete system (following some rules).

4. Applying abstraction causes some loss of information, so in some cases it would not be possible to prove the satisfaction or refutation of some properties. In these cases, it would be necessary to find better abstractions.

We have titled this section *From Practice to Theory... and Back* which describes the research cycle of how the thesis was conceived. We started out with the main directives of the project by analysing the JavaSpaces architecture. Once we had specified a formal model of the architecture and we had proved its suitability by verifying small characteristic examples, we tried to do the same kind of analysis to non-trivial examples. Realistic case studies quickly discovered the limits of the verification framework which forced us to start investigating theoretical enhancements in order to improve the verification capabilities of the verification framework. Among the different possibilities, we have chosen to investigate abstraction because of the promising results that others obtained with it. This decision lead to obtain interesting theoretical results. Finally, to bring these results into practice motivated us to integrate the theory in our framework, we have dedicated some efforts to integrate the theory in our framework by developing generic tools and patterns suitable for verification of coordination models.

The order in which we present the contents of the thesis differs from how it was conceived. We first introduce the theoretical issues and then the practical ones. The next section is dedicated to describing what can be found in every chapter.

How to Read the Thesis?

The thesis is composed by two different parts, the first one is fully dedicated to abstraction and the second to coordination. Here, we give a short guide of what can be found in every chapter of the book. Figure 2 presents graphically the structure of the first part of the thesis.

Chapter 1: Labelled Transition Systems are usually used to capture the semantics of action-based systems, such as process algebraic specifications. So, in order to apply abstraction techniques to such specifications we are going to define the meaning of abstractions on Labelled Transition Systems. Basically, this chapter describes the relations between abstract and concrete systems and the inference rules about the satisfaction and refutation of properties.

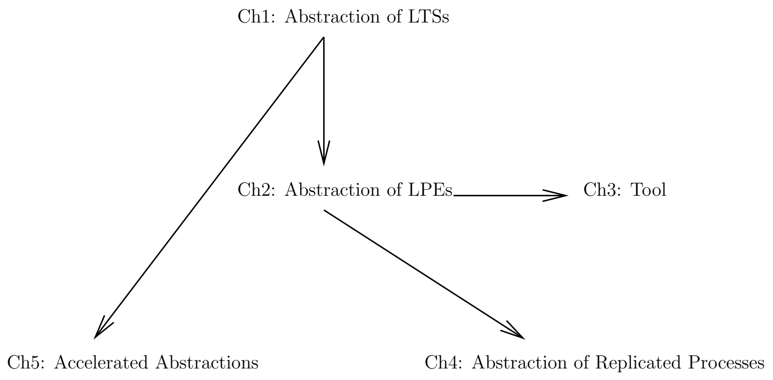

Ch1: Abstraction of LTSs
Ch2: Abstraction of LPEs
Ch3: Tool
Ch5: Accelerated Abstractions
Ch4: Abstraction of Replicated Processes

Figure 2: Organisation of the first part of the thesis

Chapter 2: If the first chapter is dedicated to study the semantic level of the specification, Chapter 2 focuses on the syntactic one. The idea is to generate abstraction directly from process algebraic specifications by relating concrete and abstract data domains. We give the conditions under which an abstract specification is a correct abstraction of a concrete one, and therefore can be used to infer the satisfaction of properties.

These first two chapters are the theoretical core of the thesis, the rest of the first part contains extensions or variations of them. They are based on the following publication: [109].

Chapter 3: To develop tools is a fundamental step in order to make the theoretical results applicable in practice. This chapter is dedicated to explaining an abstract interpretation toolkit that helps to apply abstraction to non-trivial specifications.

This chapter is based on the following publication: [100].

Chapter 4: In order to apply theory to practice, not only *tooling* is an indispensable step, but also the creation of reusable patterns. In this chapter, we present some abstraction patterns that can be used to verify systems composed by replicated processes. As we have stated above, transparent replication is an important issue when implementing efficient distributed systems, so here we try to contribute to facilitate the verification of systems composed by several copies of the same process.

This chapter is based on the following publication: [105].

Chapter 5: The loss of information caused by abstraction sometimes does not permit to infer some interesting properties about the progress of the system. In this chapter, we enhance the basic semantic framework presented in the first chapter. The extension enriches the expressiveness of abstractions.

The first two chapters of the second part can be read independently of the rest of the thesis. The third one applies the results of the abstraction part to coordination. Let us explain their contents:

Chapter 6: Here we describe the formal specification of JavaSpaces. The specification was modelled having in mind that the main purpose of the specification was to be used to verify applications built on top of it. The specification contains all the ingredients needed to implement interesting applications.
This chapter is based on the following publications: [106, 107].

Chapter 7: In this chapter, we use the formal model of JavaSpaces as a framework to analyse external applications. First, we give some small examples of verification and then we present a complex case study.

This chapter is based on the following publication: [108].

Chapter 8: This chapter is a conclusion of the second part of the thesis, in which we give some guidelines of how to apply the results on abstraction to characteristic JavaSpaces applications. It does not contain new technicalities but provides general ideas how the theoretical research contributes to the practice. This chapter closes the research cycle: *From Practice to Theory... and Back.*

Every chapter comes with its own introduction, where the main contributions are explained, the related work pointing the main references to similar research and a conclusion is given, where apart from giving a short summary of the contents of the chapter, some possible future directions of research are presented.

Part I

On Abstraction...

Chapter 1

Modal Abstractions of Labelled Transition Systems

This chapter introduces the basic theory of abstraction for transitions systems. Abstractions are represented by *Modal Labelled Transition Systems*, that are graphs whose transitions are labelled with actions that may have two modalities *may* and *must*. They are used to encode double approximations of concrete systems. We enhance the classical abstraction frameworks by allowing the possibility of abstracting actions. We prove that the abstractions are sound for the full action-based μ-calculus.

1.1 Introduction

The application of abstract interpretation to the verification of systems is suitable since it allows to formally transform possibly infinite instances of specifications into smaller and finite ones. By losing some information, we can compute a desirable view of the analysed system that preserves some interesting properties of the original. Abstract Interpretation has been successfully used to perform *control* and *data flow analysis* and to deal with the state space explosion problem. For a comprehensive introduction see [28].

The achievement of the first two chapters is to enhance existing process algebraic verification languages (e.g. LOTOS [38], μCRL [57]) with state-of-the-art abstract interpretation techniques that exist for state-based reactive systems. This chapter is dedicated to introducing the abstraction of transition systems, and the next chapter to the abstraction of process algebraic specifications.

There exist different techniques to relate abstractions with concrete systems. The two classical approaches use homomorphisms [26, 36] or Galois Connections [85, 31, 75, 55] . The first one is conceptually simpler, although the latter produces more precise abstractions that are sound for safety as well as for liveness properties. We adapt both approaches to action-based systems, commonly used to describe the semantics of process algebraic specifications, allowing abstraction of states, transitions and action labels.

Semantically, our method is based on *Modal Labelled Transition Systems* [84, 83]. MLTSs are *mixed* transition systems in which transitions are labelled with actions and with two modalities: *may* and *must.* They are appropriate structures to define abstraction/refinement relations between processes. *May* transitions determine the actions that possibly occur in all refinements of the system while *must* transitions denote the ones that do necessarily happen. On the one hand, the *may* part corresponds to an over-approximation that preserves *safety* properties of the concrete instance and on the other hand the *must* part under-approximates the model and reflects *liveness* properties.

A three-valued logic ensures that the theory can be used for proofs and refutations of temporal properties. We define approximations and prove that they are sound for all properties in the full (action-based) μ-calculus [81], including negation. To achieve this result, we had to extend existing theory by allowing abstraction and information ordering of action labels, which is already visible in the semantics of μ-calculus formulas.

The chapter is organised as follows. First we present the main results about MLTSs and the semantic abstraction of a *Labelled Transition Systems* (LTS) to modal LTSs following different approaches. Then, we introduce the logical characterisation of the abstractions, and the inference rules for satisfaction or refutation of properties from abstract to concrete systems. The fundamental proofs of the results of this theory are attached at the end of the chapter. At the conclusion, we give some explanation about the relation of this work with respect to the main references of the field. The next chapter is dedicated to the explanation of how to construct the abstraction directly from process algebraic specifications.

1.2 Abstraction of Transition Systems

The set of techniques, used to translate a concrete system to a "safe" abstract instance of it, is normally called abstract interpretation and it has already been studied for many years. These techniques have their roots in the seminal papers of Abstract Interpretation[1] by Cousot and Cousot [29, 30]. The main idea is to provide a relation between the concrete data domain and an abstract version of it in such a way that the interpretation of the system over the abstract domain preserves and/or reflects some properties of the original. Although, part of the results included in this section are well known in the field, we adapt classical frameworks for generating safe abstract approximations, by doing a non-trivial extension of them in order to allow the explicit abstraction of action labels. This will allow to build more expressive abstractions and, furthermore, to manipulate infinitely branching systems. Furthermore, we integrate in a uniform theory two broadly used approaches, the one based on homomorphic mappings between concrete and abstract domains and the other one based on Galois Connections introduced by Cousot and Cousot.

We start by presenting a small example that will be used as illustration of the techniques. The system is composed from two processes that communicate by sending natural numbers through a channel described as a FIFO buffer of size N (from now on, we assume $N \geq 2$), see Figure 1.1 below. The system may have an arbitrary number of states due to the size of the buffer. Furthermore it can be infinite if it receives data belonging to an infinite data domain:

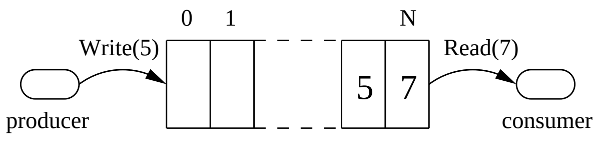


Figure 1.1: Simple buffer of size N

In order to verify properties of the system such as *"If the buffer is full, the producer cannot write anything"* we need to know neither how many items are in the buffer (we only have to know whether it is full or not) nor what is the exact value of the stored data. Therefore, we may consider an abstract and finite version of the system in which all these irrelevant details are omitted and that preserves the properties we are interested in. On the one hand, the content of the transferred items can be removed keeping only the information about the type of action performed (read or write) and, on the other hand, the FIFO list can be abstracted to a set of values determining the state of the buffer: *empty*, *full*, or anything in between: *middle*. The abstract representation of the system, although it loses some information of the original model, allows to check some interesting properties, as the ones presented above.

[1]We use abstract interpretation when we speak about the general framework and Abstract Interpretation (with capitals) to refer to Cousots' work.

Now, let us define some general concepts and then we will continue by introducing the different abstraction techniques. The semantics of a system can be defined by a *Labelled Transition System*:

Definition 1.2.1 We define a *Labelled Transition System* (LTS) as a tuple $(S, Act, \rightarrow, s_0)$ in which S is a non-empty set S of states, Act a non-empty set of transition labels, $\rightarrow$ is a possibly infinite set of transitions and s_0 in S is the initial state. A transition is a triple $s \xrightarrow{a} s'$ with $a \in Act$ and $s,s' \in S$.

Figure 1.2 illustrates the LTS corresponding to the example introduced above. Actions R and W denote respectively read and write operations.

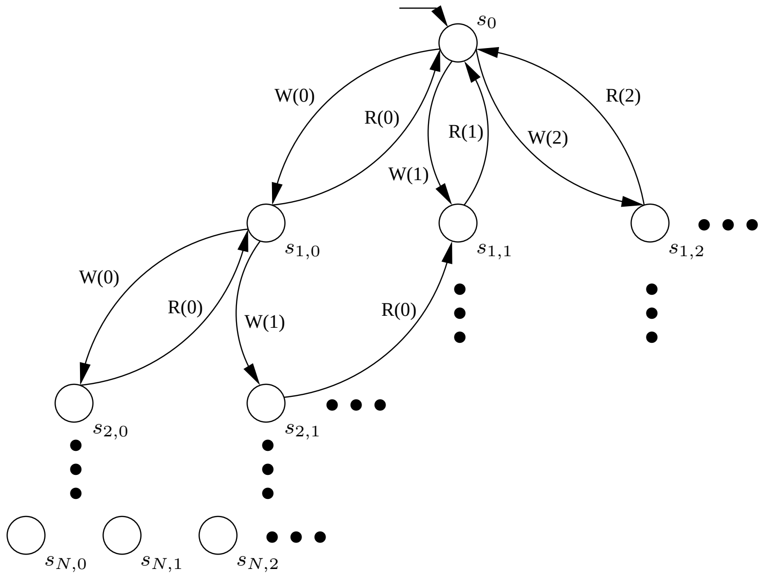


Figure 1.2: LTS of the buffer system

To model abstractions we are going to use a different structure that allows to represent approximations of the concrete system in a more suitable way. As introduced before, in *Modal Labelled Transition Systems* transitions have two modalities *may* and *must* which denote the possible and necessary steps in the refinements. This concept was introduced by Larsen and Thomsen [84]:

Definition 1.2.2 A *Modal Labelled Transition System* (MLTS) is a tuple $(S, Act, \rightarrow_\diamond, \rightarrow_\Box, s_0)$ where S, Act and s_0 are as in definition 1.2.1 and $\rightarrow_\diamond, \rightarrow_\Box$ are possibly infinite sets of (may or must) transitions of the form $s \xrightarrow{a}_x s'$ with $s,s' \in S$, $a \in Act$ and $x \in \{\diamond, \Box\}$. We require that every *must*-transition is a *may*-transition ($\xrightarrow{a}_\Box \subseteq \xrightarrow{a}_\diamond$).

MLTSs are suitable structures for stepwise refinements and abstractions. A refinement step of a system is done by preserving or extending the existing *must*-transitions and by preserving or removing the *may*-transitions. Abstraction is done the other way around (formal definitions will be presented in following sections). Note that every LTS corresponds to a trivially equivalent MLTS in which $\rightarrow_\Diamond = \rightarrow_\Box$, we called it concrete MLTS. Now, we introduce how to relate concrete and abstract systems.

1.2.1 Abstraction by Homomorphism

The first approach to extract abstract *Modal Labelled Transition Systems* from concrete MLTSs that we are going to present is based on homomorphisms that map concrete states and action labels to their abstract versions. This theory was introduced by Clarke and Long [26]. It is intuitively simple and it allows significant reductions of the state space.

Having a set of states S and a set of action labels Act with their corresponding abstract sets, denoted by $\widehat{S}$ and $\widehat{Act}$, we define a homomorphism H as a pair of total and surjective functions $\langle h_S, h_A \rangle$, where h_S is a mapping from states to abstract states, i.e., $h_S : S \rightarrow \widehat{S}$, and h_A maps action labels to abstract action labels, i.e., $h_A : Act \rightarrow \widehat{Act}$. The abstract state $\widehat{s}$ corresponds to all the states s for which $h_S(s) = \widehat{s}$, and the abstract action label $\widehat{a}$ corresponds to all the actions a for which $h_A(a) = \widehat{a}$. Then:

Definition 1.2.3 Given a concrete MLTS $\mathcal{M} = (S, Act, \rightarrow_\Diamond, \rightarrow_\Box, s_0)$, with $\rightarrow_\Diamond = \rightarrow_\Box$ and a homomorphism $H = \langle h_S, h_A \rangle$, we define the MLTS $\widehat{\mathcal{M}} = (\widehat{S}, \widehat{Act}, \dashrightarrow_\Diamond, \dashrightarrow_\Box, \widehat{s}_0)$, called the *minimal$_H$-abstraction* (denoted by $min_H(\mathcal{M})$) where $\widehat{s}_0$ is equal to $h_S(s_0)$ and the following conditions hold:

- $\widehat{s} \overset{\widehat{a}}{\dashrightarrow}_\Diamond \widehat{r} \iff \exists\, s, r, a.\, h_S(s) = \widehat{s} \wedge h_S(r) = \widehat{r} \wedge h_A(a) = \widehat{a} \wedge s \xrightarrow{a} r$
- $\widehat{s} \overset{\widehat{a}}{\dashrightarrow}_\Box \widehat{r} \iff \forall\, s. h_S(s) = \widehat{s}.\, (\exists\, r, a.\, h_S(r) = \widehat{r} \wedge h_A(a) = \widehat{a} \wedge s \xrightarrow{a} r)$

This definition gives the most accurate abstraction of a concrete system by using a given pair of homomorphisms, in other words the one that preserves most information of the original system. Less precise abstractions would contain more *may* transitions and/or fewer *must* transitions.

Figure[2] 1.3 shows the minimal abstraction of the buffer model in which only the states have been abstracted, action labels remain as in the original. H is equal to the pair $\langle h_S, Id_A \rangle$ in which h_S is defined as follows: it maps the initial state to the abstract state e, which means *empty*, the states in which there are N entries in the buffer to f, which represents *full*, and the rest of the states to m, which means something in the *middle*.

[2]In figures, *must*-transitions are represented by solid lines and *may*-transitions by dashed ones. For clarity, when there is a *must* transition we do not include the corresponding *may* one.

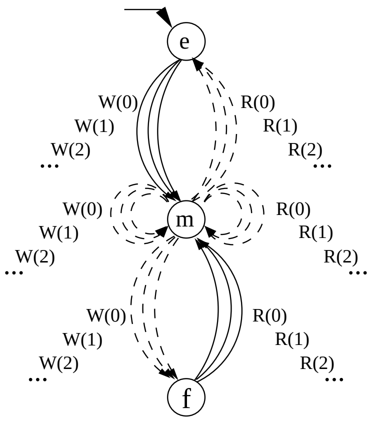


Figure 1.3: Abstract buffer size N; abstraction of states

We see that the system cannot be completely represented, even if the set of states is finite, because it is infinitely branching. Abstraction frameworks only based on abstraction of states cannot handle this kind of problems. We need also to apply abstraction on action labels in order to be able to use model checking techniques over the abstract system. We define h_A as follows: it maps all the write actions to $\widehat{w}$ and all the read actions to $\widehat{r}$.

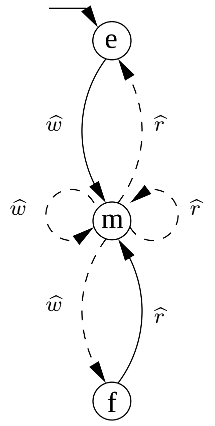


Figure 1.4: Abstract buffer size N; abstraction of states and action labels

In the final system (see Figure 1.4), by the combination of both abstractions, we have removed all the information about the values that are in the buffer and the transferred data, only preserving the information about whether the buffer is empty, full or neither of them. This abstraction allows to have a small finite model which keeps some information about the original. The example clearly illustrates the importance of the abstraction of action labels to avoid infinitely branching abstractions.

Figure 1.5 illustrates the abstraction definition. We have the abstract *must* transition $\widehat{s_1} \overset{\widehat{b}}{\rightarrow}_{\Box} \widehat{s_2}$ because all the states related to $\widehat{s_1}$ have a transition to a concrete state related to $\widehat{s_2}$ (the same applies to $\widehat{s_0} \overset{\widehat{a}}{\rightarrow}_{\Box} \widehat{s_0}$) . Furthermore, if there is some concrete state related to an abstract state $\widehat{s}$ with a transition to another state related to an abstract state $\widehat{r}$, then there is a *may* transition between $\widehat{s}$ and $\widehat{r}$. In the figure, these abstract transitions are marked by the dashed arrows. Actions labels can also be abstracted, in the example the concrete labels $\{a_0, a_1\}$ are mapped to the abstract label $\widehat{a}$ and $\{b_0, b_1\}$ to $\widehat{b}$ as is shown in the figure.

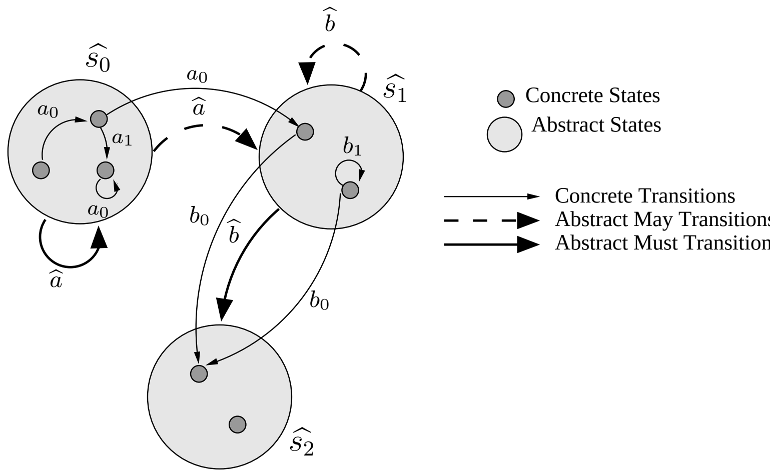


Figure 1.5: Example of modal abstraction of an LTS.

1.2.2 Abstraction by Galois Connection

Instead of using mappings between concrete and abstract domains we can define more complicated relations. For instance, the approach defined by Cousot and Cousot is based on Galois Connections [96] between domains. Formally:

Definition 1.2.4 Two functions α and γ over two partially ordered sets $(P, \subseteq)$ and $(Q, \preccurlyeq)$ such that $\alpha : P \to Q$ and $\gamma : Q \to P$ form a Galois Connection if and only if the following conditions hold:

1. α and γ are total and monotonic.
2. $\forall p : P.p \subseteq \gamma \circ \alpha(p)$.
3. $\forall q : Q.\alpha \circ \gamma(q) \preccurlyeq q$.

α is the lower adjoint and γ is the upper adjoint of the Galois Connection, and they uniquely determine each other. Galois Connections enjoy suitable properties to perform abstractions (see for example [31, 34]).

In this case we consider the concrete system built over the power-sets of states and actions. $\mathcal{P}(S)$ and $\mathcal{P}(Act)$ are partially ordered sets ordered by the set inclusion operator. The abstract system is built over $\widehat{S}$ and $\widehat{Act}$, both being posets equipped with some order $\preccurlyeq$. The order is a relation based on the precision of the information contained in the elements of the domain. To relate both systems, we define a pair G of Galois Connections: $G = \langle(\alpha_S, \gamma_S), (\alpha_A, \gamma_A)\rangle$. α is usually called the abstraction function and γ the concretisation function. Before giving the definition of abstraction, let us see the data domains for the running example.

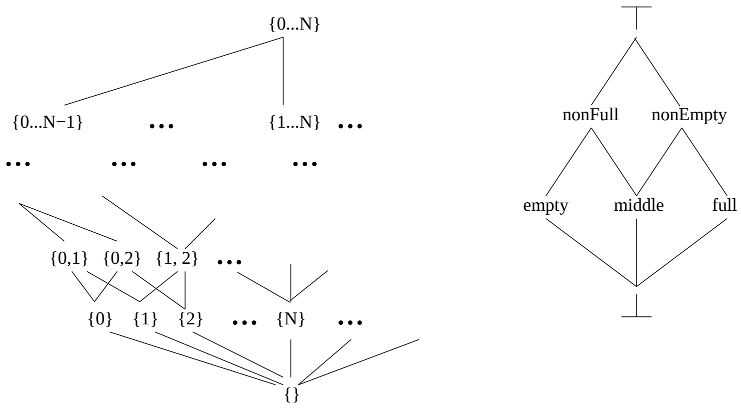


Figure 1.6: Concrete and abstract lattices of bounded buffers

Figure 1.6 shows two lattices corresponding to the representation of bounded buffers. The concrete lattice, left hand side, is built over the power sets of naturals, the upper and lower bounds are represented by the full set (*0,... ,N*) and the empty set ($\{\}$). It corresponds to the possible lengths of the buffer[3]. For instance the set $\{2\}$ means that the buffer contains exactly two elements, $\{1, 4\}$ means that it contains either $\{1\}$ or $\{4\}$ elements. The top element $\{0, ..., N\}$ represents all buffers of any length ($\leq N$). The abstract domain is the previously used set of abstract naturals *empty, middle* and *full* extended with two new values *nonEmpty* and *nonFull* and with $\top$ and $\bot$ in order to complete the lattice. We abbreviate the names as $\{\bot, e, m, f, nE, nF, \top\}$. The definition of $\alpha_S(L)$ for L is the states of the buffer which contain $l \in L$ number of elements:

[3]Note that this way of representing the buffers is already an abstraction.

if $L = \{0\}$ then *empty* else if $\forall l \in L.\, 0 < l < N$ then *middle*
else if $L = \{N\}$ then *full* else if $\forall l \in L.\, 0 < s \leq N$ then *nonEmpty*
else if $\forall l \in L.\, l < N$ then *nonFull* else if $L = \{\}$ then $\bot$ otherwise $\top$

We define $\gamma_S(\widehat{s}\,)$ as:

if $\widehat{s} = \bot$ then $\{\}$ else if $\widehat{s} =$ *empty* then $\{0\}$
else if $\widehat{s} =$ *middle* then $\{1, ..., N-1\}$ else if $\widehat{s} =$ *full* then $\{N\}$
else if $\widehat{s} =$ *nonEmpty* then $\{1, ..., N\}$ else if $\widehat{s} =$ *nonFull* then $\{0, ..., N-1\}$
else Nat

It is trivial to see that $\mathcal{P}(Nat)$, $\{\bot, e, m, f, nE, nF, \top\}$, α_S and γ_S form a Galois Connection. We can also define the abstraction of the action labels, the abstract lattice is presented in Figure 1.7. α_A of a set of action labels B only composed from write actions will be equal to $\widehat{w}$, $\alpha_A(B)$ with B composed from read actions will be equal to $\widehat{r}$, and if there are read and write actions in B then $\alpha_A(B)$ will be $\top$ and $\alpha_A(\{\})$ will be $\bot$. γ_A is defined according to α_A, as:

- $\gamma_A(\widehat{r}) = \{R(n) \mid n \in Nat\}$
- $\gamma_A(\widehat{w}) = \{W(n) \mid n \in Nat\}$
- $\gamma_A(\top) = \{X(n) \mid n \in Nat \wedge X \in \{R, W\}\}$
- $\gamma_A(\bot) = \{\}$

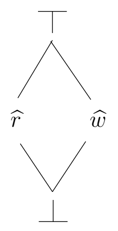


Figure 1.7: Lattice of abstract actions

We could have defined a more complicated lattice for abstract labels, for example, one that distinguishes between sending even and odd naturals. As in the case of the homomorphism, we define the minimal abstraction, as follows:

Definition 1.2.5 Given two systems $\mathcal{M}$ and $\widehat{\mathcal{M}}$ defined as in Definition 1.2.3 and a pair of Galois Connections G, $\widehat{\mathcal{M}}$ is the *minimal$_G$-abstraction* (denoted by $min_G(\mathcal{M})$) if and only if $s_0 \in \gamma_S(\widehat{s}_0)$ and the following conditions hold:

- $\widehat{s} \overset{\widehat{a}}{\multimap}_{\Diamond} \widehat{r} \iff \exists\, s \in \gamma_S(\widehat{s}),\, r \in \gamma_S(\widehat{r}),\, a \in \gamma_A(\widehat{a}).\, s \xrightarrow{a} r$
- $\widehat{s} \overset{\widehat{a}}{\multimap}_{\Box} \widehat{r} \iff \forall\, s \in \gamma_S(\widehat{s}).\, (\exists\, r \in \gamma_S(\widehat{r}),\, a \in \gamma_A(\widehat{a}).\, s \xrightarrow{a} r)$

The next figure presents part of the minimal abstraction of the buffer system[4].

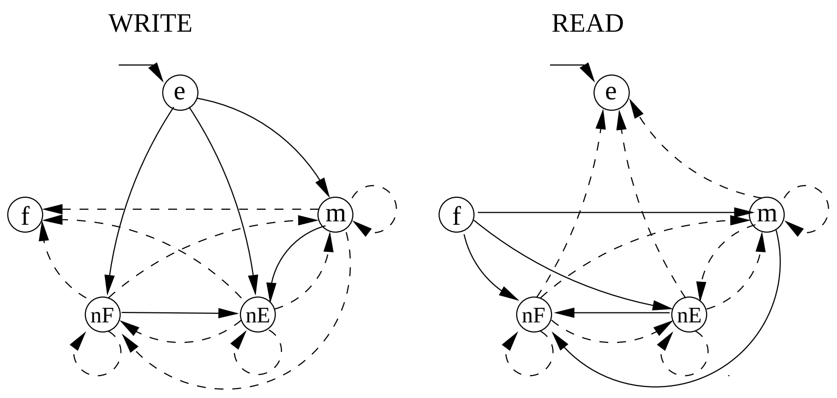


Figure 1.8: Abstract buffer size N (Galois Connection)

The order defined over the abstract lattices gives a relation about the precision of the information contained in the values. For example, *empty* has more accurate information than *nonFull* and the latter more than $\top$. Furthermore, the order induces a precision relation between transitions, for example:

- *empty* $\overset{\widehat{w}}{\multimap}_{\Box}$ *middle* is more precise than *empty* $\overset{\widehat{w}}{\multimap}_{\Box}$ *nonEmpty* because the transition points to a more precise state (*middle* $\preccurlyeq$ *nonEmpty*),
- *empty* $\overset{\widehat{w}}{\multimap}_{\Box}$ *nonEmpty* is more precise than *empty* $\overset{\widehat{w}}{\multimap}_{\Box}$ $\top$, and
- *empty* $\overset{\widehat{w}}{\multimap}_{\Box}$ *middle* is more precise than *empty* $\overset{\top}{\multimap}_{\Box}$ *middle*, because the transition is labelled with a more precise label ($\widehat{w} \preccurlyeq \top$) ...

Due to the order over the abstract states and actions, the minimal system defined by the above presented definition is saturated by *may* and *must* transitions, i.e. there are transitions that do not add any extra information. We can easily see in the previous figure that the *must* part is saturated for example, the transition $e \overset{\widehat{w}}{\multimap}_{\Box} nE$ does not add any information because we have $e \overset{\widehat{w}}{\multimap}_{\Box} m$ which is more precise. We can restrict our definition by requiring that the abstract transitions are performed only between the most precise descriptions

[4]For readability, we separate write transitions: $\overset{\widehat{w}}{\multimap}$ and read transitions $\overset{\widehat{r}}{\multimap}$ and we do not include transitions to and from the state $\top$, or labelled with the action $\top$.

of the concrete transitions, as done in Dams' theory [31]. To do so, for every abstract MLTS $\widehat{\mathcal{M}}$, first we define:

- For every $\widehat{s}$, let $\mathcal{M}_{(\widehat{s},\diamond)}$ be equal to the set of pairs (R, B) with R in $\mathcal{P}(S)$ and B in $\mathcal{P}(Act)$ such that $\exists\, s \in \gamma_S(\,\widehat{s}\,)$, $r \in R$, $a \in B$ such that $s \xrightarrow{a}_{\diamond} r$ is in $\mathcal{M}$.

Basically, $\mathcal{M}_{(\widehat{s},\diamond)}$ represents all sets of possible *may*-continuations of the concrete set of states related with $\widehat{s}$. Then we define, $\mathcal{M}^{min}_{(\widehat{s},\diamond)}$ as the set of minimal elements in $\mathcal{M}_{(\widehat{s},\diamond)}$:

- $\mathcal{M}^{min}_{(\widehat{s},\diamond)} = \{(R_m, B_m) \in \mathcal{M}_{(\widehat{s},\diamond)} \mid \forall (R, B) \in \mathcal{M}_{(\widehat{s},\diamond)}.\neg(R \subset R_m \wedge B \subset B_m)\}$.

The definition says that among all possible sets of continuations we select the ones that do not have any subset, i.e., the minimal ones. Observe that $\mathcal{M}^{min}_{(\widehat{s},\diamond)}$ will be always composed from pairs of singleton sets. Finally:

- Let $\widehat{\mathcal{M}}_{(\widehat{s},\diamond)}$ be the set of pairs: $\{(\widehat{r}, \widehat{a}) \mid \exists\, (R_{min}, B_{min}) \in \mathcal{M}^{min}_{(\widehat{s},\diamond)}.\widehat{r} = \alpha_S(R_{min}) \wedge \widehat{a} = \alpha_A(B_{min})\}$.

$\widehat{\mathcal{M}}_{(\widehat{s},\diamond)}$ is the abstraction of the minimal sets. Then, for every $\widehat{s}$, we keep the transitions of $\widehat{\mathcal{M}}$ such that $\widehat{s} \xrightarrow{\widehat{a}}_{\diamond} \widehat{r}$ if and only if $(\widehat{r}, \widehat{a}) \in \widehat{\mathcal{M}}_{(\widehat{s},\diamond)}$. This rule eliminates the redundant *may* transitions. To remove redundant *must* ones we proceed in the same way, the only change is in the first definition:

- For every $\widehat{s}$, the set $\mathcal{M}_{(\widehat{s},\square)}$ equals the set of pairs of (R, B) such that $\forall\, s \in \gamma_S(\,\widehat{s}\,)\, \exists\, r \in R,\, a \in B.s \xrightarrow{a}_{\square} r$ is in $\mathcal{M}$.

In the *must* case $\mathcal{M}^{min}_{(\widehat{s},\square)}$ is not necessarily composed from only singleton sets. After removing the redundant *may* and *must* transitions, in general the condition $\xrightarrow{a}_{\square} \subseteq \xrightarrow{a}_{\diamond}$ will not hold anymore. To preserve this property we add an extra rule requiring that for every *must* transition the system also has a *may* one.

Definition 1.2.6 Given a concrete system $\mathcal{M}$ the minimal restricted abstraction w.r.t. the Galois Connection G, denoted by $\widehat{\mathcal{M}}\!\downarrow$ is the MLTS obtained by applying the restriction rules to $min_G(\mathcal{M})$

Note that by doing the restriction no information is removed because only imprecise transitions are eliminated. Let us compute the *restricted* set of transitions for the example for the case $\widehat{s}$ equals *empty*:

- The following pairs will be in $\mathcal{M}_{(empty,\diamond)}$[5]:

[5]We denote by s_0 the initial state of Figure 1.2, by $s_{1,x}$ the concrete states of the first row, by $s_{2,x}$ the states of the second row etc... Note that for the states of the first row , $s_{1,x}$ is reached after the after the insertion of one entry x.

- $(\{s_{1,0}\}, \{w(0)\})$
- $(\{s_0, s_{1,0}\}, \{w(1), w(0)\})$
- $(\{s_{1,2}\}, \{w(2)\})$
- $(\{s_{1,0}, s_{N,1}\}, \{w(0), r(0)\})$
- ...

Therefore, $\mathcal{M}_{(empty,\diamond)}$ will be composed from an infinite number of pairs (R, B) such that there is at least one '$s_{1,x}$' in R and $w(d)$ in B with $d = x$.

- The set of the minimal pairs $\mathcal{M}^{min}_{(empty,\diamond)}$ will be: $\{(\{s_{1,x}\}, \{w(x)\})\}$ for any x.
- Considering that $\alpha_S(\{s_{1,x}\})$ is equal to *middle* and $\alpha_A(\{w(x)\})$ is equal to $\widehat{w}$ for any x, then $\widehat{\mathcal{M}}_{(empty,\diamond)}$ will be equal to $\{(middle, \widehat{w})\}$.
- Therefore, we just keep the transition $empty \xrightarrow{\widehat{w}}_{\diamond} middle$ and we remove all the rest of outgoing transitions from *empty*.

For the complete system, the following transitions are removed from Figure 1.8: $e \xrightarrow{\widehat{w}}_{\diamond,\square} nF$, $e \xrightarrow{\widehat{w}}_{\diamond,\square} nE$, $m \xrightarrow{\widehat{w}}_{\diamond} nF$, $m \xrightarrow{\widehat{r}}_{\diamond} nE$, $f \xrightarrow{\widehat{r}}_{\diamond,\square} nF$, $f \xrightarrow{\widehat{r}}_{\diamond,\square} nE$, $nF \xrightarrow{\widehat{w},\widehat{r}}_{\diamond} nF$, $nE \xrightarrow{\widehat{w},\widehat{r}}_{\diamond} nE$, $nF \xrightarrow{\widehat{r}}_{\diamond} nE$ and $nE \xrightarrow{\widehat{w}}_{\diamond} nF$. Figure 1.9 shows the result of the restriction.

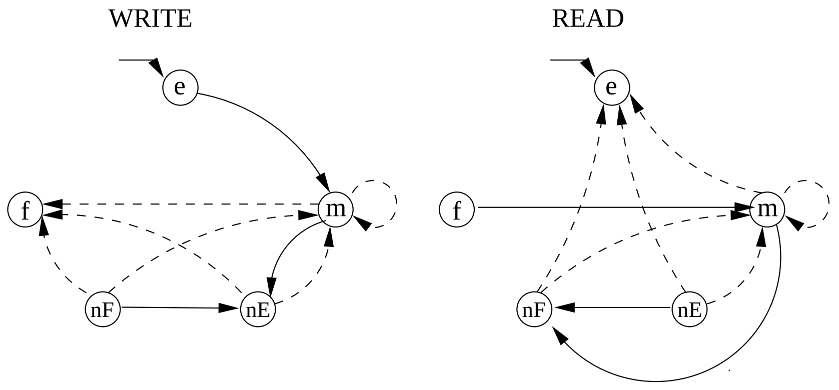


Figure 1.9: Minimal *restricted* abstract buffer

In general we would not compute the minimal *restricted* abstraction, but just approximations of it. The restricted form will be useful in order to characterise the information preserved by the abstractions as we will see in following sections.

1.3 Modal LTSs Approximations

So far, we have seen the definition of the minimal abstractions using either a homomorphism or a Galois Connection. However, we can have, as well, non minimal abstractions; let us formalise the approximation relation between MLTSs:

Definition 1.3.1 Given two MLTSs $\mathcal{M} = (S, Act, \rightarrow_{\Diamond}, \rightarrow_{\Box}, s_0)$ and $\mathcal{N} = (S,$ *Act,* $\rightarrowtriangle_{\Diamond}, \rightarrowtriangle_{\Box}, s_0')$ built over the same partially ordered sets of states $\langle S, \preccurlyeq_S \rangle$ and actions $\langle Act, \preccurlyeq_A \rangle$; $\mathcal{N}$ is an abstraction of $\mathcal{M}$, denoted by $\mathcal{M} \sqsubseteq_{\preccurlyeq} \mathcal{N}$, if the following conditions hold:

- $s_0 \preccurlyeq_S s_0'$
- $\forall\, s, a, r, s'.\, s \xrightarrow{a}_{\Diamond} r \,\wedge\, s \preccurlyeq_S s' \implies \exists\, a', r'.\, s' \overset{a'}{\rightarrowtriangle}_{\Diamond} r' \,\wedge\, r \preccurlyeq_S r' \wedge a \preccurlyeq_A a'$
- $\forall\, s', a', r', s.\, s' \overset{a'}{\rightarrowtriangle}_{\Box} r' \,\wedge\, s \preccurlyeq_S s' \implies \exists\, a, r.\, s \xrightarrow{a}_{\Box} r \wedge r \preccurlyeq_S r' \wedge a \preccurlyeq_A a'$

$\mathcal{M} \sqsubseteq_{\preccurlyeq} \mathcal{N}$ means that $\mathcal{N}$ is more abstract than $\mathcal{M}$ and it preserves all the information of the *may*-transitions of $\mathcal{M}$ and at least all *must* transitions present in $\mathcal{N}$ are reflected in $\mathcal{M}$. The *may* part of $\mathcal{N}$ is an over-approximation of $\mathcal{M}$ and the *must* part is an under-approximation. The refinement relation is the dual of the abstraction.

Note that for the homomorphism approach there is no order defined between states or actions so we substitute $\preccurlyeq$ by $=$. In this case abstractions are done simply by preserving or adding more *may* transitions and by preserving or removing some *must* transitions, i.e.:

- $s \xrightarrow{a}_{\Diamond} r \implies s \overset{a}{\rightarrowtriangle}_{\Diamond} r$
- $s \overset{a}{\rightarrowtriangle}_{\Box} r \implies s \xrightarrow{a}_{\Box} r$

Note that for arbitrary MLTSs and arbitrary orders $\preccurlyeq$, the abstraction predicate $\sqsubseteq_{\preccurlyeq}$ is not a partial order. This is because it is not reflexive. Let us consider a counter-example:

- $\mathcal{M}$ equals $s \xrightarrow{a}_{\Box} r$ and $s \xrightarrow{a}_{\Diamond} r$, with
- $s \preccurlyeq s'$
- If $\mathcal{M} \sqsubseteq_{\preccurlyeq} \mathcal{M}$, then by definition there should exists a' and r' with $r \preccurlyeq r'$, $a \preccurlyeq a'$ and $s' \xrightarrow{a'}_{\Diamond} r'$, which is not the case for $\mathcal{M}$

1.4 Logical Characterisation

To express properties about systems [101], we are going to adapt the highly expressive temporal logic (action-based) μ-calculus [81], see also [114], which is defined by the following syntax, where A stands for a set of action labels:

$$\varphi ::= \mathsf{T} \mid \mathsf{F} \mid \neg\,\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [A]\varphi \mid \langle A\rangle\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$$

To give some intuition about the semantics, first we are going to present the standard semantics of the logic for labelled transition systems and later we will define them for *Modal Labelled Transition Systems.*

All states satisfy T and none F. $\wedge$ matches all the states that match two subformulas. $\vee$ matches all the states that match one of two formulas. The negation of a formula φ matches all the states that do not satisfy the formula φ. The universal operator $[A]\varphi$ holds in a state in which all possible continuations by actions of A satisfy φ. The existential operator $\langle A\rangle\varphi$ holds in a state in which there exists a transition by an action belonging to A that satisfies φ. Y denotes a propositional variable. μ and ν are the minimal and maximal fixpoint operators respectively. We assume that propositional variable Y always appear under the scope of a fixpoint operator.

In order to guarantee the existence of fixpoints, formulas have to be syntactically monotonic. Since the negation is not monotonic, to get the monotonicity, every occurrence of a fixpoint variable has to be under the scope of an even number of negations. As illustration, Figure 1.10 presents the standard semantics of the logic.

$$\begin{aligned}
[\![\mathsf{F}]\!]_\rho &= \emptyset \\
[\![\mathsf{T}]\!]_\rho &= S \\
[\![\neg\varphi]\!]_\rho &= S \backslash [\![\varphi]\!]_\rho \\
[\![\varphi_1 \wedge \varphi_2]\!]_\rho &= [\![\varphi_1]\!]_\rho \cap [\![\varphi_2]\!]_\rho \\
[\![\varphi_1 \vee \varphi_2]\!]_\rho &= [\![\varphi_1]\!]_\rho \cup [\![\varphi_2]\!]_\rho \\
[\![[A]\varphi]\!]_\rho &= \{s \mid \forall\, r, a.\, a \in A \wedge s \xrightarrow{a} r \implies r \in [\![\varphi]\!]_\rho\} \\
[\![\langle A\rangle\varphi]\!]_\rho &= \{s \mid \exists\, r, a.\, a \in A \wedge s \xrightarrow{a} r \wedge r \in [\![\varphi]\!]_\rho\} \\
[\![Y]\!]_\rho &= \rho(Y) \\
[\![\mu Y.\varphi]\!]_\rho &= \bigcap\{S \mid [\![\varphi]\!]_{(\rho \oslash [S/Y])} \subseteq S\} \\
[\![\nu Y.\varphi]\!]_\rho &= \bigcup\{S \mid S \subseteq [\![\varphi]\!]_{(\rho \oslash [S/Y])}\}
\end{aligned}$$

Figure 1.10: Semantics of the action based μ-calculus on LTSs

ρ represents the propositional context, it is a function that assigns sets of states to propositional variables, i.e., $\rho : Y \to 2^S$. The fixpoint is defined using the results of Tarski's fixpoint theorems [115]. $\rho \oslash [S/Y]$ denotes context overriding, the value of the propositional variable Y is set to S. Normally, the following equivalences are used to reduce the complexity of manipulating properties:

- $\mathsf{T} = \neg\mathsf{F}$

- $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $[A]\varphi = \neg\langle A\rangle\neg\varphi$
- $\mu Y.\varphi(Y) = \neg\nu Y.\neg\varphi(\neg Y)$

Now we proceed by presenting the semantics of the formulas over *Modal*-LTS. Following [75], a given formula is interpreted dually over an MLTS, i.e. there will be two sets of states that satisfy it. A set of states that necessarily satisfy the formula and a set of states that possibly satisfy it. Thus, the semantics of the formulas are given by $[\![\varphi]\!]_\rho \in 2^S \times 2^S$ and the projections $[\![\varphi]\!]^{nec}_\rho$ and $[\![\varphi]\!]^{pos}_\rho$ give the first and the second component, respectively. We show below the simultaneous recursive definitions of the evaluation of a state formula.

$$\begin{aligned}
[\![\mathsf{F}]\!]_\rho &= \langle\emptyset,\emptyset\rangle\\
[\![\mathsf{T}]\!]_\rho &= \langle S,S\rangle\\
[\![\neg\varphi]\!]_\rho &= \langle S\backslash[\![\varphi]\!]^{pos}_\rho, S\backslash[\![\varphi]\!]^{nec}_\rho\rangle \quad \text{(Note the switch of } \textit{pos} \text{ and } \textit{nec}\text{)}\\
[\![\varphi_1\wedge\varphi_2]\!]_\rho &= \langle [\![\varphi_1]\!]^{nec}_\rho \cap [\![\varphi_2]\!]^{nec}_\rho, [\![\varphi_1]\!]^{pos}_\rho \cap [\![\varphi_2]\!]^{pos}_\rho\rangle\\
[\![\varphi_1\vee\varphi_2]\!]_\rho &= \langle [\![\varphi_1]\!]^{nec}_\rho \cup [\![\varphi_2]\!]^{nec}_\rho, [\![\varphi_1]\!]^{pos}_\rho \cup [\![\varphi_2]\!]^{pos}_\rho\rangle\\
[\![[A]\varphi]\!]_\rho &= \langle\{s \mid \forall r,a,a'.\, a\in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_\Diamond r \implies r \in [\![\varphi]\!]^{nec}_\rho\},\\
&\qquad \{s \mid \forall r,a,a'.\, a\in A \wedge a' \preccurlyeq a \wedge\, s \xrightarrow{a'}_\Box r \implies r \in [\![\varphi]\!]^{pos}_\rho\}\rangle\\
[\![\langle A\rangle\varphi]\!]_\rho &= \langle\{s \mid \exists r,a,a'.\, a\in A \wedge a' \preccurlyeq a \wedge s \xrightarrow{a'}_\Box r \wedge r \in [\![\varphi]\!]^{nec}_\rho\},\\
&\qquad \{s \mid \exists r,s,a'.\, a\in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_\Diamond r \wedge r \in [\![\varphi]\!]^{pos}_\rho\}\rangle
\end{aligned}$$

Figure 1.11: Semantics of the action based μ-calculus on *Modal*-LTSs (part I)

We remark, also, that from the semantics of the negation follows:

- s necessarily satisfies $\neg\varphi$ if and only if s not possibly satisfies φ.
- s possibly satisfies $\neg\varphi$ if and only if s not necessarily satisfies φ.

Note that the precise order between action labels plays an important role in the definition of the semantics of the modalities. Let us consider the following simple example:

- Given the MLTS, $s \xrightarrow{b}_\Box r$ and $s \xrightarrow{b}_\Diamond r$
- With $a \preccurlyeq b \preccurlyeq c$
- Suppose we want to prove $s \in [\![\langle\{a\}\rangle\mathsf{T}]\!]^{pos}_\rho$. Then, even if there is not an a transition from s, the formula will be true because there exists a *possible* transition from s labelled with some label less precise than a.

- Therefore, if we can *possibly* do an imprecise action then we should be able to do a more precise one.
- Suppose we want to prove $s \in [\![\langle\{c\}\rangle\mathsf{T}]\!]^{nec}_{\rho}$ (from s we can *necessarily* do a c step). The formula will be true because we can *necessarily* do a more precise action.
- Therefore, if we can *necessarily* do a more precise action then we should be able to do a more abstract one.

In this case, the propositional context $\rho : Y \to 2^S \times 2^S$ assigns two sets of states to propositional variable. ρ^{nec} denotes the first component of the image of the function and ρ^{pos} the second component. Now we proceed with the semantics of the fixpoint operators.

$$\begin{aligned}
[\![Y]\!]_{\rho} &= \rho(Y)\\
[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho} &= \langle \textstyle\bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho},\\
&\quad\ \textstyle\bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{pos}_{\rho}\rangle\\
[\![\mu Y^{\lambda}.\varphi]\!]_{\rho} &= \langle [\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])},\\
&\quad\ [\![\varphi]\!]^{pos}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])}\rangle\\
[\![\nu Y^{<\lambda}.\varphi]\!]_{\rho} &= \langle \textstyle\bigcap_{\beta<\lambda}[\![\nu Y^{\beta}.\varphi]\!]^{nec}_{\rho},\\
&\quad\ \textstyle\bigcap_{\beta<\lambda}[\![\nu Y^{\beta}.\varphi]\!]^{pos}_{\rho}\rangle\\
[\![\nu Y^{\lambda}.\varphi]\!]_{\rho} &= \langle [\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\nu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])},\\
&\quad\ [\![\varphi]\!]^{pos}_{(\rho\oslash[[\![\nu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])}\rangle
\end{aligned}$$

Figure 1.12: Semantics of the action based μ-calculus on *Modal*-LTSs (part II)

The definition of the fixpoints is done by simultaneous recursion on the ordinal λ. Instead of using the standard way of defining the fixpoints as in Figure 1.10, we have defined them by approximants. It is known (see for example [114]) that, for monotonic functions in a lattice, we can construct the least fixpoint by iterating from the bottom element in an increasing chain until the limit which is the fixed point. In both finite and countable domains, the length of the iteration is bounded. In the second case the length will be transfinite. The minimal *necessary* fixpoint $[\![\mu Y.\varphi]\!]^{nec}_{\rho}$ is the smallest ordinal for which $[\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho} = [\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho}$, the same holds for the *possible* minimal fixpoint and for the maximal fixpoints.

We have chosen this representation of the fixpoint operators because it allows to prove results about the inference of properties between systems applying simple induction schemes on the ordinals (hence, on the values of the context ρ). By using, the standard way, as in Figure 1.10, we would have to deal with arbitrary contexts ρ which complicates the task of proving the results.

We remark that:

- $[\![\mu Y^{<0}.\varphi]\!]_\rho = \langle \emptyset, \emptyset \rangle$
- $[\![\nu Y^{<0}.\varphi]\!]_\rho = \langle S, S \rangle$
- $[\![\mu Y^{0}.\varphi]\!]_\rho = [\![\varphi]\!]_{\rho \oslash [\langle \emptyset, \emptyset \rangle / Y]}$
- $[\![\nu Y^{0}.\varphi]\!]_\rho = [\![\varphi]\!]_{\rho \oslash [\langle S, S \rangle / Y]}$

We say that a state s necessarily satisfies a formula φ, denoted by $s \models^{nec}_\rho \varphi$, if and only if $s \in [\![\varphi]\!]^{nec}_\rho$ and dually s possibly satisfies a formula φ, denoted by $s \models^{pos}_\rho \varphi$, if and only if $s \in [\![\varphi]\!]^{pos}_\rho$. The notation $\mathcal{M}, s \models^{nec}_\rho \varphi$ or $s \in [\![\varphi]\!]^{nec}_{\rho,\mathcal{M}}$ means that the state s satisfies the formula φ on $\mathcal{M}$. We sometimes omit the context ρ or the model $\mathcal{M}$, when it is not needed or can be easily inferred. A trivial property is that for concrete MLTSs $[\![\varphi]\!]^{pos} = [\![\varphi]\!]^{nec}$.

Lemma 1.4.1 For the semantics of the μ-calculus over MLTS, the following equivalences hold:

- $\mathsf{T} = \neg \mathsf{F}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $[A]\varphi = \neg\langle A \rangle \neg\varphi$ and $\mu Y.\varphi(Y) = \neg \nu Y.\neg\varphi(\neg Y)$

An interesting property of the semantics is that if s necessarily satisfies a formula φ then also s possibly satisfies φ. As we will see, from this fact follow some other basic results of the theory. Unfortunately, in general the property does not hold for the semantics given in Figure 1.11. Let us reconsider the example of the previous page, with the MLTS containing the transitions $s \xrightarrow{b}_\Box r$ and $s \xrightarrow{b}_\Diamond r$, and the action labels $a \preccurlyeq b \preccurlyeq c$. We want prove the following properties:

1. $s \models^{nec} \langle \{c\} \rangle \mathsf{T}$
2. $s \models^{pos} \langle \{c\} \rangle \mathsf{T}$

Following the definitions (1) is true however (2) is false, which gives a counter-example. To avoid this kind of undesired results, we can use some restriction on the action sets. Let us define the saturated sets of actions, as follows:

- $\mathrm{SAT}(A) = \{a' \mid \exists\, a \in A \wedge a' \preccurlyeq a\}$

Now we can formally formulate the property.

Lemma 1.4.2 For any closed formula φ, containing only saturated sets of actions, the following property holds:

- $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$

This follows from the fact that every *must*-transition is also a *may*-transition and the fact that the sets of actions are saturated. Without this condition we would be able to prove $s \models^{nec} \varphi$ and $s \models^{nec} \neg\varphi$ for some φ, which would lead to an *inconsistent* logic. In fact, it cannot be proved for any formula, $s \models^{nec} \varphi$ and also $s \models^{nec} \neg\varphi$, i.e. the necessarily interpretation is consistent and it is always possible to prove $s \models^{pos} \varphi$ or $s \models^{pos} \neg\varphi$ which means that the possibly interpretation is complete. We express this in the two following lemmas:

Lemma 1.4.3 If $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$, then the *necessary* interpretation is consistent, i.e., $[\![\varphi \wedge \neg\varphi]\!]^{nec} = \emptyset$

Lemma 1.4.4 If $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$, then the *possible* interpretation is complete, i.e., $[\![\varphi \vee \neg\varphi]\!]^{pos} = S$.

The semantics gives a 3-valued logic:

- s necessarily satisfies φ.
- s possibly satisfies φ but not necessarily satisfies φ.
- s not possibly satisfies φ.

Note that for any *concrete* MLTS (system in which every *may* transition is also *must*) $[\![\varphi]\!]^{nec}$ will be equal to $[\![\varphi]\!]^{pos}$, although if the formula is interpreted over an abstract system the set of states that *possibly* satisfy the formula but not *necessarily* represents the loss of information caused by the abstraction.

1.4.1 Property Preservation of MLTSs Approximations

Now, we are going to characterise the relations between the properties satisfied by different MLTSs. There exists a relation between the abstraction order $\sqsubseteq$ and the properties satisfied. In fact, if a system *necessarily* satisfies a property then the formula will hold for all refinements of the system. On the other hand, if a system does not *possibly* satisfy a property then none of the refinements satisfy it. This idea is stated in the following Theorem:

Theorem 1.4.5 Given two MLTSs $\mathcal{M}$ and $\mathcal{N}$, over the same sets of states and labels S and *Act*, with $\mathcal{M} \sqsubseteq_{\preccurlyeq} \mathcal{N}$, for any closed formula φ and for all s and s' in S such that $s \preccurlyeq s'$,:

- $\mathcal{N}, s' \models^{nec} \varphi \implies \mathcal{M}, s \models^{nec} \varphi$
- $\mathcal{N}, s' \not\models^{pos} \varphi \implies \mathcal{M}, s \not\models^{pos} \varphi$

This result is useful because, by performing symbolic abstractions (see the next chapter) we generate approximations of the minimal abstraction, so the Theorem states that we can still infer the satisfaction/refutation of properties from the approximation to the original. The proof of the Theorem can be found at the end of the chapter. Note that we are not requiring φ to be composed by saturated set of actions, this condition is only needed for consistency and completeness.

1.4.2 Property Preservation of MLTSs Abstractions

Since the abstraction of a system preserves some information of the original one, the idea is to prove properties on the abstract and then to infer the result for the original. In order to define the inference rules we have to consider the fact that actions are abstracted, so action labels of the concrete and of the abstract system belong to different sets. Abstract formulas are built over labels from a set $\widehat{Act}$ and we want to infer the satisfaction to a system built over concrete actions of a set Act. To characterise the inference rules, we can think of different approaches:

Option 1 ***(Wrong).*** We would like to have some rules of the type:

- If the state $\widehat{s}$ *necessarily* satisfies the formula $\widehat{\varphi}$ on the abstract model, then s (with $s \in \gamma_S(\widehat{s})$) satisfies the formula φ on the concrete.
- If the state $\widehat{s}$ does not *possibly* satisfy the formula $\widehat{\varphi}$ on the abstract model, then s (with $s \in \gamma_S(\widehat{s})$) does not satisfy the formula on the concrete.

Where φ is a concrete formula (over concrete action labels) and $\widehat{\varphi}$ is an abstract formula obtained by substituting every action set A in φ by its abstraction $\alpha_A(A)$. And the semantics of the satisfaction of a formula over a concrete system is the one defined in Figure 1.10. This approach would give problems. Let us consider the following counter-example:

- Let $s \overset{a(0)}{\longrightarrow}_{(\Diamond,\Box)} r$ be the concrete system.
- $\alpha_S(\{s\}) = \widehat{s}$, $\alpha_S(\{r\}) = \widehat{r}$, and $\alpha_A(\{a(0)\}) = \alpha_A(\{a(1)\}) = \widehat{a}$.
- A correct abstraction of the system could be: $\widehat{s} \overset{\widehat{a}}{\rightarrowtriangle}_{\Box} \widehat{r}$ and $\widehat{s} \overset{\widehat{a}}{\rightarrowtriangle}_{\Diamond} \widehat{r}$
- We want to prove $s \models^{nec} \langle\{a(1)\}\rangle\mathsf{T}$ (which is trivially false on the concrete system).
- The abstraction of the formula would be $\langle\{\widehat{a}\}\rangle\mathsf{T}$, which is *necessarily* true on the abstract system. Which contradicts the concrete result.

The problem is that $\widehat{s} \overset{\widehat{a}}{\rightarrowtriangle}_{\Box} \widehat{r}$ means that between all the concrete states related to $\widehat{s}$ exists a transition to a state related to $\widehat{r}$ labelled with an action related to $\widehat{a}$. And not that for all actions related to $\widehat{a}$ there exists such a transition. So, we know that there is a transition but we do not know exactly which one. In order to deal with this lack of information we propose a second option.

Option 2 *(Correct).* We are going to define the meaning of the satisfaction relation of an *abstract* formula $\widehat{\varphi}$ on a concrete state. Let $[\![\widehat{\varphi}]\!]_{\rho,G_A}$ represent the concrete states that satisfy an abstract formula. G_A is a Galois Connection (α_A, γ_A) on the action labels (G_A is replaced by h_A for the homomorphisms). The semantics is as in Figure 1.10, the only change is in the modal operators:

$$\begin{aligned}
[\![\langle \widehat{A} \rangle \widehat{\varphi}]\!]^{nec}_{\rho,G_A} &= \{s \mid \exists\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge a \in \gamma_A(\widehat{a}) \wedge s \xrightarrow{a}_{\Box} r \wedge r \in [\![\widehat{\varphi}]\!]^{nec}_{\rho,G_A}\} \\
[\![[\widehat{A}] \widehat{\varphi}]\!]^{nec}_{\rho,G_A} &= \{s \mid \forall\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge \widehat{a} \preccurlyeq \alpha_A(\{a\}) \wedge s \xrightarrow{a}_{\Diamond} r \implies r \in [\![\widehat{\varphi}]\!]^{nec}_{\rho,G_A}\}
\end{aligned}$$

Figure 1.13: Necessary semantics of an abstract formula over a concrete system

The possible semantics are equivalent. A concrete state s necessarily satisfies the abstract formula $\widehat{\varphi}$, denoted by $s \models^{nec}_{\rho,G_A} \widehat{\varphi}$, if and only if $s \in [\![\widehat{\varphi}]\!]^{nec}_{\rho,G_A}$. Note that the only change from the abstract semantics is done in the existential operator, we see that a concrete state satisfies a formula of the type $\langle\{\widehat{a}\}\rangle\varphi$ if there exists a transition labelled with a concrete action that is included in the concretisation of $\widehat{a}$. In this case the equivalence $[A]\varphi = \neg\langle A\rangle\neg\varphi$ does not hold in general, the other equivalences do

Now we can give the property preservation result, for the Galois Connection approach:

Theorem 1.4.6 Let $\mathcal{M}$ be the concrete MLTS $(S, Act, \rightarrow_{\Diamond}, \rightarrow_{\Box}, s_0)$, in which $\rightarrow_{\Diamond} = \rightarrow_{\Box}$, $G = \langle(\alpha_S, \gamma_S), (\alpha_A, \gamma_A)\rangle$ be a Galois Connection between the sets $(\mathcal{P}(S), \mathcal{P}(Act))$ and $(\widehat{S}, \widehat{Act})$, $\widehat{\mathcal{M}}{\downarrow}$ be the MLTS $(\widehat{S}, \widehat{Act}, \dashrightarrow_{\Diamond}, \dashrightarrow_{\Box}, \widehat{s_0})$ denoting the minimal (restricted) abstraction of $\mathcal{M}$, w.r.t. G. And finally let $\widehat{\varphi}$ be a closed formula over $\widehat{Act}$. Then for all $s \in S$ and $\widehat{s} \in \widehat{S}$ such that $s \in \gamma_S(\widehat{s})$:

- $\widehat{\mathcal{M}}{\downarrow}, \widehat{s} \models^{nec} \widehat{\varphi} \implies \mathcal{M}, s \models^{nec}_{G_A} \widehat{\varphi}$

- $\widehat{\mathcal{M}}{\downarrow}, \widehat{s} \not\models^{pos} \widehat{\varphi} \implies \mathcal{M}, s \not\models^{pos}_{G_A} \widehat{\varphi}$

The proof (see Section 1.6) follows from the fact that every *may* trace of $\mathcal{M}$ is mimicked on $\widehat{\mathcal{M}}{\downarrow}$ by some related states and, on the other hand, every *must* trace of $\widehat{\mathcal{M}}{\downarrow}$ is present in $\mathcal{M}$. For the homomorphic case the Theorem can be stated similarly:

Theorem 1.4.7 Let $\mathcal{M}$ be the concrete MLTS $(S, Act, \rightarrow_{\Diamond}, \rightarrow_{\Box}, s_0)$, in which $\rightarrow_{\Diamond} = \rightarrow_{\Box}$, $H = \langle h_S, h_A\rangle$ be a pair or homomorphisms between (S, Act) and $(\widehat{S}, \widehat{Act})$, $\widehat{\mathcal{M}}{\downarrow}$ be the MLTS $(\widehat{S}, \widehat{Act}, \dashrightarrow_{\Diamond}, \dashrightarrow_{\Box}, \widehat{s_0})$ denoting the minimal (restricted) abstraction of $\mathcal{M}$, w.r.t. H. And finally let $\widehat{\varphi}$ be a closed formula over $\widehat{Act}$. Then for all $s \in S$ and $\widehat{s} \in \widehat{S}$ such that $s = h_S(\widehat{s})$:

- $\widehat{\mathcal{M}}{\downarrow}, \widehat{s} \models^{nec} \widehat{\varphi} \implies \mathcal{M}, s \models^{nec}_{h_A} \widehat{\varphi}$

- $\widehat{\mathcal{M}}{\downarrow}, \widehat{s} \not\models^{pos} \widehat{\varphi} \implies \mathcal{M}, s \not\models^{pos}_{h_A} \widehat{\varphi}$

In this case the semantics are as follows[6]:

$$\begin{aligned}
[\![\langle \widehat{A} \rangle \widehat{\varphi}]\!]^{nec}_{\rho,h_A} &= \{s \mid \exists\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge h_A(a) = \widehat{a} \wedge s \xrightarrow{a}_{\Box} r \wedge r \in [\![\widehat{\varphi}]\!]^{nec}_{\rho,h_A}\} \\
[\![[\widehat{A}] \widehat{\varphi}]\!]^{nec}_{\rho,h_A} &= \{s \mid \forall\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge h_A(a) = \widehat{a} \wedge s \xrightarrow{a}_{\Diamond} r \implies r \in [\![\widehat{\varphi}]\!]^{nec}_{\rho,h_A}\}
\end{aligned}$$

Figure 1.14: Necessary semantics of an abstract formula over a concrete system

1.5 Conclusion

This chapter includes the basic definitions to abstract action-based systems that are used, in general, to describe the semantics of process algebraic specifications. We support three kinds of abstractions:

- *Abstraction of states:* It is done either by providing a mapping between concrete and abstract states, or by using a Galois Connection between two partially ordered sets representing the concrete and the abstract universe of states. In the second case we have an order that defines the precision of the states. More precise states contain more information.
- *Abstraction of transitions:* By using the modalities (*may* and *must*) we join two approximations of the behaviours of the concrete system and over- and under-approximation. Adding *may*-transitions or removing *must*-transitions produces more abstracted systems.
- *Abstraction of actions:* Actions are abstracted in the same way as states, this can produce more expressive abstractions and furthermore helps to deal with infinitely branching systems.

The three abstractions and the approaches to relate abstract and concrete systems are combined in a homogeneous manner. Abstract systems are proved to preserve properties described in the action-based μ-calculus. The next chapter is dedicated to explaining how to extract correct approximations from system specifications.

Related Work: A complete description of the work of the past almost thirty years (from the former papers of Cousot and Cousot [29, 30]) would be an enormous task. A good overview is presented in [28]. There, the author presents a brief overview of the theory and pointers to some selected work. An extended version of the paper can be found on the author's web site that counts 47 pages of bibliography. The amount of references gives an idea of the influence of the theory on the field of static analysis and program verification.

[6] $[a]\varphi = \neg\langle a \rangle \neg\varphi$ holds for the homomorphic case.

The first papers by Cousots were difficult to digest and some time had to pass in order to be assimilated by the verification community. Important contributions were done by Jones and Nielson [77], Dams [31] and Loiseaux and al. [85], to make the former theories suitable to apply in automatic verification. They give characterisations of the abstract and concrete systems in terms of the properties that they satisfy. The characterisations are presented by considering different fragments of the logic. Some more effort was needed to unify the different results in a homogeneous theory.

Kelb in [78] integrates the possibility of inferring the satisfaction and the refutation of properties by defining a 3-valued semantics of a logic. This work is continued by Huth, Godefroid and al. [75, 55]. They base their models in modal transition systems which where previously defined by Larsen and Thomsen [84, 83]. Modal transitions system, as we have seen, are suitable structures to model under-specified systems.

On the other side, the work in [36, 26] use a different framework to define abstractions. The simplicity of their approach strongly influenced the automatic techniques of verification. One of the main problems of abstraction theories is how to find good abstractions. The use of simpler ways of relating systems allowed to partially automate the search for abstractions. [112, 80] worked on the unification of the different abstraction approaches. Schmidt characterises abstraction relations in terms of binary relations that define simulations between systems. Pnueli presents in a homogeneous way control and data abstraction.

Our work inherits from all the cited references. We have done an extension of the theories to allow the possibility of abstracting actions. Since labelled transition systems are the most common structure used to define the semantics of process algebra, our extension permits the integration of classical abstraction techniques into the process theory [8].

1.6 Proofs

Proof (Lemma 1.4.1):

We prove the *necessary* semantics, the *possible* is proved in the same way. From now on, for simplicity, we only include the context ρ when it is needed:

Case: *Necessary False*

1. We have to prove $[\![\mathsf{T}]\!]^{nec} = [\![\neg\mathsf{F}]\!]^{nec}$.
2. By semantics of negation $[\![\neg\mathsf{F}]\!]^{nec} = \neg[\![\mathsf{F}]\!]^{pos}$. Therefore,
3. $S \subseteq \neg\emptyset = S$ which trivially proves the case.

$\Box$ (Case)

Case: *Necessary Disjunction*

1. We have to prove $[\![\varphi_1 \vee \varphi_2]\!]^{nec} = [\![\neg(\neg\varphi_1 \wedge \neg\varphi_2)]\!]^{nec}$.

2. By semantics of negation $[\![\neg(\neg\varphi_1 \wedge \neg\varphi_2)]\!]^{nec} = \neg[\![\neg\varphi_1 \wedge \neg\varphi_2]\!]^{pos}$. So,
3. $\neg([\![\neg\varphi_1]\!]^{pos} \cap [\![\neg\varphi_2]\!]^{pos})$ which is equal to $\neg(\neg[\![\varphi_1]\!]^{nec} \cap \neg[\![\varphi_2]\!]^{nec})$, then
4. $\neg(\neg[\![\varphi_1]\!]^{nec} \cap \neg[\![\varphi_2]\!]^{nec})$ equals $[\![\varphi_1]\!]^{nec} \cup [\![\varphi_2]\!]^{nec} = [\![\varphi_1 \vee \varphi_2]\!]^{nec}$, which proves the case.

□ (Case)

Case: *Necessary Universal*

1. We have to prove $[\![[A]\varphi]\!]^{nec} = [\![\neg\langle A\rangle\neg\varphi]\!]^{nec}$
2. By semantics of negation $[\![\neg\langle A\rangle\neg\varphi]\!]^{nec}$ is $\neg[\![\neg\langle A\rangle\neg\varphi]\!]^{pos}$, which is equal to
3. $\neg\{s \mid \exists\, r, a, a'.\, a \in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_{\diamond} r \wedge r \in [\![\neg\varphi]\!]^{pos}\}$, which is equal to $\{s \mid \forall\, r, a, a'.\, a \in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_{\diamond} r \implies r \notin [\![\neg\varphi]\!]^{pos}\}$ which is equal to
4. $\{s \mid \forall\, r, a, a'.\, a \in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_{\diamond} r \implies r \in [\![\varphi]\!]^{nec}\}$ which is equal to $[\![[A]\varphi]\!]^{nec}$ and proves the case.

□ (Case)

Case: *Necessary Fixpoint*

We are going to prove that the equivalence is true by applying transfinite induction on the ordinals. So, for all value of λ assuming that the lemma is true for any $\beta < \lambda$. Note that we treat all ordinals as limits of the lower ones, therefore we do not need to distinguish different inductive cases.

1. We have to prove:
   (a) $[\![\mu Y^{<\lambda}.\varphi(Y)]\!]^{nec}_{\rho} = [\![\neg\nu Y^{<\lambda}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho}$
   (b) $[\![\mu Y^{\lambda}.\varphi(Y)]\!]^{nec}_{\rho} = [\![\neg\nu Y^{\lambda}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho}$
2. By Induction Hypothesis, we know that for all $\beta < \lambda$, $[\![\mu Y^{\beta}.\varphi(Y)]\!]^{nec}_{\rho} = [\![\neg\nu Y^{\beta}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho}$
3. From definition we know: $[\![\neg\nu Y^{<\lambda}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho} = \neg[\![\nu Y^{<\lambda}.\neg\varphi(\neg Y)]\!]^{pos}_{\rho}$ which is equal to $\neg\bigcap_{\beta<\lambda}[\![\nu Y^{\beta}.\neg\varphi(\neg Y)]\!]^{pos}_{\rho}$, which is equal to
4. $\bigcup_{\beta<\lambda}\neg[\![\nu Y^{\beta}.\neg\varphi(\neg Y)]\!]^{pos}_{\rho} = \bigcup_{\beta<\lambda}[\![\neg\nu Y^{\beta}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho}$ by I.H. this is equal to
5. $\bigcup_{\beta<\lambda}[\![\nu Y^{\beta}.\varphi(Y)]\!]^{nec}_{\rho} = [\![\nu Y^{<\lambda}.\varphi(Y)]\!]^{nec}_{\rho}$ which proves (1.a).
6. We proceed with (1.b), $[\![\neg\nu Y^{\lambda}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho} = \neg[\![\nu Y^{\lambda}.\neg\varphi(\neg Y)]\!]^{pos}_{\rho}$ which equals $\neg[\![\neg\varphi(\neg[[\![\nu Y^{<\lambda}.\neg\varphi(\neg Y)]\!]^{pos}_{\rho}])]\!]^{pos}$, which is equal to

7. $[\![\varphi([[\![\neg\nu Y^{<\lambda}.\neg\varphi(\neg Y)]\!]^{nec}_{\rho}])]\!]^{nec}$, then
8. by (1.a), equals $[\![\varphi([[\![\mu Y^{<\lambda}.\varphi(Y)]\!]^{nec}_{\rho}Y])]\!]^{nec} = [\![\mu Y^{\lambda}.\varphi(Y)]\!]^{nec}_{\rho}$, that proves the case

$\square$ (Case)

$\square$(*Lemma*)

Proof (Lemma 1.4.2):
The proof is done by performing simultaneously structural induction over the formula φ and transfinite induction on the ordinal λ. We need the lemma for closed formulas only. The semantics of closed formulas is independent of the valuation. However, because we will do the proof by induction on the formula, we have to consider open subformulas as well. Because the lemma does not hold for open formulas for all valuations, we must formulate the induction formula with care. So we will prove that for all open formulas ϕ, for all valuations ρ such that for any Y, $\rho^{nec}(Y) \subseteq \rho^{pos}(Y)$ holds, the following statement holds:

- $[\![\varphi]\!]^{nec}_{\rho} \subseteq [\![\varphi]\!]^{pos}_{\rho}$

This means in particular that in order to apply the induction hypothesis in the fixpoint cases, we must check that the modified valuation satisfies the constraint. Then, for any property φ, for any ρ (under the constrain explained above), and for any λ assuming that the lemma is true for any $\beta < \lambda$, we consider the following cases:

Case: *False*

1. We have to prove[7] $[\![\mathsf{F}]\!]^{nec} \subseteq [\![\mathsf{F}]\!]^{pos}$, which is trivial because $\emptyset \subseteq \emptyset$

$\square$ (Case)

Case: *Negation*

1. We have to prove $[\![\neg\varphi]\!]^{nec} \subseteq [\![\neg\varphi]\!]^{pos}$, which is equivalent to $\neg[\![\varphi]\!]^{pos} \subseteq \neg[\![\varphi]\!]^{nec}$.
2. By Induction Hypothesis, we have $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$, which trivially proves the case.

$\square$ (Case)

Case: *Conjunction*

1. We have to prove $[\![\varphi_1 \wedge \varphi_2]\!]^{nec} \subseteq [\![\varphi_1 \wedge \varphi_2]\!]^{pos}$, which is equivalent to $[\![\varphi_1]\!]^{nec} \cap [\![\varphi_2]\!]^{nec} \subseteq [\![\varphi_1]\!]^{pos} \cap [\![\varphi_2]\!]^{pos}$

[7]For now on, for simplicity, we drop the propositional context when it is not needed.

2. By Induction Hypothesis, we have $[\![\varphi_1]\!]^{nec} \subseteq [\![\varphi_1]\!]^{pos}$ and $[\![\varphi_2]\!]^{nec} \subseteq [\![\varphi_2]\!]^{pos}$, which trivially proves the case.

$\Box$ (Case)

Case: *Existential operator*

1. We have to prove that for all A, $[\![\langle \mathrm{SAT}(A)\rangle\varphi]\!]^{nec} \subseteq [\![\langle \mathrm{SAT}(A)\rangle\varphi]\!]^{pos}$
2. By Induction Hypothesis, we have $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$
3. Assume $s \in [\![\langle \mathrm{SAT}(A)\rangle\varphi]\!]^{nec}$, then by definition $\exists\, r, a, a'.\, a \in \mathrm{SAT}(A) \wedge a' \preccurlyeq a \wedge s \xrightarrow{a'}_{\Box} r \wedge r \in [\![\varphi]\!]^{nec}$
4. Let us assume r, a and a' such that:
   (a) $s \xrightarrow{a'}_{\Box} r$
   (b) $a \in \mathrm{SAT}(A)$
   (c) $a' \preccurlyeq a$
   (d) $r \in [\![\varphi]\!]^{nec}$
5. By definition of MLTS for every *must* transition there is a *may* transition, hence $s \xrightarrow{a'}_{\Diamond} r$.
6. By definition of saturated, (4.b) and (4.c) imply $a' \in \mathrm{SAT}(A)$.
7. By I.H., $r \in [\![\varphi]\!]^{pos}$. So,
8. if $s \in [\![\langle \mathrm{SAT}(A)\rangle\varphi]\!]^{nec}$ then $\exists\, r, a' \in \mathrm{SAT}(A).\, s \xrightarrow{a'}_{\Diamond} r \wedge r \in [\![\varphi]\!]^{pos}$ which implies
9. $s \in [\![\langle \mathrm{SAT}(A)\rangle\varphi]\!]^{pos}$ that proves the case.

$\Box$ (Case)

Case: *Variable Valuation*

1. We have to prove $[\![Y]\!]^{nec}_{\rho} \subseteq [\![Y]\!]^{pos}_{\rho}$.
2. By the constrain on the propositional context, we know $\rho^{nec}(Y) \subseteq \rho^{pos}(Y)$ which proves the case.

$\Box$ (Case)

Case: *Minimal Fixpoint*

1. We have to prove that for any λ:
   (a) $[\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho} \subseteq [\![\mu Y^{<\lambda}.\varphi]\!]^{pos}_{\rho}$

(b) $[\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho} \subseteq [\![\mu Y^{\lambda}.\varphi]\!]^{pos}_{\rho}$

assuming (a) and (b) are true for all $\beta < \lambda$.

2. By definition $[\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho} = \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho}$
3. By (trans)-Induction Hypothesis we know that for any $\beta < \lambda$ the following equality holds, $[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho} \subseteq [\![\mu Y^{\beta}.\varphi]\!]^{pos}_{\rho}$, therefore
4. $\bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec} \subseteq \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{pos}$, which proves (1.a).
5. We proceed with (1.b), by definition $[\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho} = [\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])}$ and $[\![\mu Y^{\lambda}.\varphi]\!]^{pos}_{\rho} = [\![\varphi]\!]^{pos}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])}$
6. By (1.a), we know that $[\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho} \subseteq [\![\mu Y^{<\lambda}.\varphi]\!]^{pos}_{\rho}$ so
7. $\rho \oslash [[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y]$ satisfies the constrain presented above. Therefore,
8. By Induction Hypothesis on the structure of φ, $[\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])} \subseteq [\![\varphi]\!]^{pos}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho}/Y])}$ which proves the case.

$\Box$ (Case)

$\Box$(Lemma)

Proof (Lemma 1.4.3):

1. $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos} \iff [\![\varphi]\!]^{nec} \cap \neg[\![\varphi]\!]^{pos} = \emptyset$
2. By the semantics of negation and conjunction, $\emptyset = [\![\varphi]\!]^{nec} \cap \neg[\![\varphi]\!]^{pos} = [\![\varphi]\!]^{nec} \cap [\![\neg\varphi]\!]^{nec} = [\![\varphi \wedge \neg\varphi]\!]^{nec}$, which proves the Lemma.

$\Box$(Lemma)

Proof (Lemma 1.4.4):

1. $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos} \iff [\![\varphi]\!]^{nec} \cap \neg[\![\varphi]\!]^{pos} = \emptyset$ which is equivalent to $\neg[\![\varphi]\!]^{nec} \cup [\![\varphi]\!]^{pos} = S$
2. By the semantics of negation and disjunction, $S = \neg[\![\varphi]\!]^{nec} \cup [\![\varphi]\!]^{pos} = [\![\neg\varphi]\!]^{pos} \cup [\![\varphi]\!]^{pos} = [\![\neg\varphi \vee \varphi]\!]^{pos}$, which proves the Lemma.

$\Box$(Lemma)

Proof: (Theorem 1.4.5) Following the same reasoning of the proof of previous Lemma 1.4.2 (see above), we will prove the following stronger statement: for all s, s' with $s \preccurlyeq s'$, for all open formulas φ, and for all ρ and $\bar{\rho}$, such that:

- $s' \in \bar{\rho}^{nec}(Y) \implies s \in \rho^{nec}(Y)$
- $s \in \rho^{pos}(Y) \implies s' \in \bar{\rho}^{pos}(Y)$

it holds that:

- $\mathcal{N}, s' \models^{nec}_{\bar{\rho}} \varphi \implies \mathcal{M}, s \models^{nec}_{\rho} \varphi$
- $\mathcal{N}, s' \not\models^{pos}_{\bar{\rho}} \varphi \implies \mathcal{M}, s \not\models^{pos}_{\rho} \varphi$

This in turn is proved by induction on the structure of the formula φ considering at the same time the *possible* and *necessary* semantics and transfinite induction on the ordinal λ. We consider the following cases[8]:

Case: *Necessary False*

1. We have to prove $\mathcal{N}, s' \models^{nec} \mathsf{F} \implies \mathcal{M}, s \models^{nec} \mathsf{F}$
2. $[\![\mathsf{F}]\!]^{nec}_{\mathcal{N}}$ and $[\![\mathsf{F}]\!]^{nec}_{\mathcal{M}}$ are both $\emptyset$ which trivially proves the case.

□ (Case)

Case: *Necessary Negation*

1. We have to prove $\mathcal{N}, s' \models^{nec} \neg\varphi \implies \mathcal{M}, s \models^{nec} \neg\varphi$
2. By the semantics of the negation (1) is equal to $\mathcal{N}, s' \not\models^{pos} \varphi \implies \mathcal{M}, s \not\models^{pos} \varphi$
3. By Induction Hypothesis, we know $\mathcal{N}, s' \not\models^{pos} \varphi \implies \mathcal{M}, s \not\models^{pos} \varphi$ which trivially proves the case.

□ (Case)

Case: *Necessary Conjunction*

1. We have to prove $\mathcal{N}, s' \models^{nec} \varphi_1 \wedge \varphi_2 \implies \mathcal{M}, s \models^{nec} \varphi_1 \wedge \varphi_2$
2. By Induction Hypothesis, we know $\mathcal{N}, s' \models^{nec} \varphi_1 \implies \mathcal{M}, s \models^{nec} \varphi_1$ and $\mathcal{N}, s' \models^{nec} \varphi_2 \implies \mathcal{M}, s \models^{nec} \varphi_2$ therefore,
3. By the semantics of the conjunction $\mathcal{N}, s' \models^{nec} \varphi_1 \wedge \varphi_2 \implies \mathcal{N}, s' \models^{nec} \varphi_1 \wedge \mathcal{N}, s' \models^{nec} \varphi_2$, then by I.H. follows trivially the case.

[8]Remember that we denote with $\rightarrow$ the transitions of $\mathcal{M}$ and $\twoheadrightarrow$ the ones of $\mathcal{N}$, and $\mathcal{N}$ is more abstract than $\mathcal{M}$, i.e., $\mathcal{M} \sqsubseteq_{\preccurlyeq} \mathcal{N}$

□ (Case)

Case: *Necessary Existential*

1. We have to prove $\mathcal{N}, s' \models^{nec} \langle A \rangle \varphi \implies \mathcal{M}, s \models^{nec} \langle A \rangle \varphi$
2. Assume $\mathcal{N}, s' \models^{nec} \langle A \rangle \varphi$, then $\exists\, r', a, a'.\, a \in A \wedge a' \preccurlyeq a \wedge s' \xrightarrow{a'}_{\Box} r' \wedge \mathcal{N}, r' \models^{nec} \varphi$
3. Assume some r', a and a' such that
   (a) $a \in A$
   (b) $a' \preccurlyeq a$
   (c) $s' \xrightarrow{a'}_{\Box} r'$
   (d) $\mathcal{N}, r' \models^{nec} \varphi$
4. By Induction Hypothesis, we know that for all $r \in \mathcal{M}$ and $r' \in \mathcal{N}$, such that $r \preccurlyeq r'$ follows $\mathcal{N}, r' \models^{nec} \varphi \implies \mathcal{M}, r \models^{nec} \varphi$
5. By definition of approximation, $s \preccurlyeq s'$ and (3.c), there exist b and r such that $s \xrightarrow{b}_{\Box} r$ and $r \preccurlyeq r' \wedge b \preccurlyeq a'$, then
6. Assume some b and r such that:
   (a) $r \preccurlyeq r'$
   (b) $b \preccurlyeq a'$
   (c) $s \xrightarrow{b}_{\Box} r$
7. By transitivity of $\preccurlyeq$, (3.b) and (6.b) follows that $b \preccurlyeq a$.
8. By I.H., (3.d) and (6.a) follows $\mathcal{M}, r \models^{nec} \varphi$
9. Therefore, if $\mathcal{N}, s' \models^{nec} \langle A \rangle \varphi$ we have proved that $\exists\, r, a, b.\, a \in A \wedge b \preccurlyeq a \wedge s \xrightarrow{b}_{\Box} r \wedge \mathcal{M}, r \models^{nec} \varphi$ or what is the same $\mathcal{M}, s \models^{nec} \langle A \rangle \varphi$, which proves the case.

□ (Case)

Case: *Necessary Variable Valuation*

1. We have to prove $\mathcal{N}, s' \models^{nec}_{\bar{\rho}} Y \implies \mathcal{M}, s \models^{nec}_{\rho} Y$
2. By the restriction on ρ and $\bar{\rho}$, we know $s' \in \bar{\rho}^{nec}(Y) \implies s \in \rho^{nec}(Y)$ which trivially proves the case.

□ (Case)

Case: *Necessary Minimal Fixpoint*

1. We have to prove that for any λ:

   (a) $s' \in [\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} \implies s \in [\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$

   (b) $s' \in [\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} \implies s \in [\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$

   assuming (a) and (b) are true for all $\beta < \lambda$.
2. By definition $[\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} = \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}}$
3. By (trans)-Induction Hypothesis we know that for any $\beta < \lambda$, $s' \in [\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} \implies s \in [\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$.
4. If $s' \in \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}}$ implies there exists some $\beta < \lambda$ such that $s' \in [\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}}$, then
5. Assuming $s' \in [\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}}$ by (3) implies $s \in [\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$, so
6. $s' \in \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}}$ implies $s \in \bigcup_{\beta<\lambda}[\![\mu Y^{\beta}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$, which proves the (1.a).
7. We proceed with (1.b). $[\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} = [\![\varphi]\!]^{nec}_{(\bar{\rho}\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\bar{\rho},\mathcal{N}}/Y]),\mathcal{N}}$
8. The same holds for $\mathcal{M}$, $[\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho,\mathcal{M}} = [\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho,\mathcal{M}}/Y]),\mathcal{M}}$
9. By (1.a), we know that $s' \in [\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} \implies s \in [\![\mu Y^{<\lambda}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$, so
10. $\bar{\rho} \oslash [[\![\mu Y^{<\lambda}.\varphi]\!]_{\bar{\rho},\mathcal{N}}/Y]$ and $\rho_{\mathcal{M}} \oslash [[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho,\mathcal{M}}/Y]$ satisfy the constrain presented above. Therefore,
11. Then we apply the Induction Hypothesis on the structure of φ, $s' \in [\![\varphi]\!]^{nec}_{(\bar{\rho}\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\bar{\rho},\mathcal{N}}/Y]),\mathcal{N}} \implies s \in [\![\varphi]\!]^{nec}_{(\rho\oslash[[\![\mu Y^{<\lambda}.\varphi]\!]_{\rho,\mathcal{M}}/Y]),\mathcal{M}}$. Therefore, $s' \in [\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\bar{\rho},\mathcal{N}} \implies s \in [\![\mu Y^{\lambda}.\varphi]\!]^{nec}_{\rho,\mathcal{M}}$ which proves the case.

□ (Case)

The rest of the proofs for the *necessary* part follow by the equivalences proved in Lemma 1.4.1. The proofs for the *possible* part are as follows:

- F, $\wedge$, $\neg$ and *variable valuation* are analogous as the *necessary* ones.
- The *possible* existential is dual to the one presented, it is included below for illustration.
- The *fixpoint* operator is identical to the *necessary* one. It is enough to change *nec* for *pos* and to swap $\mathcal{M}$ with $\mathcal{N}$.

Case: *Possible Existential*

1. We have to prove $\mathcal{N}, s' \not\models^{pos} \langle A \rangle \varphi \implies \mathcal{M}, s \not\models^{pos} \langle A \rangle \varphi$ or what is the same $\mathcal{M}, s \models^{pos} \langle A \rangle \varphi \implies \mathcal{N}, s' \models^{pos} \langle A \rangle \varphi$
2. Assume $\mathcal{M}, s \models^{pos} \langle A \rangle \varphi$, then $\exists\, r, a, a'.\, a \in A \wedge a \preccurlyeq a' \wedge s \xrightarrow{a'}_{\diamond} r \wedge \mathcal{M}, r \models^{pos} \varphi$
3. Assume some r, a and a' such that
   (a) $a \in A$
   (b) $a \preccurlyeq a'$
   (c) $s \xrightarrow{a'}_{\diamond} r$
   (d) $\mathcal{M}, r \models^{pos} \varphi$
4. By Induction Hypothesis, we know that for all $r \in \mathcal{M}$ and $r' \in \mathcal{N}$, such that $r \preccurlyeq r'$ follows $\mathcal{M}, r \models^{pos} \varphi \implies \mathcal{N}, r' \models^{pos} \varphi$
5. By definition of approximation, $s \preccurlyeq s'$ and (3.c), there exist b and r' such that $s' \xrightarrow{b}_{\diamond} r'$ and $r \preccurlyeq r' \wedge a' \preccurlyeq b$, then
6. Assume some b and r' such that:
   (a) $r \preccurlyeq r'$
   (b) $a' \preccurlyeq b$
   (c) $s' \xrightarrow{b}_{\diamond} r'$
7. By transitivity of $\preccurlyeq$, (3.b) and (6.b) follows that $a \preccurlyeq b$.
8. By I.H., (3.d) and (6.a) follows $\mathcal{N}, r' \models^{pos} \varphi$
9. Therefore, if $\mathcal{M}, s \models^{pos} \langle A \rangle \varphi$ we have proved that $\exists\, r', a, b.\, a \in A \wedge a \preccurlyeq b \wedge s' \xrightarrow{b}_{\diamond} r' \wedge \mathcal{N}, r' \models^{pos} \varphi$, or what is the same $\mathcal{N}, s' \models^{pos} \langle A \rangle \varphi$ which proves the case.

$\square$ (Case)

$\square$ (Lemma)

Proof: (Theorem 1.4.6) The proof is done by structural induction over the formula φ simultaneously over the *necessary* and *possible* parts. As in the proof of Theorem 1.4.5 we are going to prove the statement for open formulas, or all $s, \widehat{s}$ with $s \in \gamma_S(\widehat{s})$, for all open formulas φ, and for all ρ and $\widehat{\rho}$, such that:

- $\widehat{\mathcal{M}}\!\downarrow, \widehat{s} \in \widehat{\rho}^{nec}(Y) \implies \mathcal{M}, \mathcal{M} \in \rho^{nec}(Y)$
- $\mathcal{M}, s \in \rho^{pos}(Y) \implies \widehat{\mathcal{M}}\!\downarrow, \widehat{s} \in \widehat{\rho}^{pos}(Y)$

the following statement holds:

- $\widehat{\mathcal{M}}\!\downarrow, \widehat{s} \models^{nec}_{\widehat{\rho}} \widehat{\varphi} \implies \mathcal{M}, s \models^{nec}_{\rho, G_A} \widehat{\varphi}$

- $\widehat{\mathcal{M}}\!\downarrow, \widehat{s} \not\models^{pos}_{\widehat{\rho}} \widehat{\varphi} \implies \mathcal{M}, s \not\models^{pos}_{\rho, G_A} \widehat{\varphi}$

The base cases: F, $\neg$ and $\wedge$ are trivial. We just consider the following cases:

Case: *Necessary Existential*

1. We have to prove $\widehat{\mathcal{M}}\!\downarrow, \widehat{s} \models^{nec} \langle \widehat{A} \rangle \widehat{\varphi} \implies \mathcal{M}, s \models^{nec}_{G_A} \langle \widehat{A} \rangle \widehat{\varphi}$
2. Assume $\widehat{s} \models^{nec} \langle \widehat{A} \rangle \widehat{\varphi}$ (from now on we drop the reference to the system) then $\exists\, \widehat{r}, \widehat{a}, \widehat{b}.\, \widehat{a} \in \widehat{A} \wedge \widehat{b} \preccurlyeq \widehat{a} \wedge \widehat{s} \overset{\widehat{b}}{\rightarrow}_{\Box} \widehat{r} \wedge \widehat{r} \models^{nec} \widehat{\varphi}$
3. Assume some $\widehat{r}, \widehat{a}$ and $\widehat{b}$ such that
   (a) $\widehat{a} \in \widehat{A}$
   (b) $\widehat{b} \preccurlyeq \widehat{a}$
   (c) $\widehat{s} \overset{\widehat{b}}{\rightarrow}_{\Box} \widehat{r}$
   (d) $\widehat{r} \models^{nec} \widehat{\varphi}$
4. By Induction Hypothesis, we know that for all r and $\widehat{r}$ such that $r \in \gamma_S(\widehat{r})$, $\widehat{r} \models^{nec} \widehat{\varphi} \implies r \models^{nec}_{G_A} \widehat{\varphi}$
5. By definition of minimal restricted abstraction and (3.c), exists a pair (R_{min}, B_{min}) in $\widehat{\mathcal{M}}^{min}_{(\widehat{s},\Box)}$ such that:
   (a) $\widehat{r} = \alpha_S(R_{min})$
   (b) $\widehat{b} = \alpha_A(B_{min})$
6. By definition, (R_{min}, B_{min}) in $\widehat{\mathcal{M}}^{min}_{(\widehat{s},\Box)}$ implies (R_{min}, B_{min}) in $\widehat{\mathcal{M}}_{(\widehat{s},\Box)}$, hence:
   (a) $\forall\, s \in \gamma_S(\,\widehat{s}\,),$ implies $\exists\, r \in R_{min},\, a \in B_{min}.s \xrightarrow{a}_{\Box} r$
7. Assume r and a, such that:
   (a) $r \in R_{min}$
   (b) $a \in B_{min}$
   (c) $s \xrightarrow{a}_{\Box} r$
8. Then,
   (a) By monotonicity of α_S and (7.a), $\alpha_S(\{r\}) \preccurlyeq \alpha_S(R_{min})$
   (b) by (5.a), $\alpha_S(\{r\}) \preccurlyeq \widehat{r}$, so

   (c) by monotonicity of γ_S, $\gamma_S(\alpha_S(\{r\})) \subseteq \gamma_S(\widehat{r})$, so
   (d) by the properties of the Galois Connection, $s \in \gamma_S(\widehat{r})$, therefore
   (e) by I.H. and (3.d), $r \models^{nec}_{G_A} \widehat{\varphi}$

9. And,
   (a) By monotonicity of α_A and (7.b), $\alpha_A(\{a\}) \preccurlyeq \alpha_A(B_{min})$
   (b) by (5.b), $\alpha_A(\{a\}) \preccurlyeq \widehat{b}$, so
   (c) by transitivity and (3.b), $\alpha_A(\{a\}) \preccurlyeq \widehat{a}$, so
   (d) by monotonicity of γ_A, $\gamma_A(\alpha_S(\{a\})) \subseteq \gamma_A(\widehat{a})$, then
   (e) by the properties of the Galois Connection, $a \in \gamma_A(\widehat{a})$

10. Assuming $\widehat{s} \models^{nec} \langle\widehat{A}\rangle\widehat{\varphi}$, by (3.a), (7.c), (8.e) and (9.e) we have proved that $\exists\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge a \in \gamma_A(\widehat{a}) \wedge s \xrightarrow{a}_{\Box} r \wedge r \models^{nec}_{G_A} \widehat{\varphi}$ or what is the same $s \models^{nec}_{G_A} \langle\widehat{A}\rangle\widehat{\varphi}$ which proves the case.

□ (Case)

Case: *Necessary Universal*

1. We have to prove $\widehat{s} \models^{nec} [\widehat{A}]\widehat{\varphi} \implies s \models^{nec}_{G_A} [\widehat{A}]\widehat{\varphi}$
2. $s \models^{nec}_{G_A} [\widehat{A}]\widehat{\varphi}$ implies $\forall\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge \widehat{a} \preccurlyeq \alpha_A(\{a\}) \wedge s \xrightarrow{a}_{\Diamond} r \implies r \models^{nec}_{G_A} \widehat{\varphi}$
3. By Induction Hypothesis, we know that for all r and $\widehat{r}$ such that $r \in \gamma_S(\widehat{r})$, $\widehat{r} \models^{nec} \widehat{\varphi} \implies r \models^{nec}_{G_A} \widehat{\varphi}$
4. Let us choose some $r, \widehat{a}$ and a such that:
   (a) $\widehat{a} \in \widehat{A}$
   (b) $\widehat{a} \preccurlyeq \alpha_A(\{a\})$
   (c) $s \xrightarrow{a}_{\Diamond} r$

   then we have to prove $r \models^{nec}_{G_A} \widehat{\varphi}$
5. By definition of restriction, (3.b) and $s \in \gamma_S(\widehat{s})$ follows that $(\{r\}, \{a\}) \in \widehat{\mathcal{M}}_{(\widehat{s},\Diamond)}$
6. $\nexists (R, B) \in \widehat{\mathcal{M}}_{(\widehat{s},\Diamond)}$ with $R \subset \{r\}$ or $B \subset \{a\}$, hence $(\{r\}, \{a\}) \in \widehat{\mathcal{M}}^{min}_{(\widehat{s},\Diamond)}$, then
7. $\widehat{s} \xrightarrow{\alpha_A(\{a\})}_{\Diamond} \alpha_S(\{r\})$
8. $\widehat{s} \models^{nec} [\widehat{A}]\widehat{\varphi} \implies \forall\, \widehat{r}, \widehat{a}, \widehat{b}.\, \widehat{a} \in \widehat{A} \wedge \widehat{a} \preccurlyeq \widehat{b} \wedge \widehat{s} \xrightarrow{\widehat{b}}_{\Diamond} \widehat{r} \implies \widehat{r} \models^{nec} \widehat{\varphi}$
9. By (4.a), (4.b), (7) and (8), we know that $\alpha_S(\{r\}) \models^{nec} \widehat{\varphi}$

10. By definition of Galois Connection, $r \in \gamma_S(\alpha_S(\{r\}))$
11. Then, by I.H (with $\widehat{r} = \alpha_S(\{r\})$), (9) and (10), follows $r \models^{nec}_{G_A} \widehat{\varphi}$, therefore
12. if $\widehat{s} \models^{nec} [\widehat{A}]\widehat{\varphi} \implies \forall\, r, \widehat{a}, a.\, \widehat{a} \in \widehat{A} \wedge \widehat{a} \preccurlyeq \alpha_A(\{a\}) \wedge s \xrightarrow{a}_{\diamond} r \implies r \models_{G_A} \widehat{\varphi}$ which proves the case.

□ (Case)

The proofs of the *possible* universal and existential case are dual to the ones presented, so there is no value to include them. In order to prove the fixpoint operators:

- $\widehat{s} \in [\![\mu Y.\widehat{\varphi}]\!]^{nec}_{\widehat{\rho}} \implies s \in [\![\mu Y.\widehat{\varphi}]\!]^{nec}_{\rho,G_A}$
- $\widehat{s} \notin [\![\mu Y.\widehat{\varphi}]\!]^{pos}_{\widehat{\rho}} \implies s \notin [\![\mu Y.\widehat{\varphi}]\!]^{pos}_{\rho,G_A}$

We are going to define the operators in terms of approximants. So the minimal fixpoint will be (the maximal fixpoint can be defined by the negation of the minimal):

$$\begin{aligned} [\![\mu Y^{<\lambda}.\widehat{\varphi}]\!]_{\rho,G_A} &= \textstyle\bigcup_{\beta<\lambda} [\![\mu Y^{\beta}.\widehat{\varphi}]\!]_{\rho,G_A} \\ [\![\mu Y^{\lambda}.\widehat{\varphi}]\!]_{\rho,G_A} &= [\![\widehat{\varphi}]\!]_{(\rho\oslash[[\![\mu Y^{<\lambda}.\widehat{\varphi}]\!]_{\rho,G_A}/Y]),G_A} \end{aligned}$$

We know that for countable domains and monotonic functions, there is an ordinal λ such that $[\![\mu Y.\widehat{\varphi}]\!]_{\rho,G_A} = [\![\mu Y^{\lambda}.\varphi]\!]_{\rho,G_A}$. These definitions are true for both possible and necessary semantics. Therefore, we can use transfinite induction, exactly as we did for Theorem 1.4.5, to conclude the proofs.

□ (Theorem)

Proof: (Theorem 1.4.7) The proof of this Theorem has no difficulties, it is enough to adapt the previous proof.

□ (Theorem)

Chapter 2

Modal Abstractions of Processes

This chapter describes a framework to generate modal abstract approximations from process algebraic specifications, written in μCRL. We introduce a new format for process specification called *Modal Linear Process Equation* (MLPE). An MLPE represents all the possible interleavings of the parallel composition of a number of processes. Every transition step may lead to a set of abstract states labelled with a set of abstract actions. We use MLPEs to characterise abstractions of systems and to generate *Modal Labelled Transition Systems.*

2.1 Introduction

A preliminary step to abstract process algebraic specifications was already taken in [42]; those authors show how process algebras [72, 94, 93, 20, 8, 4] can benefit from abstract interpretation *in principle*. To this end they work with a basic LOTOS [38] language and a simple temporal logic; their abstractions preserve linear-time safety properties only. Here, we enhance previous work on abstraction for process algebra by adapting the results presented in the previous chapter.

We introduce the theory for applying abstract interpretation techniques to μCRL specifications, which (as in LOTOS) consist of an ADT part defining data operations, and a process specification part, specifying an event-based reactive system. Processes are defined using a.o. sequential and parallel composition, non-deterministic choice and hiding. Furthermore, atomic actions, conditions and recursion are present, and may depend on data parameters. The μCRL toolset transforms specifications to *linear process equations* (LPE), by eliminating parallel composition and hiding efficiently.

We implement abstract interpretation as a transformation from LPEs to *Modal*-LPEs (MLPEs). MLPEs capture the extra non-determinism arising from abstract interpretation. They allow a single transition to lead to a *set* of states with a *set* of action labels. We show that the MLTS generated from an MLPE is a proper abstraction of the LTS generated from the original LPE. By Theorem 1.4.6, this implies soundness for μ-calculus properties. Section 2.4 is devoted to this issue.

The next figure shows the different possibilities to extract abstract approximations from concrete specifications.

$$\begin{tikzcd}
\text{concrete spec (LPE)} \arrow[r, "(1)"] \arrow[d, "(4)"'] \arrow[dr, "(3)"] & \text{concrete system (LTS)} \arrow[d, "(2)"] \\
\text{abstract spec (MLPE)} \arrow[r, "(5)"] & \text{abstract system (MLTS)}
\end{tikzcd}$$

From a concrete system, encoded as an LPE, we can:

- Generate the concrete transition system (1), from which we compute the abstraction (2). Even though the resulting abstraction is optimal, this option is not very useful for verification because the generation of the concrete transition system may be impossible (or too expensive) due to the size of the state space.

- Generate directly the abstract *Modal*-LTS (3), by interpreting the concrete specification over the abstract domain. This solution avoids the generation of the concrete transition system.

- First, generate a symbolic abstraction of the concrete system (4), and then extract the abstract transition system (5).

Typically, standard abstract interpretation frameworks implement the second approach (arrow (3) of the figure), however we believe that the third (arrow (4) followed by (5)) one is more modular. *Modal*-LPEs act as intermediate representation that may be subjected to new transformations. There exist several tools (see [13, 15]) that manipulate linear equations that do, for example, symbolic model checking, state space reduction, elimination of dead code, confluence analysis, ... Furthermore, different techniques for algebraic verification [62] are based on linear processes [63, 48].

As we will show in the next chapter, MLPEs can be transformed back to LPEs. Thus our method integrates perfectly with the existing transformation and state space generation tools of the μCRL toolset [12, 13]. Also, the three valued model checking problem can be rephrased as the usual model checking problem, along the lines of [55]. This enables the reuse of the model checkers in the CADP toolset [43, 52].

The chapter is organised as follows; first we introduce the syntax and semantics of μCRL. Then we present the basic concepts about abstract interpretation of data types and the transformation of concrete μCRL specifications to abstract ones. We conclude by illustrating the use of the theory in a case study. The next chapter describes a tool that implements the theory and facilitates the application of the technique to realistic specifications.

2.2 μCRL in a Nutshell

μCRL [57] is a combination of process algebra [8, 45, 9] and abstract data types. Data is represented by an *algebraic specification* $\Omega = (\Sigma, E)$, in which Σ denotes a many-sorted *signature* (S, F), see [64]. Formally:

- A many-sorted *signature* $\Sigma = (S, F)$, where S is a set of types or sort names, and F a set of function symbols. A function f of some sort S, and with arity n, is typed by $f : S_0 \times \cdots \times S_{n-1} \rightarrow S_n$, where $S_0, \ldots, S_{n-1}$ are the sorts of the arguments of f, and S_n the sort of f.

- A set E of Σ-*equations*, which are expressions of the form $s = t$ where s and t are equally typed terms constructed from variables and function symbols in the usual way.

From process algebra μCRL inherits the following operators:

- $p.q$ performs p and then performs q;

- $p + q$ performs arbitrarily either p or q;

- $\sum_{d:D} p(d)$ performs $p(d)$ with an arbitrarily chosen d of sort D;

- $p \ \lhd \ b \ \rhd \ q$ if b is true, performs p, otherwise performs q;

- $p \,||\, q$ runs processes p and q in parallel.
- τ represents the silent step.
- δ stands for deadlock.

Atomic actions may have data parameters. The operator $|$ allows synchronous parameterised communication. If two actions are able to synchronise we can force that they occur always in communication using the encapsulation operator (∂_H). The operator τ_I hides enclosed actions by renaming into τ actions. The initial behaviour of the system can be specified with the keyword **init** followed by a process term:

$\mathsf{System} = \tau_I \partial_H(p_0 \parallel p_1 \parallel ...)$
init System

The following μCRL process specifies a bounded buffer implemented using a list. The process can non-deterministically choose between executing a *write* or a *read* action. The *write* can only be performed if the buffer is not full, i.e., the length of the list that models the buffer is smaller that the maximal length (MAX). The *read* action can be performed if the buffer is not empty. In the first case, the state parameter is updated by concatenating a new bit to the list; in the second case, the first element of the list is removed.

$$\begin{aligned} \mathit{Buffer}(l : \mathit{List}) = \sum_{b:Bit} & \mathit{write}(b).\mathit{Buffer}(\mathit{cons}(b,l)) \lhd \mathit{lt}(\mathit{len}(l), MAX) \rhd \delta + \\ & \mathit{read}(\mathit{head}(l)).\mathit{Buffer}(\mathit{tail}(l)) \lhd \mathit{not}(\mathit{isEmpty}(l)) \rhd \delta \end{aligned}$$

All specifications must include the boolean sort *Bool* with the constants true and false (T and F), because it is needed to define the conditional choice. We assume also the existence of the sorts naturals (with the standard operations equality *eq*, successor *succ*, predecessor *pred*, ...), and bits. The sort *List* can be defined as follows:

sort	$List, Bool, Nat$	**var**	$l : List$
func	$emptyList :\to List$		$b : Bit$
	$cons : Bit \times List \to List$	**rew**	$head(cons(b,l)) = b$
	$head : List \to Bit$		$tail(cons(b,l)) = l$
	$tail : List \to List$		$len(emptyList) = 0$
	$len : List \to Nat$		$len(cons(b,l)) = succ(len(l))$
	$isEmpty : List \to Bool$		$isEmpty(l) = eq(0, len(l))$

In the specification of the data type we have used the following keywords: **sort** to define the name of the data type, **func** to define the signature of the operations, **var** to define the auxiliary and **rew** to define the defining equations of the type.

μCRL, in combination with other tools has been use for verification purposes in different works. The list of cases of study is long, we just point to a few references. For example [11, 46] present the correctness of the sliding window protocol, [58] describes a mutual exclusion algorithm, in [99] some errors in a Java cache coherence coherence protocol were found and corrected. Furthermore, [49] and [86] analyse respectively the Itai-Rodeh leader election and the link layer of the 1394 protocol. [47] illustrates an in-flight data acquisition unit and [74, 106] analyse two different shared data spaces architectures, the latter is presented in Chapter 6.

2.2.1 Linear Process Equations

Every μCRL system can be transformed to a special format, called *Linear Process Equation* or *Operator* [61, 117]. An LPE (see definition below) is a single μCRL process which represents the complete system and from which parallel composition, encapsulation and hiding have been eliminated.

$$X(d : D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i[d, e_i]).X(g_i[d, e_i]) \lhd c_i[d, e_i] \rhd \delta \tag{2.1}$$

In the definition, d denotes a vector of parameters d of type D that represents the state of the system at every moment. We use the keyword *init* to declare the initial vector of values of d. Action labels a_i are selected from a set of action names *ActNames*[1]. The process is composed by a finite number I of summands, every summand i, has a list of local variables e_i, of possibly infinite domains, and it is of the following form: a condition $c_i[d, e_i]$, if the evaluation of the condition is true the process executes the action a_i with the parameter $f_i[d, e_i]$ and will move to a new state $g_i[d, e_i]$, which is a vector of terms of type D. $f_i[d, e_i]$, $g_i[d, e_i]$ and $c_i[d, e_i]$ are terms built recursively over variables $x \in [d, e_i]$, applications of function over terms $t = f(t')$ and vectors of terms. For example, we compose two *buffers* in parallel, as follows:

comm $read_0|write_1 = w$
System $= \tau_{\{w\}}\partial_{\{read_0, write_1\}}(Buffer_0(emptyList) \parallel Buffer_1(emptyList)$
init System

where $Buffer_0$ is equal to the process *Buffer* in which l is renamed to $l0$, *write* to $write_0$ and *read* to $read_0$ (similar for $Buffer_1$). We obtain the following linear form

[1]From now on we will use *ActNames* to refer just to the labels and *Act* to the set of action label together with the arguments

$$\begin{aligned} X(l0, l1 : List) = \sum_{b:Bit} & write_0(b).X(cons(b, l0), l1) \lhd lt(len(l0), MAX) \rhd \delta + \\ & \tau.X(tail(l0), cons(head(l0), l1)) \\ & \qquad\qquad \lhd not(isEmpty(l0)) \land lt(len(l1), MAX) \rhd \delta + \\ & read_1(head(l1)).X(l0, tail(l1)) \lhd not(isEmpty(l1)) \rhd \delta \end{aligned}$$

To every LPE specification corresponds a labelled transition system. The semantics of the system described by an LPE are given by the following rules:

- $s_0 = \text{init}_{lpe}$
- $s \xrightarrow{a} s'$ if and only if exists $i \in I$ and exists $e : E_i$ such that $c_i[s, e] = \mathsf{T}, a_i(f_i[s, e]) = a$ and $g_i[s, e] = s'$

Data terms are interpreted over the universe of values $\mathcal{D}$. The LTS corresponding to the *Buffer* LPE can be generated for any finite value of the constant *MAX*. The *Buffer* is modelled to contain bits which makes it finite. If we change the specification to have a container of natural numbers then the system will have an infinitely branching behaviour as the one presented in the previous chapter.

2.3 Data Abstraction

2.3.1 Abstraction of Sorts

Abstractions from μCRL specifications are generated by interpreting the data terms over an abstract domain that is in general smaller than the concrete one. Therefore, to produce an abstraction one should, first, define an abstract data domain and a relation with the concrete. As we have seen in the previous chapter, the relation can be expressed as a mapping $H : \mathcal{D} \to \widehat{\mathcal{D}}$ or a Galois Connection ($\alpha : \mathcal{P}(\mathcal{D}) \to \widehat{\mathcal{D}}, \gamma : \widehat{\mathcal{D}} \to \mathcal{P}(\mathcal{D})$). For example, an abstract domain for lists may be:

```
sort  abs_List                   rew  H(emptyList) = empty
func  empty, middle,                  H(cons(b, l)) =
          full :→ abs_List                if(MAX ≥ len(cons(b, l)),
      H : List → abs_List                    full, middle)
var   l : List
      b : Bit
```

We could write a similar data specification for the Galois Connection case using the extra values *nonEmpty, nonFull*, $\top$ and $\bot$. In order to obtain consistent abstract specifications (which generate transition systems without meaningless transitions), we impose the following restrictions on the Galois Connections:

- $\alpha(\emptyset) = \bot$ and $\neq \bot$ for any other value S.
- $\gamma(\bot) = \emptyset$ and $\neq \emptyset$ for any other value $\widehat{s}$.

In some cases, it would be interesting to define a Galois Connection from an homomorphism. In general, it is more intuitive to think in terms of mappings than of Connections. In practise the lifting of homomorphisms will save some effort during the task of defining abstractions. To define a Galois Connection from an homomorphism H we proceed as follows:

If we have the concrete domain D and the abstract $\widehat{D}$, then we build the abstract lattice as the power set of abstract values $\mathcal{P}(\widehat{D}\,)$ ordered by the set inclusion operator. Furthermore, we define $\mathcal{A} : \mathcal{P}(D) \rightarrow \mathcal{P}(\widehat{D})$ and $\mathcal{G} : \mathcal{P}(\widehat{D}) \rightarrow \mathcal{P}(D)$ as:

- $\mathcal{A}(S) = \{H(s) \mid s \in S\}$
- $\mathcal{G}(\widehat{S}\,) = \{s \mid \exists\, \widehat{s} \in \widehat{S} \wedge H(s) = \widehat{s}\}$

Note that not all Galois Connections can be defined from homomorphisms. Figure 2.1 displays the abstract lattice of lists.

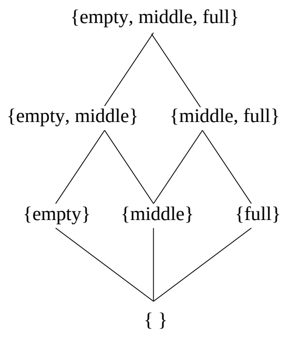


Figure 2.1: Lifted abstract lists

Lemma 2.3.1 The pair $(\mathcal{A}, \mathcal{G})$ built from a lifted homomorphism H forms a Galois Connection.[2]

The idea is to integrate the three ways of relating concrete and abstract domains. We have seen that on the one hand we may have plain homomorphisms, that can be lifted to Galois Connections by considering the abstract domain consisting of sets of abstract values, and on the other hand we may have plain Galois Connections in which the abstract domains are simple abstract values. To treat homogeneously these approaches, we are going to define a different

[2]Proofs are at the end of the chapter.

form of the latter case with the same signature as the lifted homomorphism case.

We define a new Galois Connection $(\mathcal{A}, \mathcal{G})$ from $(\mathcal{P}(D), \subseteq)$ to $(\mathcal{P}(\widehat{D}), \dot{\preccurlyeq})$, based on (α, γ) from $(\mathcal{P}(D), \subseteq)$ to $(\widehat{D}, \preccurlyeq)$.

- $\mathcal{A}(S) = \{\alpha(\{s\}) \mid s \in S\}$
- $\mathcal{G}(\widehat{S}) = \cup\{\gamma(\widehat{s}) \mid \widehat{s} \in \widehat{S}\}$
- $\widehat{S} \dot{\preccurlyeq} \widehat{S}'$ if and only if $\forall \widehat{s} \in \widehat{S}\, \exists \widehat{s'} \in \widehat{S}'. \widehat{s} \preccurlyeq \widehat{s'}$

Lemma 2.3.2 $(\mathcal{A}, \mathcal{G})$ built from (α, γ) as defined above form a Galois Connection.

From now on, we will use $(\mathcal{A}, \mathcal{G})$ with the order $\dot{\preccurlyeq}$ (which in the case of the homomorphisms will be equivalent to $\subseteq$).

2.3.2 Abstraction of Data Terms

Formal specifications are composed by process and data terms. In the previous section we have presented the possibilities to relate abstract and concrete values. Now, we describe how to define the abstraction of data terms.

Formally, we start by introducing an abstract operator to relate the concrete data specification and the abstract version: "$\widehat{\ }$": $\Sigma \to \widehat{\Sigma}$. In order to maintain the semantics of the process condition, we apply the restriction that booleans are not abstracted: $\widehat{Bool} = Bool$.

We are going to overload the syntactic operator "$\widehat{\ }$" to denote also the abstraction of data terms. Data terms are recursively built over constants (unary functions), variables from a set X, tuples, and function symbols. First, we introduce a new set $\widehat{X}$ of variables of type $\widehat{D}$. Then we define abstraction of data terms being a function from concrete terms built over concrete values to abstract terms built over sets of abstract values: $\widehat{\ }$: $\mathcal{T}_{\overrightarrow{D}}(\Sigma, X) \to \mathcal{T}_{\overrightarrow{\mathcal{P}(\widehat{D})}}(\widehat{\Sigma}, \widehat{X})$ (we explain below why we use sets of values).

We try to keep the definition of "$\widehat{\ }$" as general as possible, so we do not fully define it on terms. We only define the following two cases:

- The abstraction of tuples, as: $[\widehat{t_0, ..., t_n}] = [\widehat{t_0}, ..., \widehat{t_n}]$
- The abstraction of constants, as: $\widehat{c} = \mathcal{A}(\{c\})$[3]

In order to illustrate how function symbols are going to be abstracted, we present the following example: in our buffer specification we have a function *cons* which adds an element to a list. The abstract version of *cons* may be defined as follows:

[3]From now on, we simplify the notation by using $\mathcal{A}(c)$ instead of $\mathcal{A}(\{c\})$, and $\mathcal{G}(\widehat{c})$ instead of $\mathcal{G}(\{\widehat{c}\})$

- For the homomorphism:
  - $abs\_cons(b, empty) = middle$
  - $abs\_cons(b, middle) = middle$ or $abs\_cons(b, middle) = full$
  - $abs\_cons(b, full) = full$
- For the Galois Connection approach:
  - $abs\_cons(b, empty) = middle$
  - $abs\_cons(b, middle) = nonEmpty$
  - $abs\_cons(b, nonFull) = nonEmpty$
  - $abs\_cons(b, nonEmpty) = nonEmpty$
  - $abs\_cons(b, full) = \top$
  - $abs\_cons(b, \bot) = \bot$
  - $abs\_cons(b, \top) = \top$

We see in the example that abstract interpretation of functions may add non-determinism to the system, for instance $abs\_cons(b, middle)$ in the homomorphism case may return different values ($middle$ and $full$). The reason why we have defined abstract terms over sets of abstract values is to deal with this fact.

In the example, the signature of the concrete successor is $cons : Bit \times List \rightarrow List$, the abstraction will be $abs\_cons : Bit \times \widehat{List} \rightarrow \mathcal{P}(\widehat{List})$ (the sort Bit is left concrete). Then, for the homomorphism $abs\_cons(b, middle) = \{middle, full\}$ and for the Galois Connection: $abs\_cons(b, empty) = \{nonEmpty\}$

We have said that booleans are not abstracted. Else for instance a function $isEmpty : List \rightarrow Bool$ could be abstracted to the function $abs\_isEmpty : \widehat{List} \rightarrow \mathcal{P}(Bool)$. Then *abs_isEmpty(nonFull)* would be equal to $\{\mathsf{T}, \mathsf{F}\}$ and *abs_isEmpty(empty)* $= \{\mathsf{T}\}$. We could have defined the abstract type of booleans with $\{\bot, \widehat{\mathsf{T}}, \widehat{\mathsf{F}}, \top\}$ being $abs\_isEmpty(nonFull)$ equal to $\top$, but we wanted to avoid the redefinition of the semantics of the conditions of the processes for the abstract values.

Formally, if we would use only simple Galois Connections we would not need terms over sets of values in the abstract specification. However, keeping the sets gives more flexibility because it allows to easily combine the different types of abstractions and to reuse concrete sorts in the abstract specification (as for example the booleans of the sort *abs_isEmpty* or *Bit* in the signature of *abs_cons*). Below we present a possible definition of the abstraction of *List*.

Note how the abstraction removes the value of the entries stored in the list, therefore the *head* functions returns the collection of all possible values of bits. The true length of the list is also abstracted, when the list is equal to *middle* we do not know the exact number of bits stored, this is represented by returning a set of all possible lengths. We could have chosen to specify $abs\_cons(b, full)$ to be equal to $\{\bot\}$ (instead of $\{full\}$) denoting an execution error.

sort	abs_List
func	$empty, middle, full :\to abs\_List$
	$abs\_cons : Bit \times abs\_List \to \mathcal{P}(abs\_List)$
	$abs\_head : abs\_List \to \mathcal{P}(Bit)$
	$abs\_tail : abs\_List \to \mathcal{P}(abs\_List)$
	$abs\_len : abs\_List \to \mathcal{P}(Nat)$
	$abs\_isEmpty : abs\_List \to \mathcal{P}(Bool)$
var	$abs\_l : abs\_List$
	$b : Bit$
rew	$abs\_cons(b, empty) = \{middle\}$
	$abs\_cons(b, middle) = \{middle, full\}$
	$abs\_cons(b, full) = \{full\}$
	$abs\_head(abs\_l) = \{b0, b1\}$
	$abs\_tail(empty) = \{empty\}$
	$abs\_tail(middle) = \{empty, middle\}$
	$abs\_tail(full) = \{middle\}$
	$abs\_len(empty) = \{0\}$
	$abs\_len(middle) = \{1...pred(MAX)\}$
	$abs\_len(full) = \{MAX\}$
	$abs\_isEmpty(empty) = \{\mathsf{T}\}$
	$abs\_isEmpty(middle) = \{\mathsf{F}\}$
	$abs\_isEmpty(full) = \{\mathsf{F}\}$

In summary, the abstraction of the data specification will consist of a new data specification, whose values are related with the concrete by means of a homomorphism or a Galois Connection, and data terms built from the data type represent sets of values. Functions and equations of the concrete data type have to be defined over the abstract domain considering the use of sets of values.

2.3.3 Safety Condition

In the previous section we have given the guidelines to define an abstraction operator for data terms. We remark that not all possible abstract interpretations are correct; in order to generate *safe* abstractions the data terms involved in the specification and their abstract versions have to satisfy a formal requirement, usually called *safety condition.* A pair $(t, \widehat{t})$, where t is concrete term that applies point-wisely to sets and $\widehat{t}$ an abstract one, satisfy the safety condition if and only if, for all abstract values $\widehat{d}$:

- $\mathcal{A}(t[\mathcal{G}(\widehat{d})]) \mathrel{\dot{\preccurlyeq}} \widehat{t}\,[\,\widehat{d}\,]$

For consistency, we also require that the evaluation of abstract terms $\widehat{t}[\widehat{d}]$ built over values $\widehat{d}$ different from $\perp$ is not equal to $\{\perp\}$.

2.4 Modal Linear Process Equation

We present now a new format, the *Modal Linear Process Equation* that will be used to represent the symbolic abstraction of an LPE. An MLPE has the following form:

$$X(d:\mathcal{P}(\widehat{D})) = \sum_{i\in I}\sum_{e_i:E_i} a_i(F_i[d,e_i]).X(G_i[d,e_i]) \lhd C_i[d,e_i] \rhd \delta \tag{2.2}$$

The definition is similar to the one of *Linear Process Equation*, the difference is that the state is represented by a list of power sets of abstract values and for every i: C_i returns a non-empty set of booleans, G_i a non-empty set of states and F_i also a non-empty set of action parameters. Actions are parameterised with sets of values, as well. From an MLPE we can generate a *Modal Labelled Transition System* following these semantic rules:

- $S_0 = \text{init}_{mlpe}$
- $S \xrightarrow{A}_{\Box} S'$ if and only if exists $i \in I$ and exists $e \in E_i$ $(e \neq \bot)$ such that $\mathsf{F} \notin C_i[S,e]$, $A = a_i(F_i[S,e])$ and $S' = G_i[S,e]$
- $S \xrightarrow{A}_{\Diamond} S'$ if exists $i \in I$ and exists $e \in E_i$ $(e \neq \bot)$ such that $\mathsf{T} \in C_i[S,e]$, and $A = a_i(F_i[S,e])$ and $S' = G_i[S,e]$

MLPEs allow to capture in a uniform way both approaches: Galois Connection and Homomorphism as well as the combination of both consisting of the lifting of a mapping to a Galois Connection. In case we use a plain homomorphism (without lifting it to a Galois Connection), we restrict the rules by letting S_0, S, A and S' be only singleton sets.

To compute an abstract interpretation of a linear process, we define the operator "$^-$": $LPE \to MLPE$ that pushes the abstraction through the process operators till the data part:

$p = X(t)$ then $\bar{p} = X(\widehat{t})$ with X being a process name
$p = a(t)$ then $\bar{p} = a(\widehat{t})$ with a being an action label
$p = p_0 + ... + p_n$ then $\bar{p} = \bar{p_0} + ... + \bar{p_n}$
$p = \delta$ then $\bar{p} = \delta$
$p = p_0.p_1$ then $\bar{p} = \bar{p_0}.\bar{p_1}$
$p = p_l \lhd t_c \rhd p_r$ then $\bar{p} = \bar{p_l} \lhd \widehat{t_c} \rhd \bar{p_r}$
$p = \sum_{e:E} p$ then $\bar{p} = \sum_{\widehat{e}:\widehat{E}} \bar{p}$

Furthermore, the initial value of the *mlpe* will be equal to the abstraction of the initial value of the *lpe*, i.e., $\text{init}_{mlpe} = \widehat{\text{init}_{lpe}}$. Note that actions are abstracted by abstracting the arguments of them, the action label is kept unchanged. Let us show the result of abstracting the buffer process:

$$\begin{aligned}
&abs\_Buffer(l : \mathcal{P}(abs\_List)) = \\
&\qquad \sum_{b:Bit} write(b).abs\_Buffer(\widehat{cons}(b, l)) \lhd lt(\widehat{len(l), MAX}) \rhd \delta + \\
&\qquad read(\widehat{head(l)}).abs\_Buffer(\widehat{tail(l)}) \lhd not(\widehat{isEmpty}(l)) \rhd \delta
\end{aligned}$$

In the next chapter we will discuss an implementation for the abstraction of the data terms. Just considering that the terms return sets of values, as defined in section 2.3.2, the MLPE form can be used equally for any kind of relation between the data domains: homomorphisms, arbitrary Galois Connections and lifted homomorphisms. The following lemma shows that abstractions do not contain meaningless transitions.

Lemma 2.4.1 For every transition of $\widehat{s} \overset{a(\widehat{t})}{\rightarrow} \widehat{r}$, the values $\widehat{s}, \widehat{r}$ and $\widehat{t}$ are not equal to $\{\perp\}$

Now, we present the main result of the chapter that states that an abstraction of a process, in which data terms satisfy the safety condition, generates an abstract approximation of the transition system generated by the concrete process.

Theorem 2.4.2 Given a Linear Process Equation *lpe*, a Modal Linear Process Equation $\bar{\text{lpe}}$ and a Galois Connection $(\mathcal{A}, \mathcal{G})$ from $(\mathcal{P}(D), \subseteq)$ to $(\mathcal{P}(\widehat{D}), \dot{\preccurlyeq})$ between their data domains (as defined on section 2.3.1). If:

1. $\bar{\text{lpe}}$ is the abstract interpretation of lpe,
2. lts is the LTS generated from lpe[4]
3. $\widehat{\text{mlts}\downarrow}$ the minimal *restricted* abstraction w.r.t $(\mathcal{A}, \mathcal{G})$ of lts
4. And the pairs of concrete and abstract data terms (all pairs (f, F), (g, G) and (c, C) of *lpe* and $\bar{\text{lpe}}$) satisfy the safety condition.

- Then, the MLTS ($\widehat{\text{mlts}}$) generated from *mlpe* is an abstraction of $\widehat{\text{mlts}\downarrow}$, i.e, $(\widehat{\text{mlts}\downarrow} \sqsubseteq_{\dot{\preccurlyeq}} \widehat{\text{mlts}})$

$$\begin{tikzcd}
\text{lpe} \arrow[r] \arrow[d] & & \text{lts} \arrow[d] \\
\bar{\text{lpe}} \arrow[r] & \widehat{\text{mlts}} \arrow[r, phantom, "\sqsupseteq_{\dot{\preccurlyeq}}"] & \widehat{\text{mlts}\downarrow}
\end{tikzcd}$$

[4]More precisely we should use the concrete MLTS equivalent to lts, see definition 1.2.5.

In (3), we say that $\widehat{\text{mlts}\downarrow}$ is the minimal *restricted* abstraction with respect to $(\mathcal{A}, \mathcal{G})$ of mlts. By definition 1.2.5, to compute the minimal *restricted* abstraction, we need two Galois Connections (α_S, γ_S) and (α_A, γ_A) (for the states and for the actions), we will use the following:

- (α_S, γ_S) will be $(\mathcal{A}, \mathcal{G})$
- For (α_A, γ_A), we remember that on the abstraction of processes we only abstract the action arguments and not the action labels. Therefore, we are going to consider:
  - $\alpha_A(a(t)) = \{a(\widehat{t}) \mid \widehat{t} \in \mathcal{A}(t)\}$
  - $\gamma_A(\widehat{a(t)}) = \{a(t) \mid t \in \mathcal{G}(\widehat{t})\}$
  - $\widehat{a(t)} \dot{\preccurlyeq} \widehat{a(t'))}$ if and only if $\widehat{t} \dot{\preccurlyeq} \widehat{t'}$

The proof is done by checking that every *may* transition generated by the abstract *Modal Linear Process Equation* has at least one precise counterpart in the minimal *restricted* abstraction of the concrete system (and the other way around for the *must* transitions). The proof is included at the end of the chapter. By theorems 1.4.5, 1.4.6 and 1.4.7, we can prove (refute) properties for *lpe* by considering *mlpe* directly.

In the example of the buffer, in order to assure that the abstraction is correct, we will have to prove the following safety conditions:

- $\mathcal{A}(cons(\mathcal{G}(\widehat{b}, \widehat{l}))) \dot{\preccurlyeq} abs\_cons(\widehat{b}, \widehat{l})$
- $\mathcal{A}(lt(len(\mathcal{G}(\widehat{l}))), MAX) \dot{\preccurlyeq} lt(abs\_len(\widehat{l}), MAX)$
- $\mathcal{A}(head(\mathcal{G}(\widehat{l}))) \dot{\preccurlyeq} abs\_head(\widehat{l})$
- $\mathcal{A}(tail(\mathcal{G}(\widehat{l}))) \dot{\preccurlyeq} abs\_tail(\widehat{l})$
- $\mathcal{A}(not(isEmpty((\mathcal{G}(\widehat{l})))) \dot{\preccurlyeq} not(abs\_isEmtpy(\widehat{l}))$

As illustration we include the proof of the first safety condition at the end of the chapter.

2.5 A Case Study: *The Bounded Retransmission Protocol*

The BRP is a simplified variant of a Philips' telecommunication protocol that allows to transfer large files across a lossy channel. Files are divided in packets and are transmitted by a sender through the channel. The receiver acknowledgements every delivered data packet. Both data and confirmation messages may be lost. The sender will attempt to retransmit each packet at most *MAX* times.

The protocol presents a number of parameters, such as the length of the lists, the maximum number of retransmissions and the contents of the data, that cause the state space of the system to be infinite and limit the application of automatic verification techniques such as model checking. We describe, here, the application of the abstract interpretation techniques to remove uninteresting information of this protocol in order to use model checking to verify it.

We base our solution on the μCRL model presented in the paper [56], in which Groote and v.d. Pol proved using algebraic methods that the model is branching bisimilar to the desired external behaviour also specified in μCRL. This proof requires a strong and creative human interaction in order to be accomplished. However by computing an abstraction of the original we can automatically model check some properties.

The figure below shows the different agents that participate in the system. The system contains a sender that gets a file which consists of a list of elements. It delivers the file frame by frame through a channel. The receiver sends an acknowledgement for each frame, when it receives a packet it delivers it to the external receiver client attaching a positive indication I_{fst}, I_{inc} or I_{ok}. The sender, after each frame, waits for the acknowledgements, if the confirmation message does not arrive, it retransmits the packet. If the transmission was successful, i.e., all the acknowledgements have arrived, then the sender informs the sending client with a positive indication. When the maximum number of retransmissions is exceeded, the transmission is cancelled and I_{nok} is sent to the exterior by both participants. If the last frame or its confirmation are lost the sender cannot know whether the receiver has received the complete list, therefore it sends *"I don't know"* to the sending client, I_{dk}.

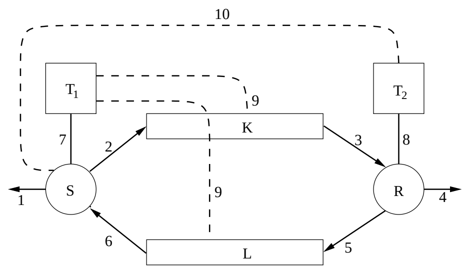


Figure 2.2: Overview of the bounded retransmission protocol

The protocol depends on the time behaviour, which is controlled by two timers T_1 and T_2. They determine when the messages are either delivered or lost, the retransmission of the packets and the timeout that makes the participants give up the transmission. The solution of [56] does not deal with the explicit time delays but with some non-deterministic signals (modelled by channels 9 and 10).

We are interested in proving that the external indications delivered by the sender and the receiver are "consistent". For that purpose, we chose an abstrac-

$$
\boxed{\begin{array}{ll}
\textbf{sort} & \mathit{abs\_List} \\
\textbf{func} & \mathit{empty} :\to \mathit{abs\_List} \\
 & \mathit{one} :\to \mathit{abs\_List} \\
 & \mathit{more} :\to \mathit{abs\_List} \\
 & H : \mathit{List} \to \mathit{abs\_List} \\
 & \mathit{eq} : \mathit{abs\_List} \times \mathit{abs\_List} \to \mathit{Bool} \\
 & \mathit{abs\_head} : \mathit{abs\_List} \to \mathcal{P}(\mathit{abs\_D}) \\
 & \mathit{abs\_tail} : \mathit{abs\_List} \to \mathcal{P}(\mathit{abs\_List}) \\
 & \mathit{abs\_last} : \mathit{abs\_List} \to \mathcal{P}(\mathit{Bool}) \\
 & \mathit{abs\_indl} : \mathit{abs\_List} \to \mathcal{P}(\mathit{Bit}) \\
\textbf{var} & \mathit{abs\_l} :\to \mathit{abs\_List} \\
 & l : \mathit{List} \\
\textbf{rew} & \mathit{abs\_head}(\widehat{l}) = \{d0, d1, d2\} \\
 & \mathit{abs\_tail}(\mathit{empty}) = \{\mathit{empty}\} \\
 & \mathit{abs\_tail}(\mathit{one}) = \{\mathit{empty}\} \\
 & \mathit{abs\_tail}(\mathit{more}) = \{\mathit{more}, \mathit{one}\} \\
 & \mathit{abs\_last}(\mathit{empty}) = \{\mathsf{T}\} \\
 & \mathit{abs\_last}(\mathit{one}) = \{\mathsf{T}\} \\
 & \mathit{abs\_last}(\mathit{more}) = \{\mathsf{F}\} \\
 & \mathit{abs\_indl}(\mathit{empty}) = \{\mathsf{e}_1\} \\
 & \mathit{abs\_indl}(\mathit{one}) = \{\mathsf{e}_1\} \\
 & \mathit{abs\_indl}(\mathit{more}) = \{\mathsf{e}_0\} \\
 & \mathit{eq}(\mathit{abs\_l}, \mathit{abs\_l}) = \mathsf{T} \\
 & \mathit{eq}(\mathit{empty}, \mathit{one}) = \mathsf{F} \\
 & \mathit{eq}(..., ...) = ... \\
 & H(\mathit{emptyList}) = \mathit{empty} \\
 & H(\mathit{cons}(d, \mathit{emptyList})) = \mathit{one} \\
 & H(\mathit{cons}(d, \mathit{cons}(d', l))) = \mathit{more}
\end{array}}
$$

tion that abstracts away the data stored in the file to transmit and maps the list to three critical values: *empty, one, more. empty* for when the list is empty, *one* when it has only one element, and *more* when it has more than one. We provide a new data specification which is shown in the μCRL code below.

The function *indl* gives different bits when either a list is at its end (it is empty or has only one element) or when there is more than one element, the rest of the functions are standard. The maximum number of retransmissions is abstracted away which makes the sender non-deterministically choose between resending a lost packet or giving up the transmission.

Once we obtain the abstract MLPE[5], we can use the state space generator of the μCRL toolset to obtain the abstract *Modal*-LTS. The result consists of 446 states and 1016 transitions, from which 448 are *must* transitions and the

[5]All the steps are done using the toolset implemented to abstract μCRL specifications that will be described in the next chapter.

rest *may*s.

The abstraction we have used allows to reason about the execution of the final part of the protocol without knowing the exact content of data files or the number of retrials. For example the following *safety* property: *"after a positive notification by the receiver, the sender cannot send a negative one"* is *necessarily* satisfied by the abstract system.

(C1): [true* . 'R(.*,$\mathsf{I_{ok}}$)' . ($\neg$ 'S(.*)')* . 'S($\mathsf{I_{nok}}$)'] F

We have expressed the property in the logic Regular Alternation-free μ-calculus [88, 89]. The logic embeds regular expressions with modal and fixpoint operators. There are three types of formulas, action (α), regular (β) and state formulas (φ), expressed by the following grammars:

$$\begin{aligned}
&\alpha ::= \mathsf{T} \mid \mathsf{F} \mid \neg\,\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid a(\bar{d}) \mid \mathit{reg} - \mathit{exp}\\
&\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1|\beta_2 \mid \beta* \mid \beta+\\
&\varphi ::= \mathsf{T} \mid \mathsf{F} \mid \neg\,\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\beta]\varphi \mid \langle\beta\rangle\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi
\end{aligned}$$

a stands for an action label from *ActNames*, and $\bar{d}$ for a, possibly empty, list of arguments. When the list is empty, we just write a. $a(\bar{d})$ matches transitions with the same action label and exactly the same arguments. T matches all actions with any argument, $\neg\,\alpha$ matches all actions but the ones matched by α. F matches no action, it could have been expressed by $\neg\,\mathsf{T}$. $\alpha_1 \wedge \alpha_2$ matches all action that match α_1 and α_2. $\alpha_1 \vee \alpha_2$ matches all action that match α_1 or α_2. Action formulas can be also expressed as regular expressions which match using the standard syntactic rules.

Regular formulas match sequences of actions; '.' stands for the concatenation operator, '|' is the choice operator, '$*$' is the transitive and reflexive closure operator, and '+' is the transitive closure operator.

The semantics of the state formulas is standard. $[\beta]\varphi$ states that all continuations by sequences matching β satisfy φ. $\langle\beta\rangle\varphi$ states that there exists at least one β sequence satisfying φ. μ and ν are the minimal and maximal fixpoint operators.

As we will see in the next chapter, in order to prove the properties we are going to use the CADP toolset, which only allows the use of alternation free formulas (formulas in which μ and ν do not alternate). This logic can be transformed into standard μ-calculus with the syntax given in definition 1.4 using the following equivalences:

$$\begin{aligned}
\beta+ &= \beta \cdot \beta* \\
\langle \beta 1 \cdot \beta 2 \rangle \varphi &= \langle \beta 1 \rangle \langle \beta 2 \rangle \varphi \\
\langle \beta 1 \mid \beta 2 \rangle \varphi &= \langle \beta 1 \rangle \varphi \vee \langle \beta 2 \rangle \varphi \\
\langle \beta * \rangle \varphi &= \mu X.(\varphi \vee \langle \beta \rangle) \\
[\beta 1 \cdot \beta 2]\varphi &= [\beta 1][\beta 2]\varphi \\
[\beta 1 \mid \beta 2]\varphi &= [\beta 1]\varphi \wedge [\beta 2]\varphi \\
[\beta *]\varphi &= \nu X.(\varphi \wedge [\beta])
\end{aligned}$$

The following *liveness* property expresses that: *"After a negative notification by the receiver, there exists a path which leads to a negative or* don't know *notification by the sender"*.

(C2): $[\ \mathsf{T}^*\ .\ \text{'}\mathsf{R}(.*,\mathsf{I}_{\mathsf{nok}})\text{'}\ ]\ \langle\ \mathsf{T}^*\ .\ (\text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'} \vee \text{'}\mathsf{S}(\mathsf{I}_{\mathsf{nok}})\text{'})\rangle\ \mathsf{T}$

The next property is stronger than the previous, instead of only requesting that there exists a path it states that the expected sender notification is inevitably achieved:

(C3): $[\ \mathsf{T}^*\ .\ \text{'}\mathsf{R}(.*,\mathsf{I}_{\mathsf{nok}})\text{'}\ ]\ \mu\ \mathrm{X}.\ (\langle\ \mathsf{T}\rangle\ \mathsf{T}\wedge\ [\ \neg(\text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'} \vee \text{'}\mathsf{S}(\mathsf{I}_{\mathsf{nok}})\text{'})\ ]\ \mathrm{X})$

These three properties are *necessarily* satisfied in the abstract system, therefore we can infer its satisfaction in the original one. However, the following property which states that *"after a positive notification by the receiver there exists a path which leads to a* don't know *notification by the sender"* is not satisfied in the abstract system. The reason is that we have abstracted away the maximum number of retransmissions, therefore if all the acknowledgements are lost the sender can retransmit the frames forever:

(C4): $[\ \mathsf{T}^*\ .\ \text{'}\mathsf{R}(.*,\mathsf{I}_{\mathsf{ok}})\text{'}\ ]\ \langle\ \mathsf{T}^*\ .\ \text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ \rangle\ \mathsf{T}$

C4 is not *necessarily* satisfied but is *possibly* satisfied on the abstract, therefore we cannot conclude anything on the concrete.

Other papers have verified some properties of the protocol using abstract interpretation, we refer among others to [87, 33]. The approach of Manna et al. is based on automatic predicate abstractions and is limited to the proof of invariants. Dams and Gerth propose a number of creative abstractions in order to prove the satisfaction of some safety properties about the sequentiality of the delivering of the frames.

2.6 Conclusion

Our approach to extract abstractions from system specifications differs from the classical one on the fact that instead of giving the abstract semantics of the original model we symbolically generate a new specification that captures the abstract behaviour. A μCRL specification is first transformed to an LPE. We have defined a function that abstracts LPEs The function preserves the structure of the specification and pushes the abstraction until the data.

The correctness of the abstraction of the system depends on the correctness of the abstraction of data. We have not described how exactly data terms are abstracted, we have just introduced the basic ideas of an operator to abstract data that uses sets to capture the non-determinism that abstract functions may introduce. The next chapter describes a possible implementation of the operator. The toolkit that is described in the next chapter facilitates the application of the theory to non-trivial applications, as the *bounded retransmission protocol* presented in the previous section.

The result of the abstractions are always *Modal*-LPEs. The abstract transformation of the concrete system to the MLPE format permits to apply other symbolic transformation techniques for processes and tools to the abstract systems [12].

2.7 Proofs

Proof (Lemma 2.3.1) We have to prove:

1. $\mathcal{A}$ and $\mathcal{G}$ are monotonic.
2. $\forall S : \mathcal{P}(D), S \subseteq \mathcal{G} \circ \mathcal{A}(S)$.
3. $\forall \widehat{S} : \mathcal{P}(\widehat{D}), \mathcal{A} \circ \mathcal{G}(\widehat{S}) \subseteq \widehat{S}$.

- $\mathcal{A}$ is monotonic.
  1. We have to prove, if $S \subseteq S'$ then $\mathcal{A}(S) \subseteq \mathcal{A}(S')$
  2. $s \in S$ implies $s \in S'$, therefore by definition of $\mathcal{A}$ follows that $H(s) \in \mathcal{A}(S)$ implies $H(s) \in \mathcal{A}(S')$. Hence,
  3. $\mathcal{A}(S) \subseteq \mathcal{A}(S')$, which proves the case.
- $\mathcal{G}$ is monotonic.
  1. We have to prove, if $\widehat{S} \subseteq \widehat{S}'$ then $\mathcal{G}(\widehat{S}) \subseteq \mathcal{G}(\widehat{S}')$
  2. Assuming $s \in \mathcal{G}(\widehat{S})$, implies, by definition, $H(s) \in \widehat{S}$
  3. $\widehat{S} \subseteq \widehat{S}'$ implies $H(s) \in \widehat{S}'$, so
  4. $s \in \mathcal{G}(\widehat{S}')$, which proves the case.
- $S \subseteq \mathcal{G} \circ \mathcal{A}(S)$
  1. Let s be in S, then $H(s) \in \mathcal{A}(S)$,
  2. therefore, $s \in \mathcal{G} \circ \mathcal{A}(S)$, which proves the case.

- $\mathcal{A} \circ \mathcal{G}(\widehat{S}) \subseteq \widehat{S}$
  1. If $\widehat{s} \in \mathcal{A} \circ \mathcal{G}(\widehat{S})$ then for some s, $H(s) \in \widehat{S}$ and $s \in \mathcal{G}(\widehat{S})$
  2. therefore, $\widehat{s} = H(s) \in \widehat{S}$ which proves the case.

□ (Lemma)

Proof (Lemma 2.3.2). We have to prove:

1. $\mathcal{A}$ and $\mathcal{G}$ are monotonic.
2. $\forall S : \mathcal{P}(D), S \subseteq \mathcal{G} \circ \mathcal{A}(S)$.
3. $\forall \widehat{S} : \mathcal{P}(\widehat{D}), \mathcal{A} \circ \mathcal{G}(\widehat{S}) \dot{\preccurlyeq} \widehat{S}$.

- $\mathcal{A}$ is monotonic.
  1. We have to prove, if $S \subseteq S'$ then $\mathcal{A}(S) \dot{\preccurlyeq} \mathcal{A}(S')$
  2. Let us assume $\widehat{s} \in \mathcal{A}(S)$, then $\exists\, s \in S.\alpha(\{s\}) = \widehat{s}$
  3. Assume $s \in S$ implies $s \in S'$. Then $\alpha(\{s\}) \in \mathcal{A}(S)$, therefore
  4. $\widehat{s} \in \mathcal{A}(S')$, which proves the case.
- $\mathcal{G}$ is monotonic.
  1. We have to prove, if $\widehat{S} \dot{\preccurlyeq} \widehat{S'}$ then $\mathcal{G}(\widehat{S}) \subseteq \mathcal{G}(\widehat{S'})$
  2. Let us assume $s \in \mathcal{G}(\widehat{S})$, then $\exists \widehat{s} \in \widehat{S} \wedge s \in \gamma(\widehat{s})$
  3. By definition of $\dot{\preccurlyeq}$ follows that if $\widehat{s} \in \widehat{S}$ then $\exists\, \widehat{s'} \in \widehat{S'}$ with $\widehat{s} \preccurlyeq \widehat{s'}$
  4. $\widehat{s} \preccurlyeq \widehat{s'}$ implies that $\gamma(\widehat{s}) \subseteq \gamma(\widehat{s'})$ then
  5. $s \in \gamma(\widehat{s})$ implies $s \in \gamma(\widehat{s'})$, therefore if $s \in \mathcal{G}(\widehat{S})$ then $s \in \mathcal{G}(\widehat{S'})$, which proves the case.
- $S \subseteq \mathcal{G} \circ \mathcal{A}(S)$
  1. Let us assume $s \in S$, then $\alpha(\{s\}) \in \mathcal{A}(S)$, so
  2. $\gamma(\alpha(\{s\})) \subseteq \mathcal{G}(\mathcal{A}(S))$
  3. By the properties of the Galois Connection (α, γ), $s \in \gamma(\alpha(\{s\}))$, then
  4. $s \in \mathcal{G} \circ \mathcal{A}(S)$, which proves the case.
- $\mathcal{A} \circ \mathcal{G}(\widehat{S}) \dot{\preccurlyeq} \widehat{S}$
  1. Assume $\widehat{s} \in \mathcal{A}(\mathcal{G}(\widehat{S}))$, then,$\exists\, s \in \mathcal{G}(\widehat{S})$ with $\widehat{s} = \alpha(\{s\})$
  2. Assume $s \in \mathcal{G}(\widehat{S})$ and $\widehat{s} = \alpha(\{s\})$, then $\exists\, \widehat{s'} \in \widehat{S}$ such that $s \in \gamma(\widehat{s'})$
  3. By the properties of the Galois Connection (α, γ), $\alpha(\{s\}) \preccurlyeq \alpha(\gamma(\widehat{s'}))$, then $\widehat{s} = \alpha(\{s\}) \preccurlyeq \widehat{s'}$

4. Therefore, $\forall\,\widehat{s} \in \mathcal{A}(\mathcal{G}(\widehat{S}))\,\exists\,\widehat{s'} \in \widehat{S}$ with $\widehat{s} \preccurlyeq \widehat{s'}$, which proves the case.

$\square$ (Lemma)

Proof (Lemma 2.4.1) We apply induction over sequences of transitions:

1. For the basic case $\text{init}_{mlpe} = \widehat{\text{init}}_{lpe}$, which is equal to $\mathcal{A}(\{\text{init}_{lpe}\})$, by consistency requirement on the abstraction function it is different from $\{\perp\}$
2. For the inductive case, assuming $\widehat{s} \neq \{\perp\}$ and $\widehat{s} \overset{a(\widehat{t})}{\rightarrow} \widehat{r}$ then, we have to prove:
   (a) $\widehat{r} \neq \{\perp\}$
   (b) $\widehat{t} \neq \{\perp\}$
3. $\widehat{s} \overset{a(\widehat{t})}{\rightarrow} \widehat{r}$ implies, exists i and $\widehat{e}$ such that:
   (a) $G_i(\widehat{s}, \widehat{e}) = \widehat{r}$
   (b) $F_i(\widehat{s}, \widehat{e}) = \widehat{t}$
4. $\widehat{e}$ is not equal to $\perp$ and $\widehat{s}$ is not equal to $\{\perp\}$, then the consistency requirement of the abstract data terms $\widehat{r}$ and $\widehat{t}$ are not equal to $\{\perp\}$

$\square$ (Lemma)

(Theorem 2.4.2)

Proof: (may part)

1. Let $\widehat{s} \overset{a(\widehat{t})}{\rightarrow}_{\diamond} \widehat{r}$ be a *may* transition of the minimal *restricted* abstraction $\widehat{\text{mlts}}\!\downarrow$.
2. Then, by definition of minimal *restricted* abstraction, there exists s, t, r, R, B such that:
   (a) $s \overset{a(t)}{\rightarrow} r$ in lts
   (b) $s \in \mathcal{G}(\widehat{s})$
   (c) $r \in R$
   (d) $t \in B$
   (e) $\widehat{t} = \mathcal{A}(B)$
   (f) $\widehat{r} = \mathcal{A}(R)$
   (g) (R, B) in $\text{lts}^{min}_{(\widehat{s},\diamond)}$
3. As R and B are singletons, we obtain:

   (a) $R = \{r\}$
   (b) $B = \{t\}$

4. lts is generated from lpe, therefore, we will have a summand in lpe that generates the transition $s \xrightarrow{a(t)}_{\diamond} r$, i.e., exists i and e such that:
   (a) $\mathsf{T} = c_i(s, e)$
   (b) $a(t) = a(f_i(s, e))$
   (c) $r = g_i(s, e)$

5. We have to mimic this step in $\bar{\text{lpe}}$, for which we only know that the safety conditions are met, so:
   (a) $\mathcal{A}(c_i(\mathcal{G}(\widehat{s}, \widehat{e}))) \mathrel{\dot{\preccurlyeq}} C_i(\widehat{s}, \widehat{e})$
   (b) $\mathcal{A}(f_i(\mathcal{G}(\widehat{s}, \widehat{e}))) \mathrel{\dot{\preccurlyeq}} F_i(\widehat{s}, \widehat{e})$
   (c) $\mathcal{A}(g_i(\mathcal{G}(\widehat{s}, \widehat{e}))) \mathrel{\dot{\preccurlyeq}} G_i(\widehat{s}, \widehat{e})$

6. Now, we define:
   (a) $\widehat{e} = \mathcal{A}(e)$
   (b) $\widehat{t'} = F_i(\widehat{s}, \widehat{e})$
   (c) $\widehat{r'} = G_i(\widehat{s}, \widehat{e})$

7. In order to prove $\widehat{\text{mlts}\!\downarrow} \sqsubseteq_{\dot{\preccurlyeq}} \widehat{\text{mlts}}$, it suffices to prove:
   (a) $\mathsf{T} \in C_i(\widehat{s}, \widehat{e})$
   (b) $\widehat{t} \mathrel{\dot{\preccurlyeq}} \widehat{t'}$
   (c) $\widehat{r} \mathrel{\dot{\preccurlyeq}} \widehat{r'}$

8. To prove (7.a) we proceed as follows:
   (a) Booleans are not abstracted therefore by safety condition (5.a) is reduced to $c_i(\mathcal{G}(\widehat{s}, \widehat{e})) \subseteq C_i(\widehat{s}, \widehat{e})$
   (b) By definition of Galois Connection, $e \in \mathcal{G}(\mathcal{A}(e))$, hence by (6.a) $e \in \mathcal{G}(\widehat{e})$, so
   (c) by (2.b), follows $(s, e) \in \mathcal{G}(\widehat{s}, \widehat{e})$[6], so
   (d) $\mathsf{T} = c_i(s, e) \in c_i(\mathcal{G}(\widehat{s}, \widehat{e}))$ by set-wise application of c_i, then
   (e) by (8.a) the case is proved.

9. To prove (7.b) we proceed as follows:
   (a) as above, $(s, e) \in \mathcal{G}(\widehat{s}, \widehat{e})$ and $f_i(s, e) \in F_i(\mathcal{G}(\widehat{s}, \widehat{e}))$,
   (b) by monotonicity of $\mathcal{A}$ follows $\mathcal{A}(f_i(s, e)) \mathrel{\dot{\preccurlyeq}} \mathcal{A}(F_i(\mathcal{G}(\widehat{s}, \widehat{e}))$

[6]Remember that the abstraction functions apply pointwisely to tuples.

   (c) then, by the safety condition (5.b), $\mathcal{A}(f_i(s,e)) \mathrel{\dot{\preccurlyeq}} F_i(\widehat{s},\widehat{e})$, so
   (d) The case is proved by (1.e), (3.b), (4.b) and (6.b).

10. (7.c) is proved as the previous case.

□ (may part)

Proof: (must part)

1. Let $\widehat{s} \overset{a(\widehat{t})}{\longrightarrow}_{\Box} \widehat{r}$ be a *must* transition in $\widehat{\text{mlts}}$, so there exists a summand i in $\bar{\text{lpe}}$ and a value $\widehat{e}$ such that:
   (a) $\mathsf{F} \notin C_i(\widehat{s},\widehat{e})$
   (b) $a(\widehat{t}) = a(F_i(\widehat{s},\widehat{e}))$
   (c) $\widehat{r} = G_i(\widehat{s},\widehat{e})$
2. By $\widehat{\text{mlts}\!\downarrow} \sqsubseteq_{\dot{\preccurlyeq}} \widehat{\text{mlts}}$, the transition $\widehat{s} \overset{a(\widehat{t})}{\longrightarrow}_{\Box} \widehat{r}$ has to reflect a transition in the the minimal *restricted* abstraction, so there have to exist $\widehat{r'}, \widehat{t'}$ such that:
   (a) $\widehat{s} \overset{a(\widehat{t'})}{\longrightarrow}_{\Box} \widehat{r'}$ is in $\widehat{\text{mlts}\!\downarrow}$
   (b) $\widehat{t'} \mathrel{\dot{\preccurlyeq}} \widehat{t}$
   (c) $\widehat{r'} \mathrel{\dot{\preccurlyeq}} \widehat{r}$
3. In other words, we need to prove that exists (R_m, B_m) in $\text{lts}^{min}_{(\widehat{s},\Box)}$ such that:
   (a) $\widehat{s} \overset{a(\mathcal{A}(B_m))}{\longrightarrow}_{\Box} \mathcal{A}(R_m)$ is in $\widehat{\text{mlts}\!\downarrow}$
   (b) $\mathcal{A}(R_m) \mathrel{\dot{\preccurlyeq}} \widehat{r}$
   (c) $\mathcal{A}(B_m) \mathrel{\dot{\preccurlyeq}} \widehat{t}$
4. By the safety conditions (see may part of the proof) and by monotonicity of $\mathcal{G}$, we have:
   (a) $\mathcal{G}(\mathcal{A}(g_i(\mathcal{G}(\widehat{s},\widehat{e})))) \subseteq \mathcal{G}(G_i(\widehat{s},\widehat{e}))$, so
   (b) by definition of Galois Connection, $g_i(\mathcal{G}(\widehat{s},\widehat{e})) \subseteq \mathcal{G}(G_i(\widehat{s},\widehat{e}))$
5. The same follows for F_i, so:
   (a) $f_i(\mathcal{G}(\widehat{s},\widehat{e})) \subseteq \mathcal{G}(F_i(\widehat{s},\widehat{e}))$
6. We are going to prove that the pair $(g_i(\mathcal{G}(\widehat{s},\widehat{e})), f_i(\mathcal{G}(\widehat{s},\widehat{e})))$ is in $\text{lts}_{(\widehat{s},\Box)}$
7. To prove (6), we have to prove there exists s, r and t, such that:
   (a) $s \in \mathcal{G}(\widehat{s})$

   (b) $r \in g_i(\mathcal{G}(\widehat{s}, \widehat{e}))$
   (c) $t \in f_i(\mathcal{G}(\widehat{s}, \widehat{e}))$, such that
   (d) $s \overset{a(t)}{\rightarrow}_{\Box} r$ is in lts

8. By Lemma 2.4.1, $\widehat{s} \neq \bot$, therefore $\mathcal{G}(\widehat{s})$ is non-empty (the same holds for $\mathcal{G}(\widehat{e})$), then:
   (a) Exists s, e such that $s \in \mathcal{G}(\widehat{s})$ and $e \in \mathcal{G}(\widehat{e})$, hence
   (b) $(s, e) \in \mathcal{G}(\widehat{s}, \widehat{e})$, so
   (c) $g_i(\mathcal{G}(\widehat{s}, \widehat{e}))$ is the set-wise application of g_i to $\mathcal{G}(\widehat{s}, \widehat{e})$. Hence $g_i(s, e) \in g_i(\mathcal{G}(\widehat{s}, \widehat{e}))$
   (d) The same holds for $f_i(s, e)$ and $c_i(s, e)$
9. To prove (7.d) with $r = g_i(s, e)$ and $t = f_i(s, e)$, we have to prove that the lpe generates the transition $s \overset{a(f_i(s,e))}{\rightarrow} g_i(s, e)$. So, we just need:
   (a) $c_i(s, e) = \mathsf{T}$, which follows by the safety condition, (1.a) and (8.d),
10. Therefore, $(g_i(\mathcal{G}(\widehat{s}, \widehat{e})), f_i(\mathcal{G}(\widehat{s}, \widehat{e})))$ is in $\text{lts}_{(\widehat{s},\Box)}$
11. Then, either:
   (a) exists (R_m, B_m) in $\text{lts}^{min}_{(\widehat{s},\Box)}$, with:
      i. $R_m \subset g_i(\mathcal{G}(\widehat{s}, \widehat{e}))$ and
      ii. $B_m \subset f_i(\mathcal{G}(\widehat{s}, \widehat{e}))$
   (b) or, $(g_i(\mathcal{G}(\widehat{s}, \widehat{e})), f_i(\mathcal{G}(\widehat{s}, \widehat{e})))$ is in $\text{lts}^{min}_{(\widehat{s},\Box)}$
12. If (11.a) then:
   (a) by (4.b), $R_m \subset \mathcal{G}(G_i(\widehat{s}, \widehat{e}))$
   (b) By monotonicity of $\mathcal{A}$ follows $\mathcal{A}(R_m) \dot{\preccurlyeq} \mathcal{A}(\mathcal{G}(G_i(\widehat{s}, \widehat{e})))$, and
   (c) by definition of Galois Connection and (1.c), $\mathcal{A}(R_m) \dot{\preccurlyeq} G_i(\widehat{s}, \widehat{e}) = \widehat{r}$, and
   (d) The same holds for B_m, $\mathcal{A}(B_m) \dot{\preccurlyeq} \widehat{t}$
   (e) (R_m, B_m) in $\text{lts}^{min}_{(\widehat{s},\Box)}$ implies $\widehat{s} \overset{a(\mathcal{A}(B_m))}{\rightarrow}_{\Box} \mathcal{A}(B_m))$, which proves (3).
13. If (11.b) is true then:
   (a) By monotonicity of $\mathcal{A}$ and (4.b), $\mathcal{A}(g_i(\mathcal{G}(\widehat{s}, \widehat{e}))) \dot{\preccurlyeq} \mathcal{A}(\mathcal{G}(G_i(\widehat{s}, \widehat{e})))$, so
   (b) then by definition of Galois Connection and (1.c), it follows that $\mathcal{A}(g_i(\mathcal{G}(\widehat{s}, \widehat{e}))) \dot{\preccurlyeq} G_i(\widehat{s}, \widehat{e}) = \widehat{r}$
   (c) The same for $f_i(\mathcal{G}(\widehat{s}, \widehat{e}))$
   (d) Then it proves (3), which concludes the proof.

□ (must part)

□ (Theorem)

Proof (Safety Condition):

We are going to prove $\mathcal{A}(cons(\mathcal{G}(\widehat{b},\widehat{l}))) \mathrel{\dot{\preccurlyeq}} abs\_cons(\widehat{b},\widehat{l})$, then

- b is not abstracted therefore $\widehat{b} = b$, then the proof reduces to:

$$\mathcal{A}(cons(b,\mathcal{G}(\{\widehat{l}\}))) \mathrel{\dot{\preccurlyeq}} abs\_cons(b,\widehat{l})$$

- We are going to prove for the homomorphism, therefore, we have to prove:

$$\forall\ l.H(l) = \widehat{l} \Rightarrow H(cons(b,l)) \in abs\_cons(b,\widehat{l})$$

Then we proceed by cases:

- Case $\widehat{l}$ equals $empty$:
  1. $l = emptyList$
  2. $H(emptyList) = empty$
  3. $H(cons(b, emptyList)) = middle$
  4. $abs\_cons(b, empty) = \{middle\}$, which proves the case.
- Case $\widehat{l}$ equals $middle$:
  1. Case l equals $cons(b,l')$ and $len(l) = MAX - 1$:
     (a) $H(l) = middle$
     (b) $H(cons(b,l)) = full$
     (c) $abs\_cons(b, middle) = \{middle, full\}$, which proves the case.
  2. Case l equals $cons(b,l')$ and $0 < len(l) < MAX - 1$:
     (a) $H(l) = middle$
     (b) $H(cons(b,l)) = middle$
     (c) $abs\_cons(b, middle) = \{middle, full\}$, which proves the case.
- Case $\widehat{l}$ equals $full$:
  1. $l = cons(b,l')$ and $len(l) \geq MAX$:
  2. $H(l) = full$
  3. $H(cons(b,l)) = full$
  4. $abs\_cons(b, full) = \{full\}$, which proves the case.

□ (Safety Condition)

Chapter 3

An Abstract Interpretation ToolKit

The implementation of the previously developed theory is an indispensable step in order to apply abstract interpretation techniques to realistic systems. This chapter presents a toolkit that assists in the task of generating modal approximations of μCRL specifications. The tool implements the ideas presented in the previous two chapters. It is conceived to be completely integrated with the existing μCRL toolset and verification methodology.

3.1 Introduction

One of the most important issues to bring abstract interpretation techniques to practise is how to select reasonable abstractions. In general, there exist different abstraction approaches that can be applied within the verification methodology. For example, in *variable hiding* or *pointwise* abstraction first the value of some variables of the specification is considered as unknown, and subsequently, extra non-determinism is added to the system when there are predicates over the abstracted variables. Another automated abstraction technique is so-called *predicate abstraction* in which only the value of some conditions is retained and propagated over the predicates of the specification that the depend on the conditions. *Program slicing* is a technique that tries to eliminate all parts of the specification that are not relevant for the current verification.

The most common abstraction technique consists in interpreting the concrete specification over a smaller data domain. The user selects the set of variables to abstract and provides a new abstract domain that reflects some aspects of the original. This technique requires creative human interaction in order to select the parts of the system that are suitable to abstract and to provide the corresponding data domains. Furthermore, the user must ensure that the abstract interpretation satisfies some so-called safety requirements.

Our tool implements the automatic *pointwise* abstraction and, moreover, assists the user to create his own abstractions. The tool supports the use of two mainstream techniques for data abstraction. The one in which the concrete and the abstract data domain are related via homomorphic functions, as well as the one based on Galois Connections. Furthermore, a lifting mechanism is also implemented which allows to automatically build Galois Connections from homomorphisms. All these techniques were explained in the previous chapters.

Standard abstraction frameworks are only based on the abstraction of states which make them unable to deal with infinitely branching systems with action labels. A unique feature of our tool is that it allows the abstraction of both states and action labels. In the implementation, we try to reuse existing tools as much as possible. In particular, we encode *Modal*-LPEs as LPEs and *Modal*-LTSs as LTSs, in order to reuse the μCRL and CADP toolsets. We also provide a new method to reduce the 3-valued model checking problem to two 2-valued model checking problems.

This chapter starts by giving a general view of the tool and the methodology for applying abstraction techniques to realistic applications. Then, we introduce with more detail the different components of the tool. The chapter concludes with some references to other related tools.

3.2 Overview of the Tool

The following figure describes the tool architecture, whose main components are:
Abstractor. It is in charge of performing the symbolic transformation from LPEs

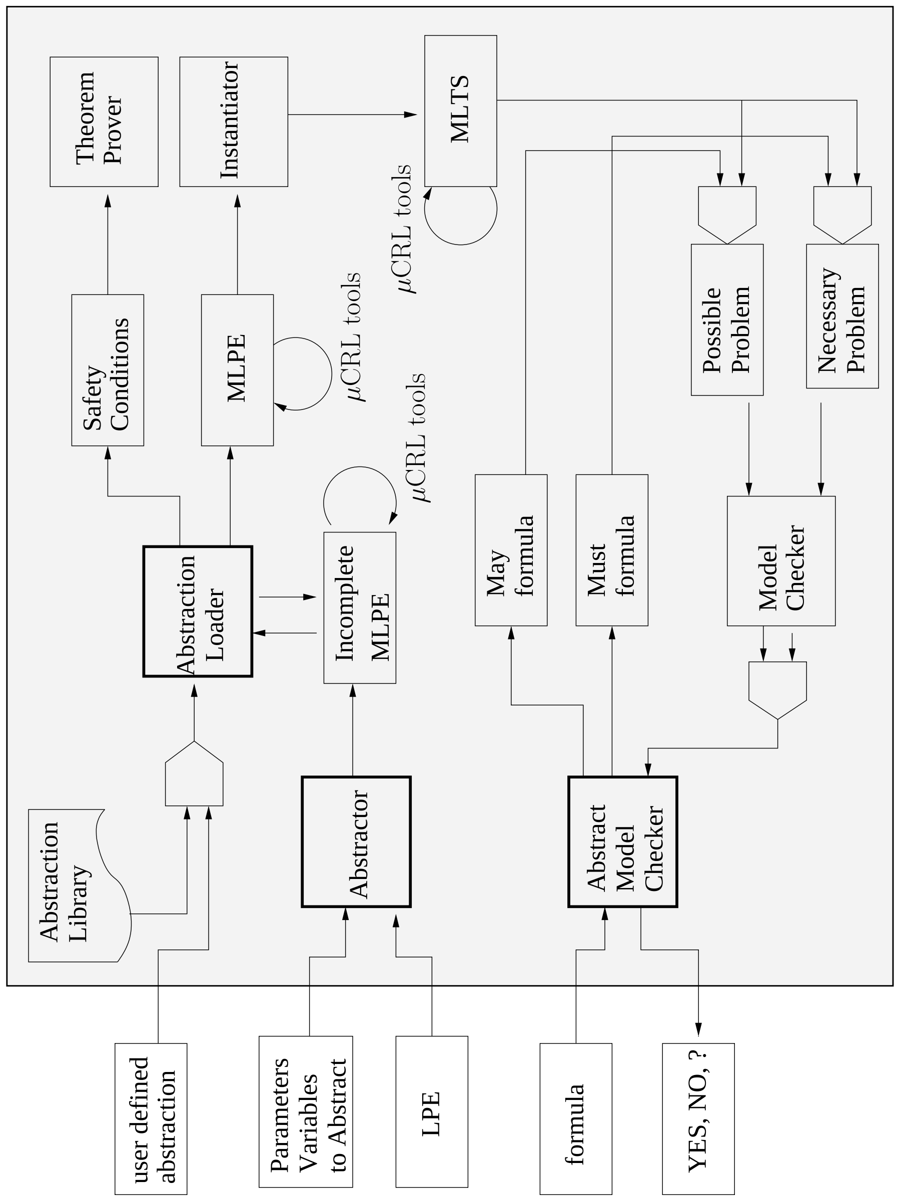


Figure 3.1: Architecture of the toolkit

to *Modal*-LPEs. It gets a μCRL specification in linear format and, typically, a set of parameters and variables to abstract, then it generates a new specification. The new specification is the skeleton of the abstraction, it has to be completed by adding the abstract data specification. The tool allows the use of different ways of abstracting (homomorphisms, Galois Connections and lifted homomorphisms), the resulting specification will depend on the user's choice.

Abstraction Loader. It is in charge of managing the data specifications. From the *Modal*-LPE skeleton, the *Loader* may export the abstract signature that the user has to provide in order to complete the specification. It is also used to import abstract data types from external files, and to generate automatic abstractions by *hiding* variables. As we previously marked, abstract interpretations have to be proved correct, the tool generates the safety conditions that abstract functions have to satisfy. Some safety requirements can be automatically proved correct using the μCRL theorem prover, the others need human interaction.

Abstract Model Checker. The transition system generated from an abstraction represents a double approximation of the original. We use a 3-valued logic in order to infer the satisfaction or refutation of properties. The 3-valued model checking problem can be transformed to two standard 2-valued problems. Hence one can use the existing model-checking tools.

Action labels may be abstracted. Therefore, formulas have to be abstracted according to the abstract action labels. Due to the abstraction of formulas, in some cases, we cannot infer the exact result of the model checking of the concrete formula; in section 3.5, we provide the guidelines to model check and to infer the results.

3.3 Abstractor

We recall that in the previous chapter we have defined the function "$\widehat{\ }$" over process terms that pushes the abstraction till the data part. The *Abstractor* implements the function over process and data terms.

The tool gets as input a *Linear Process Equation* and a set of parameters and local variables to abstract. Alternatively, the user can provide a list of sorts, in this case all parameters and variables of the selected sorts will be abstracted. Subsequently, the input is transformed conforming the user selection by replacing the different symbols that appear in the specification by their abstract counterparts. And propagating the abstraction when it is needed. The output is a *Modal*-LPE.

We have seen that data terms $f_i(d, e_i), g_i(d, e_i)$ and $c_i(d, e_i)$ are composed by function symbols, parameters and local variables. Based on the parameters and variables selected by the user some (or all) function symbols are replaced by their abstract definition. In this section, we present the abstraction criteria that the tool implements.

3.3.1 Abstraction of Function Symbols

The *Abstractor* will traverse the process specification, transforming the function symbols according to the user input. In case the arguments of a function are modified the tool will generate a new signature for the function and will replace the old one.

We recall that a way of capturing the non-determinism induced by the abstracted functions is using sets of values. For instance, we can consider the abstraction of the integers to their sign, i.e., $\{neg, zero, pos\}$; Intuitively, the definition of the abstract successor of *zero* and *pos* will in both cases be *pos*. However the abstract successor of *neg* can be either *neg* or *zero*, therefore, the sort of the abstract successor will be a set of abstract integers. We define *lifting* to be the operation of replacing single values to sets of values. We give now the rules for abstracting and lifting function symbols. Let us consider the function f: $S_0 \times ... \times S_{n-1} \rightarrow S_n$:

1. If there is a data term in the process specification in which the ith argument of f is abstracted then the signature of the function will change according to the following rules:

   (a) All sorts $S_j = S_i$ with $j \in [0, ..., n-1]$ will be abstracted.

   (b) If $S_n = S_i$ then the target sort of f will be abstracted and lifted.

   (c) If $S_n \neq S_i$ then the target sort of f will be lifted.

Let us consider again the abstraction of integers, with the following functions: $succ : Int \rightarrow Int$, $+ : Int \times Int \rightarrow Int$ and $<: Int \times Int \rightarrow Bool$. Then, following the above presented rules:

- If there is a data term in which the argument of *succ* is abstracted then the target sort of the abstract version of *succ* will be abstracted and lifted, i.e., $abs\_succ : abs\_Int \rightarrow \mathcal{P}(abs\_Int)$.
- If one argument of $+$ is abstracted then the other argument will be abstracted as well and the target sort will be abstracted and lifted, i.e., $\widehat{+} : abs\_Int \times abs\_Int \rightarrow \mathcal{P}(abs\_Int)$
- If one argument of $<$ is abstracted then the other argument will be abstracted as well and the target sort will be lifted, i.e., $\widehat{<} : abs\_Int \times abs\_Int \rightarrow \mathcal{P}(Bool)$

Let us consider, now, the abstraction of lists of type D, with the following standard functions: $cons : D \times List \rightarrow List$ and $head : List \rightarrow D$. Then, following the rules:

- If the first argument of *cons* is abstracted then the new signature will be: $abs\_cons : abs\_D \times List \rightarrow \mathcal{P}(List)$.
- If the second argument of *cons* is abstracted then the new signature will be: $abs\_cons : D \times abs\_List \rightarrow \mathcal{P}(abs\_List)$.
- If both arguments of *cons* are abstracted then the new signature will be: $abs\_cons : abs\_D \times abs\_List \rightarrow \mathcal{P}(abs\_List)$.
- If the argument of *head* is abstracted then the new signature will be: $abs\_head : abs\_List \rightarrow \mathcal{P}(D)$.

Furthermore:

2. If there is a data term in which the ith argument of f is lifted then:
   (a) The target sort of f will be lifted.

For example:

- If the argument of *succ* is lifted (but not abstracted) then the target will be lifted, i.e., $succ : \mathcal{P}(Int) \rightarrow \mathcal{P}(Int)$.
- If the argument of *succ* is lifted and abstracted then the target be abstracted and lifted, i.e., $abs\_succ : \mathcal{P}(abs\_Int) \rightarrow \mathcal{P}(abs\_Int)$. (Note that this is the result of the application of rule 1.b).

The tool will automatically generate auxiliary functions and equations to manipulate sets, by providing the *pointwise* lifting of the not lifted ones. For instance, for a function f in which the ith sort has been lifted and the rest remains unlifted, i.e., f: $D_0 \times ... \times \mathcal{P}(D_i) \times ...D_{n-1} \rightarrow \mathcal{P}(D_n)$, the following equation will be generated:

- Let X be of type $\mathcal{P}(D_i)$ and x of type D_i
- $\mathsf{f}(..., X, ...) = \cup\{\mathsf{f}(..., x, ...) \,|\, x \in X\}$[1]

3.3.2 Abstraction of Parameters and Variables

The user selects the list of parameters and variables that he wants to abstract. The choice may influence the sorts of other related parameters. To determine the sorts of the abstract specification we follow the next rules:

3. If a parameter $d_i : D_i$ is selected to be abstracted then its sort will change. The new sort of the abstract parameter will be the powerset of the abstract version of its concrete sort, i.e., $d_i : \mathcal{P}(abs\_D_i)$. The explanation why abstracted parameters are also lifted is that after every recursion, the updated values of the parameters are computed from functions. And, as we have seen in the previous section, to capture the extra non-determinism, functions are lifted to sets. Therefore, the specification may contain assignments in which parameters receive sets of values.
4. If a variable $e_{a_i} : E_{a_i}$ is selected to be abstracted then its sort is changed to the abstract version of the concrete one, i.e., $e_{a_i} : abs\_E_{a_i}$. In this case, we do not lift the sorts of the values to powersets because their values are not induced from any abstracted data term.
5. If a parameter $d_i : D_i$ is not selected to be abstracted but there is an assignment of a data term in which appears an abstract parameter or an abstract variable, or a lifted or abstracted function then the parameter is lifted, i.e., $d_i : \mathcal{P}(D_i)$.

In section 3.3.5, we show some examples of these rules.

[1]Note that function symbols are overloaded.

3.3.3 Abstraction of Sorts

For every abstracted sort the user will have to provide the abstract domain and the relation with the concrete one. The tool supports three ways of relating the domains, the homomorphic and the Galois Connection approach and also the combination of them that consists of the lifting of a homomorphism to a Galois Connection (all introduced in chapters 1 and 2). In practise, this possibility is very fruitful because it permits the user just to provide the mapping between the concrete and the abstract data domain and the definition of the abstract functions. The tool automatically lifts the structure to a Galois Connection.

Not all Galois Connections can be represented by a lifted homomorphism, however the use of the lifted homomorphism may be convenient to perform rapid and powerful abstractions. The lifting technique reduces the number of abstract definitions that the user has to provide to specify the abstract system.

For every abstracted sort abs_D, the tool will generate the signature of the functions $alpha : \mathcal{P}(D) \to \mathcal{P}(abs\_D)$, $gamma : \mathcal{P}(abs\_D) \to \mathcal{P}(D)$ and $\preccurlyeq$: $\mathcal{P}(abs\_D) \times \mathcal{P}(abs\_D) \to \mathcal{P}(Bool)$. The first one represents the abstraction function, the second one the concretisation function and the third the order on the abstract domain. The user selects one of the three types of abstraction. Then in case of homomorphism or lifted homomorphism the following auxiliary function definitions will be generated:

- $H : D \to abs\_D$
- $H^{-1} : abs\_D \to \mathcal{P}(D)$
- $alpha(X) = \{H(x) | x \in X\}$
- $gamma(abs\_X) = \cup\{H^{-1}(abs\_x) \,|\, abs\_x \in abs\_X\}$

In case the user selects Galois Connections the tool will generate:

- $\alpha : \mathcal{P}(D) \to abs\_D$
- $\gamma : abs\_D \to \mathcal{P}(D)$
- $alpha(X) = \{\alpha(\{x\}) \mid x \in X\}$
- $gamma(abs\_X) = \cup\{\gamma(abs\_x) \,|\, abs\_x \in abs\_X\}$
- $lt(abs\_X, abs\_Y) = \forall\, abs\_x \in abs\_X \,\exists\, abs\_y \in abs\_Y.abs\_x \preccurlyeq abs\_y$

In the first case, the user will have to provide H and H^{-1} (if there are conflicting cases, see next section). In the second α, γ and the order $\preccurlyeq$. Note that these definitions correspond to the ones given in Chapter 2, section 2.3.1.

H^{-1} and γ do not have to be provided in general, they are use to solve the conflicts. In some cases, it is not possible to define them because they produce infinite sets. The definition can be always avoided by abstracting more parameters or variables.

3.3.4 Type Conflicts

Abstraction of data terms is done by abstracting first the parameters and variables that appear inside the terms, and then by propagating the abstraction to the function symbols according to the rules specified above. The abstraction may raise some type conflicts. We list below the different conflicts and how they are resolved:

6. There is an assignment in which a parameter of sort $\mathcal{P}(D)$ gets a term d of sort D. Then d is replaced by $\{d\}$.
7. There is an assignment in which a parameter or an argument of a function of sort $\mathcal{P}(abs\_D)$ gets a term d of sort D. Then d is replaced by $alpha(\{d\})$
8. There is an assignment in which a parameter of sort $\mathcal{P}(D)$ gets a term d of sort $\mathcal{P}(abs\_D)$. Then d is replaced by $gamma(d)$.
9. If the data term C_a of a condition is abstracted then it is replaced by $gamma(C_a)$.

The next section includes some examples of these rules.

3.3.5 From LPEs to *Modal*-LPEs

The *Abstractor* replaces the data terms of the LPEs by their abstract counterparts, producing MLPEs. The user can select the parameters and variables to abstract, then the abstraction is propagated over the data terms of the specification, with the rules that we presented in the previous sections. Let us reconsider the example of the buffer:

$$\begin{aligned} Buffer(l:List) = \sum_{b:Bit} & write(b).Buffer(cons(b,l)) \lhd lt(len(l),MAX) \rhd \delta + \\ & read(head(l)).Buffer(tail(l)) \lhd not(isEmpty(l)) \rhd \delta \end{aligned}$$

The linear process specifies a bounded buffer. The process can choose non-deterministically between executing a *write* or a *read* action. The *write* can only be performed if the buffer is not full, i.e., the length of the list that models the buffer is smaller that the maximal length (MAX). The *read* action can be performed if the buffer is not empty. In the first case, the state parameter is updated by concatenating a new bit to the list; in the second case, the first element of the list is removed. The concrete specification has the following signatures:

- $cons : Bit \times List \to List$
- $len : List \to Nat$
- $lt : Nat \times Nat \to Bool$

- $gt : Nat \times Nat \rightarrow Bool$
- $head : List \rightarrow Bit$
- $tail : List \rightarrow List$

If the user selects the parameter l to be abstracted then the propagation of the abstraction will yield the following signatures[2]:

- $abs\_cons : Bit \times \mathcal{P}(abs\_List) \rightarrow \mathcal{P}(abs\_List)$
- $abs\_len : \mathcal{P}(abs\_List) \rightarrow \mathcal{P}(Nat)$
- $lt : \mathcal{P}(Nat) \times Nat \rightarrow \mathcal{P}(Bool)$
- $gt : Nat \times \mathcal{P}(Nat) \rightarrow \mathcal{P}(Bool)$
- $abs\_head : \mathcal{P}(abs\_List) \rightarrow \mathcal{P}(Bit)$
- $abs\_tail : \mathcal{P}(abs\_List) \rightarrow \mathcal{P}(abs\_List)$

To complete the specification, the user has to provide the domain of the abstract list, *abs_List*, the relation between the concrete domain and the abstract one and the definitions for the new functions. All the functions needed to manipulate sets of values are automatically provided by the tool by performing a pointwise application of the non-abstracted ones.

Let us, now, present a process that manipulates lists. The actions the process performs are meaningless and are just selected to illustrate the transformations:

$$
\begin{aligned}
X(l_0 : List, & l_1 : List, n : Nat) = \\
& \sum_{d_0 : D} write(d).X(cons(d_0, l_0), l_1, len(l_0)) \triangleleft lt(n, 3) \triangleright \delta + \\
& swap.X(l_1, concat(l_0, l_1), length(l_1)) \triangleleft isFull(l_0) \triangleright \delta + \\
& \sum_{d_1 : D} display(position(l_0, d_1)).X(l_0, l_1, n) \triangleleft \mathsf{T} \triangleright \delta
\end{aligned}
$$

The concrete signature of the functions $cons, len$ and lt is the same as in the previous example, the other ones are:

- $concat : List \times List \rightarrow List$
- $isFull : List \rightarrow Bool$
- $position : List \times D \rightarrow Nat$

[2]The complete output of the *Abstractor* for this example is given in the next section.

Let the user select the parameter l_0 and the local variable d_1 to abstract then the resulting *Modal*-LPE will be:

$$
\begin{aligned}
&X(\widehat{l_0} : \mathcal{P}(abs\_List), l_1 : \mathcal{P}(List), \mathcal{P}(n) : Nat) = \\
&\quad \sum_{d_0 : D} write(d).X(abs\_cons(d_0, \widehat{l_0}), \{l_1\}, abs\_len(\widehat{l_0})) \lhd lt(n, 3) \rhd \delta + \\
&\quad swap. \\
&\quad X(alpha(\{l_1\}), gamma(abs\_concat(\widehat{l_0}, alpha(\{l_1\}))), abs\_len(l_1)) \\
&\quad \lhd abs\_isFull(\widehat{l_0}) \rhd \delta + \\
&\quad \sum_{\widehat{d_1} : abs\_D} display(abs\_position(\widehat{l_0}, \widehat{d_1})).X(\widehat{l_0}, l_1, n) \lhd \mathsf{T} \rhd \delta
\end{aligned}
$$

The parameters l_1 and n are lifted because they get the values of lifted terms. In general not all the parameters of the vector are lifted, some of them remain unabstracted and unlifted. The second summand shows how type conflicts are solved by using the abstraction and concretisation functions *alpha* and *gamma*. In the third summand we see how the parameters of the action may be lifted. Furthermore, they may also be abstracted. The signature of the function symbols will be:

- $abs\_cons : \mathcal{P}(abs\_List) \times D \to \mathcal{P}(abs\_List)$
- $abs\_len : \mathcal{P}(abs\_List) \to \mathcal{P}(Nat)$
- $len : \mathcal{P}(List) \to \mathcal{P}(Nat)$
- $lt : \mathcal{P}(Nat) \times Nat \to \mathcal{P}(Bool)$.
- $abs\_concat : \mathcal{P}(abs\_List) \times \mathcal{P}(abs\_List) \to \mathcal{P}(abs\_List)$.
- $abs\_isFull : \mathcal{P}(abs\_List) \to \mathcal{P}(Bool)$
- $abs\_position : \mathcal{P}(abs\_List) \times \mathcal{P}(abs\_D) \to \mathcal{P}(Nat)$

Theorems 1.4.6 and 1.4.7 prove that the *Modal*-LTS generated by an *Modal*-LPE is a correct abstraction of the concrete one generated from the LPE if the pairs of data terms (f, F), (g, G) and (c, C) for all summands satisfy the safety conditions. The *Abstractor* transforms concrete data terms by abstracting the parameters, variables and functions from which they are constructed. If the abstract functions satisfy the safety conditions then the full data terms will also satisfy them, the reason is that all the transformations performed by the *Abstractor* (lifting operations to sets, applying α or γ, ...) are monotonic and preserve the safety relation of the functions. The *Loader* can extract from the specification the safety requirements for the abstracted functions.

3.3.6 From *Modal*-LPEs to LPEs$_{may/must}$

Modal-LPEs can be transformed back to standard *Linear Process Equations*. This allows the reuse of the μCRL tools that are conceived to manipulate LPEs. To do that, first we extend the action labels by adding two suffixes. Let *Act-Names* (or *ActN* for short) be the set of action labels of a *Modal*-LPE. We define $ActNames_{may/must} = \{a\_may \,|\, a \in ActNames\} \cup \{a\_must \,|\, a \in ActNames\}$. Then, we duplicate the number of summands generating for every summand of the *Modal*-LPE two new ones, one for the *may* transitions and the other for the *must* transitions. These new summands are built following the patterns presented below. By $\vec{G}_a$ we denote the sort of elements of G_a (the same holds for $\vec{F}_a$). The pattern for homomorphisms is:

$$\begin{aligned}
X(d:\mathcal{P}(D)) = &\sum_{a\in ActN}\sum_{e_a:E_a}\sum_{f_a:\vec{F}_a}\sum_{g_a:\vec{G}_a} a\_may(f_a).X(\{g_a\})\\
&\lhd\, member(\mathsf{T}, C_a(d,e_a))\wedge\\
&member(f_a, F_a(d,e_a))\wedge\\
&member(g_a, G_a(d,e_a))\\
&\rhd\, \delta +\\
&\sum_{a\in ActN}\sum_{e_a:E_a}\sum_{f_a:\vec{F}_a} a\_must(f_a).X(G_a(d,e_a))\\
&\lhd\, not(member(\mathsf{F}, C_a(d,e_a)))\wedge\\
&singleton(F_a(d,e_a))\wedge\\
&member(f_a, F_a(d,e_a))\wedge\\
&singleton(G_a(d,e_a))\\
&\rhd\, \delta
\end{aligned}
\tag{MLPE to LPE (H)}$$

The patterns is derived from the semantics of *Modal*-LPEs presented in section 2.4. For the homomorphism, we require the states of the process and the arguments of the actions to be single abstract values, because every concrete value is mapped to only one abstract one. However, for the Galois Connection we allow them to be sets of values. The pattern for Galois Connections and lifted homomorphisms is:

$$
\begin{aligned}
X(d:\mathcal{P}(D)) = & \sum_{a\in ActN}\sum_{e_a:E_a} a\_may(F_a(d,e_a)).X(G_a(d,e_a)) \\
& \lhd\, member(\mathsf{T}, C_a(d,e_a)) \\
& \rhd\, \delta + \\
& \sum_{a\in ActN}\sum_{e_a:E_a} a\_must(F_a(d,e_a)).X(G_a(d,e_a)) \\
& \lhd\, not(member(\mathsf{F}, C_a(d,e_a))) \\
& \rhd\, \delta
\end{aligned}
\tag{\textit{MLPE to LPE (GC)}}
$$

For the above example, using the Galois Connection approach, the resulting LPE$_{may/must}$ will be:

$$
\begin{aligned}
& X(\widehat{l}:\mathcal{P}(abs\_List)) = \\
& \quad \sum_{b:Bit} write\_may(b).X(abs\_cons(b,\widehat{l})) \\
& \qquad\qquad \lhd\, member(\mathsf{T}, lt(abs\_len(\widehat{l}), MAX)) \rhd \delta + \\
& \quad \sum_{b:Bit} write\_must(b).X(abs\_cons(b,\widehat{l})) \\
& \qquad\qquad \lhd\, not(member(\mathsf{F}, lt(abs\_len(\widehat{l}), MAX))) \rhd \delta + \\
& \quad read\_may(abs\_head(\widehat{l})).X(abs\_tail(\widehat{l})) \\
& \qquad\qquad \lhd\, member(\mathsf{T}, not(abs\_isEmpty(\widehat{l}))) \rhd \delta + \\
& \quad read\_must(abs\_head(\widehat{l})).X(abs\_tail(\widehat{l})) \\
& \qquad\qquad \lhd\, not(member(\mathsf{F}, not(abs\_isEmpty(\widehat{l})))) \rhd \delta
\end{aligned}
$$

The equivalence of the *Modal*-LPE and the LPE$_{may/must}$ is given by the following proposition:

Proposition 3.3.1 Let $\mathcal{M}$ be a *Modal*-LPE, and let m$\mathcal{L}$ be the corresponding *Modal*-LTS $(S, Act, \rightarrow_{may}, \rightarrow_{must}, s_0)$. Moreover, let $\mathcal{M}_{may/must}$ be the equivalent LPE$_{may/must}$ of $\mathcal{M}$, and let $\mathcal{L}$ be its corresponding LTS $(S, Act_{may/must}, \rightarrow, s_0)$. Then, for all $s, s' \in S$ and $a \in ActNames$, with a possibly empty vector $\bar{d}$ of arguments, we have:

- $s \overset{a\_may(\bar{d})}{\rightarrow} s' \iff s \overset{a(\bar{d})}{\rightarrow}_{\Diamond} s'$
- $s \overset{a\_must(\bar{d})}{\rightarrow} s' \iff s \overset{a(\bar{d})}{\rightarrow}_{\Box} s'$

The proposition holds for both types of abstraction.

3.4 Loader

The *Abstractor* returns the skeleton of the abstraction, i.e, an incomplete *Modal*-LPE. In order to generate the corresponding *Modal*-LTS, the user has to complete the *Modal*-LPE by providing the abstract domains and the definition of the abstract functions. The *Abstraction Loader* assists the user to manage abstract domains by providing import/export mechanisms and an automatic abstraction generator.

In the previous example, *abs_List* may be described by a domain with three values $\{empty, one, more\}$, determining when the list is empty, has a single element or more, removing the information about the value of the stored elements. Then, the user has to provide the mapping $H : List \rightarrow abs\_List$[3], as for example:

- $H(emptyList) = empty$
- $H(cons(b, nil)) = one$
- $H(cons(b, cons(b', l))) = more$

Furthermore, he has to provide the definition of the abstracted functions, for instance:

- $abs\_cons(b, empty) = \{one\}$, $abs\_cons(b, one) = \{more\}$ and $abs\_cons(b, more) = \{more\}$
- $abs\_len(empty) = \{0\}$, $abs\_len(one) = \{1\}$ and $abs\_len(more) = \{2, 3, \ldots, maxLength\}$[4]
- $abs\_head(l) = \{b_0, b_1\}$
- $abs\_tail(one) = \{empty\}$ and $abs\_tail(more) = \{one, more\}$

The mode *export* of the *Loader* lists the functions needed to complete the specification, we recall that the functions needed to manipulate sets are automatically generated by the tool. The mode *load* is used to import the definitions. The mode *auto* automatically performs the pointwise abstraction of the sorts and functions.

A *Modal*-LTS, generated from an abstract *Modal*-LPE (over and under) approximates the original system, if every pair of functions (f, abs_F) satisfies a formal requirement. The list of safety conditions is generated by the *Loader* in the format of the μCRL prover [104]. The form of the safety conditions depends also on the type of abstraction. For the example above, choosing the Galois Connection, the following conditions will be generated.

- $\forall\, b, abs\_l :\ lt(alpha(cons(b, gamma(\{abs\_l\}))), abs\_cons(b, abs\_l))$

[3]or $\alpha : \mathcal{P}(List) \rightarrow abs\_List$ depending on the type of abstraction selected by the user.

[4]Concrete lists are considered of bounded length (*maxLength*). Alternatively, one could abstract the sort *Nat* as well.

- $\forall\, abs\_l :\ lt(alpha(len(gamma(\{abs\_l\}))), abs\_len(abs\_l))$
- $\forall\, abs\_l :\ lt(alpha(head(gamma(\{abs\_l\}))), abs\_head(abs\_l))$
- $\forall\, abs\_l :\ lt(alpha(tail(gamma(\{abs\_l\}))), abs\_tail(abs\_l))$

For the (lifted)-homomorphisms, the safety conditions are reduced to:

- $\forall\, b, l :\ H(cons(b, l)) \in abs\_cons(b, H(l))$
- $\forall\, l :\ len(l) \in abs\_len(H(l))$
- $\forall\, l :\ head(l) \in abs\_head(H(l))$
- $\forall\, l :\ H(tail(l)) \in abs\_tail(H(l))$

If the safety conditions hold for the function symbols then, by construction, they will hold for the full data terms. Therefore, instead of proving the safety conditions for every guard, action and next in a process specification we can prove in general that the abstract data specification satisfies the "safety conditions" and then infer that any particular system does as well. This would allow to reuse abstract specifications of the data into different systems and create libraries of abstractions.

3.5 Abstract Model Checking

To integrate the abstract interpretation techniques in the verification methodology we have to provide a relation between the satisfaction of a formula over the abstract system and its reflection to the concrete. This section describes the abstract model checking process for the homomorphic approach, the Galois Connection one may be defined in an equivalent way following the premises explained in Chapter 1. Typically, the process is as follows:

1. The user gives a *concrete* formula φ to prove in the concrete system (from now on M).
2. The arguments of the actions in φ, which are given as concrete sorts, are abstracted, resulting in abs_φ.
3. We check the satisfaction of abs_φ over the abstract model (abs_M, which is described by a *Modal*-LTS).
4. The result of the satisfaction is inferred to the concrete system. The inferences, as we will see, have some restrictions.

(step i) Concrete properties φ are described using the regular alternating-free action-based μ-calculus [88, 89], see Section 2.5. We remember the syntax of the logic:

$$\alpha ::= \mathsf{T} \mid \mathsf{F} \mid \neg\, \alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid a(\bar{d}) \mid \mathit{reg} - \mathit{exp}$$
$$\beta ::= \alpha \mid \beta_1 . \beta_2 \mid \beta_1 | \beta_2 \mid \beta * \mid \beta +$$
$$\varphi ::= \mathsf{T} \mid \mathsf{F} \mid \neg\, \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\beta]\varphi \mid \langle\beta\rangle\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi$$

We recall that in order to be used as input for CADP toolset formulas have to be alternating free.

(step ii) As we have shown in the previous section, action arguments may be abstracted and/or lifted to sets during the abstraction process. In order to prove φ, we transform it to abs_φ by replacing every concrete argument of the actions by its abstract counterpart, i.e, $a(d)$ will be rewritten to $a(H(d))$.

(step iii) Following [75], an abstract formula is interpreted dually over an *Modal*-LTS, i.e. there will be two sets of states that satisfy it. A set of states that necessarily satisfy the formula and a set of states that possibly satisfy it. From the practical point of view, an interesting fact is that the 3-valued model checking problem can be easily transformed to two standard 2-valued problems. This allows the use of existing model checking tools such as the evaluator of the CADP toolset [43].

To do the translation, we follow the ideas of [21, 55]. Basically, given a formula abs_φ we generate two different formulas abs_φ_{must} and abs_φ_{may}, the first one will be used to determine when a system *necessarily* satisfies a property and the second when it *possibly* does. They have the same structure as abs_φ but are built over $ActNames_{may/must}$ instead of over $ActNames$. For this purpose, we define two recursive operators $\mathcal{T}_{may}$ and $\mathcal{T}_{must}$. See below for the definition of the first one ($\mathcal{T}_{must}$ is dual):

- $\mathcal{T}_{may}(\neg\, abs\_\varphi) = \neg \mathcal{T}_{must}(abs\_\varphi)$
- Replace each occurrence of $[\beta]$ in abs_φ by $[\beta_{must}]$
- Replace each occurrence of $\langle\beta\rangle$ in abs_φ by $\langle\beta_{may}\rangle$
- For the rest of the cases, $\mathcal{T}_{may}$ is pushed inwards.

β_{may} replaces all occurrences of α by α_{may} which is defined as follows:

- if $\alpha = a(\bar{d})$ then $\alpha_{may} = a\_may(\bar{d})$.
- if $\alpha = \mathsf{T}$ then $\alpha_{may} = T_{may}$. It matches all *may* actions.
- if $\alpha = \mathsf{F}$ then $\alpha_{may} = \neg\,(T_{may})$. It matches actions that are not *may*. $\neg\,(T_{may})$ is equivalent to T_{must}.
- if $\alpha = \neg\,(\alpha')$ then $\alpha_{may} = \neg\, \alpha'_{may} \wedge T_{may}$. It matches all may actions that do not match α'_{may} .

These transformations are done time linear in the size of the formula. The difference between this approach and the one used by Godefroid and al. [55] is that instead of generating two different models and using one single formula, we use a single model and two versions of the formula. In general formulas are much smaller than systems and their duplication is less expensive. We present, below, some typical properties, in the abs_φ_{must} form:

Deadlock freedom, with regular expressions:

(P1): $[\ '.*_{\mathsf{may}}.*'*\ ]\ \langle\ '.*_{\mathsf{must}}.*'*\ \rangle\ \mathsf{T}$

The dot . in the regular expressions inside the action formulas matches any character, therefore $.*$ matches any number of occurrences of any character. Deadlock freedom, with fixed point operators:

(P2): ν X. $(\langle\ '.*_{\mathsf{must}}.*'\ \rangle\ \mathsf{T} \wedge\ [\ '.*_{\mathsf{may}}.*']\ X)$

No execution sequence leads to a:

(P3): $[\ '.*_{\mathsf{may}}.*'*\ .\ '\mathsf{a}_{\mathsf{may}}'\ ]\ \mathsf{F}$

There exists a sequence leading to a:

(P4): $\langle\ '.*_{\mathsf{must}}.*'*\ .\ '\mathsf{a}_{\mathsf{must}}'\ \rangle\ \mathsf{T}$

All sequences lead to a:

(P5): μ X. $(\langle\ '.*_{\mathsf{must}}.*'\ \rangle\ \mathsf{T} \wedge\ [\ \neg('\mathsf{a}_{\mathsf{may}}' \wedge\ '.*_{\mathsf{may}}.*')\ ]\ X)$

The next two properties are the abstraction of **C2** and **C3** presented in the previous chapter:

(A2): $[\ '.*_{\mathsf{may}}.*'*\ .\ '\mathsf{R}_{\mathsf{may}}(.*,\{\mathsf{I}_{\mathsf{nok}}\})'\ ]\ \langle\ '.*_{\mathsf{must}}.*'*\ .$
$('\mathsf{S}_{\mathsf{must}}(\{\mathsf{I}_{\mathsf{dk}}\})' \vee\ '\mathsf{S}_{\mathsf{must}}(\{\mathsf{I}_{\mathsf{nok}}\})'))\rangle\ \mathsf{T}$

(A3): $[\ '.*_{\mathsf{may}}.*'*\ .\ '\mathsf{R}_{\mathsf{may}}(.*,\{\mathsf{I}_{\mathsf{nok}}\})'\ ]\ \mu$ X. $(\langle\ '.*_{\mathsf{must}}.*'\ \rangle\ \mathsf{T}\wedge$
$[\ \neg(('\mathsf{S}_{\mathsf{may}}(\{\mathsf{I}_{\mathsf{dk}}\})' \vee\ '\mathsf{S}_{\mathsf{may}}(\{\mathsf{I}_{\mathsf{nok}}\})) \wedge\ '.*_{\mathsf{may}}.*')\ ]\ X)$

(step iv) The result of the abstract model checking process gives a 3-valued result:

- abs_M satisfies abs_φ_{must}.
- abs_M satisfies abs_φ_{may} but does not satisfy abs_φ_{must}.

- abs_M does not possibly satisfy abs_φ_{may}.

In the first case, we are able to infer the satisfaction of φ, i.e., $abs\_M \models \mathcal{T}_{must}(abs\_\varphi) \Rightarrow M \models_H abs\_\varphi$. In the third case, we are able infer the refutation of φ , i.e., $abs\_M \not\models \mathcal{T}_{may}(abs\_\varphi) \Rightarrow M \not\models_H abs\_\varphi$ However, the second case does not give any information about satisfaction or refutation of the property. The inference of the satisfaction or refutation of the concrete formulas is not straightforward. The reason is that by abstracting actions we have lost the exact information about concrete transitions.

Above, $\models_H$ defines the satisfaction of an abstract formula over a concrete system. The semantics of state and regular formulas do not change. We represent by $[\![abs\_\alpha]\!]_H$ the set of concrete actions that satisfy the abstract action formula abs_α. The semantics is given below:

$$\begin{array}{ll}
[\![\mathsf{T}]\!]_H = Act & [\![\mathsf{F}]\!]_H = \emptyset \\
[\![abs\_\alpha_1 \wedge abs\_\alpha_2]\!]_H & = [\![abs\_\alpha_1]\!]_H \cap [\![abs\_\alpha_2]\!]_H \\
[\![abs\_\alpha_1 \vee abs\_\alpha_2]\!]_H & = [\![abs\_\alpha_1]\!]_H \cup [\![abs\_\alpha_2]\!]_H \\
[\![\neg\, abs\_\alpha']\!]_H & = Act \setminus [\![abs\_\alpha']\!]_H \\
[\![a(abs\_d)]\!]_H & = \{a(d) \mid H(d) = abs\_d\}
\end{array}$$

We now give an example. Let us consider the system in Figure 3.2:

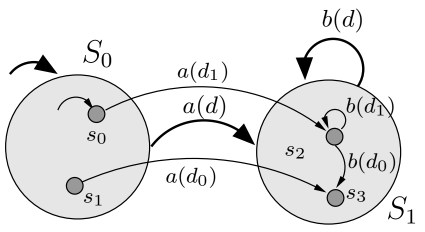


Figure 3.2: Example of abstract Model Checking.

The abstraction is built by mapping s_0 and s_1 to S_0, s_2 and s_3 to S_1 and d_0 and d_1 to d. We want to prove the following properties:

- *"It is possible to do a transition $a(d_0)$ from the initial state"*
  $s_0 \models \langle a(d_0)\rangle\mathsf{T}$. The abstract version of the formula is $\langle a(d)\rangle\mathsf{T}$, which trivially holds for S_0. Therefore, we can infer that there exists x such that $H(x) = d$ for which $\langle a(x)\rangle\mathsf{T}$ holds in s_0. In other words, $s_0 \models \langle a(d_0) \vee a(d_1)\rangle\mathsf{T}$ which implies that $s_0 \models \langle a(d_0)\rangle\mathsf{T}$ or $s_0 \models \langle a(d_1)\rangle\mathsf{T}$.

- *"It is not possible to do a transition $b(d_0)$ from the initial state"*
  $s_0 \models [\,b(d_0)\,]\,\mathsf{F}$. The abstract version of the formula is $[\,b(d)\,]\,\mathsf{F}$, which trivially holds for S_0. Therefore, we can infer that for all x such that $H(x) = d$ implies $[\,b(x)\,]\,\mathsf{F}$ holds in s_0. In other words, $s_0 \models [\,b(d_0) \vee b(d_1)\,]\,\mathsf{F}$ which implies that $s_0 \models [\,b(d_0)\,]\,\mathsf{F}$ and $s_0 \models [\,b(d_1)\,]\,\mathsf{F}$.

In the first case, we have less information than we requested due to the abstraction, and we cannot infer the exact satisfaction or refutation of the original formula in the concrete model. In the second case we have enough to infer the exact result. The output of the inference can be described by quantifiers over the actions using a logic such as the one presented in [118].

Note that in the special case of action labels without data arguments abs_φ will be equal to φ so the abstract model checking problem coincides with the classical theories based on state abstraction only.

An important aspect of abstract model checking refers to spurious counter-examples. Some formulae are not satisfied due to non-realistic scenarios i.e. abstract traces that do not have any corresponding concrete one. In these cases, it is possible to improve the precision of the abstract model in two ways:

- By removing the spurious *may* traces. Which gives a model with less possible behaviours.
- By adding extra *must* traces. Which gives a model with more necessary behaviours.

Classical theories such as [82] eliminate possible behaviours using refinement by program execution. The second possibility is studied in chapter 5. Another way to deal with spurious counter-example is to strength the formula to prove in order to discriminate the non-realistic traces, this possibility is studied in [51]. None of these theories are implemented in the tool yet.

3.6 Conclusion

Automated applications are indispensable to apply formal methods to realistic industrial systems. Here we have described a toolkit that helps in using the abstraction techniques theoretically introduced in the first chapters. The tool described is not the only one dedicated to such tasks.

The existing tool closest to ours is αSpin [51] which provides an interface for abstracting PROMELA specifications. The user can select abstractions from a library. The tool produces an over-approximation of the system. The Bandera toolset [69] implements the same method of abstraction, furthermore it provides algorithms for *program slicing* and data dependency analysis in order to automatically find suitable variables to abstract. Bandera generates PROMELA code from simple Java programs.

FeaVer [73] and abC [35] abstract C programs by hiding variables. The first one translates the code to PROMELA, furthermore it also allows the user to define his own abstractions, the latter abstracts directly the C code by implementing an extension of the GCC compiler. Java PathFinder [70], BeBop [5] and SLAM [6] use *predicate abstraction.* We refer to [32] for an extended overview of tools and techniques for abstract model checking.

All the enumerated tools only generate over-approximations, therefore they are only able to check for the satisfaction of *safety* properties. Our tool supports

μ-calculus, therefore, we can use indistinctly *safety* and *liveness* properties. Furthermore, the transformation from LPEs to *Modal*-LPEs allows to reason about the abstract system on a syntactic level, and embeds all the techniques in the existing μCRL tools. Finally, another feature that is not provided by any other tool is the possibility of abstracting action labels.

Besides the case of study of the bounded retransmission protocol presented in the previous chapter. We have used to tool to verify other applications like a real-life distributed system for lifting trucks (lorries, railway carriages, buses and other vehicles) [60]. Furthermore, we have studied abstractions of characteristic applications implemented on a shared data-space architecture. These case studies are briefly presented further in this thesis.

In [98], the tool was used to attempt to improve the performance of distributed algorithms for model checking and state space reduction. The idea is to introduce a new distribution policy of state spaces over workers. This policy reduces the number of transitions between states located at different workers. This in turn is expected to reduce the communication costs of the distributed algorithms. We have used the automatic abstraction mechanism of the tool to compute a small approximation of the state space, starting from some high level description of the system. Based on this approximation, the connectivity of concrete states is predicted. This information is used to distribute states with expected connectivity to the same worker.

The tool implements a simple automatic abstraction approach, as *variable hiding*, and facilitates the use of creative abstractions. More work is needed to automate the task of selecting suitable abstractions and of providing correct abstract domains. The next chapter describes a general pattern to abstract replicated processes that can be used in many different applications.

As presented at the end of the previous section, an interesting aspect that should be studied deeper is how to deal with spurious counter-examples. Chapter 5 analyses a way of dealing with false negatives raised while model checking of progress formulae.

Chapter 4

Linearization and Abstraction of Replicated Processes

In practise, distributed systems are quite often composed by an arbitrarily large but finite number of processes that execute a similar program. The automatic verification of such systems is very limited by the well known state explosion problem. We propose a general framework for specifying and abstracting the parallel composition of uniform processes with data, which allows to perform model checking of realistic instances of such systems. Furthermore, in some cases, the technique allows to generalise the satisfaction of the properties to any number of processes.

4.1 Introduction

Abstraction is a powerful technique to reduce the complexity of systems. It is well known that selecting suitable abstractions is in general very hard. There are no universal abstractions that assist in the verification of any kind of systems. In this chapter we consider a class of applications that appear quite often: distributed systems composed of an arbitrary but finite number of uniform processes with data.

Based on the results of the previous chapters and in [65, 109], we develop a general framework and its formal requirements for safely abstracting the system by performing abstract interpretation of data. The framework consists in a special linear process equation that is proved suitable to define and abstract such systems. Moreover, we present two abstraction patterns, which fulfil the requirements and can be embedded in the general framework. The patterns are:

1. *Abstraction of process state:* instead of keeping the state of each process, we only count the number of processes that are in a certain state.
2. *Abstraction of the state counter:* instead of storing the exact number of processes that are in a same state, we only consider some specific cases of the counter.

Furthermore, we present a special abstraction schema for systems composed by indistinguishable processes, i.e., their behaviour does not depend on their identity. We illustrate the feasibility of our technique by verifying a (simplified) distributed lift system [60] and a shared data space architecture built over the JavaSpaces architecture [106] (see the second part of this thesis).

Our approach can be used to verify large instances of distributed systems. Moreover, in combination with classical data abstraction, we can generalise the results to any instance of parameters of the system. We include two examples using the results of the chapter.

Related work. The *Parameterized model checking problem*, which is in general not decidable [3], has been addressed in several works using different approaches. Abstraction techniques for model checking are sound but incomplete, and need human creative interactions in order to select the appropriate abstractions.

The closest to ours are [76, 110]. Ip and Dill [76] use a special data type to represent process identities and perform an abstraction that maps the processes that are in a certain state to the values {*zero, more, zero_or_more*}. The work by Pong and Dubois [110] follows the same idea but needs more user interaction in order to define the abstract behaviour of the abstracted processes. An improvement of our approach with respect to theirs is that we can deal with both safety and liveness properties. Moreover, we do not give a fixed abstraction mapping but a general pattern that can be instantiated with different abstraction relations. The parallel composition of processes is automatically translated to a required form, therefore the user only needs to define the desired abstractions.

Liveness for parameterized systems was already addressed in [103]. To use this approach, one has to define safe acceleration schemes in order to infer liveness properties. Automated and complete techniques, e.g., the one proposed by Emerson and Kahon for Snoopy Cache Coherence Protocols [40], are restricted to a particular set of systems. Other sound but incomplete methods use, for example, inductive invariants [116] automatically generated from small instances of a system that hold in every larger instance of it. Another approach is based on a *cutoff* theorem, which has to be found and proved in order to generalise the verification result (see a.o. [39]).

4.2 Parallel Uniform Processes

Linearization. We use μCRL to specify systems composed by an arbitrary number of uniform processes. We assume that the processes are *loosely coupled*, i.e., they do not communicate directly with each other. This requirement is not necessary, it is just to simplify the development (in section 4.5, we give the general form for any kind of process). However, we allow them to communicate with external processes that may play the role of networks or coordination architectures. Uniform processes share the same specification, i.e., they are syntactically the same. This does not mean that their behaviour is equal for all of them. Every processes is uniquely identified, by a natural number k, and its behaviour may be determined by its identity. From now on, we assume that the uniform processes share the following linear form (see definition 2.1):

$$\begin{aligned}
&P(k : Nat, d : D) = \\
&\qquad \textstyle\sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(k, d, e_i)). \\
&\qquad P(k, g_i(k, d, e_i)) \lhd c_i(k, d, e_i) \rhd \delta
\end{aligned} \tag{*}$$

This linear form makes no restriction on the specification of the processes. For a process $P(k, d)$, k is the identity and d the data parameter of some arbitrary data type D representing the state of the process.[1] We assume the existence of a global constant $N > 1$ denoting the number of uniform processes, therefore k is in the range $\{0, \ldots, N-1\}$.

Groote and van Wamel [65] defined an equation that models the parallel composition N of such processes, it uses a data type *DTable* to store the values of parameters d of each process. It defines tables indexed by natural numbers, and each element has the data type D. Based on their definitions, we specify a different representation that is more appropriate for performing abstractions.

Let K denote the set $\{0, \ldots, N-1\}$. The data type *DTable* has the signature of $K \to D$. Each table is a function from K to D. Thus, a process with identity k only has one state in a table. Furthermore, we define *update* as a function to update the old value e of process $P(k)$ with d, and *test* as a function to check whether the specified position and data are in the table.

[1] Typically processes have a vector of parameters, using pairing and projections we can easily see that the use of a single parameter d is not an essential limitation.

$$\begin{aligned} \mathit{update} : &\quad K \times D \times D \times \mathit{DTable} \to \mathit{DTable} \\ \mathit{test} : &\quad K \times D \times \mathit{DTable} \to \mathit{Bool} \end{aligned}$$

The defining equations are:

$$\begin{aligned} \mathit{update}(k,d,e,dt) =_{def} &\quad dt\,[\,k := d\,] \\ \mathit{test}(k,d,dt) =_{def} &\quad dt(k) = d \end{aligned}$$

The argument e of *update* represents the old value of the process k, it is not necessary for the definitions of concrete linear systems, however we will see that it is helpful to define abstraction patterns. Let d_k be the initial value of the process k and dt be initially defined as $dt(k) = d_k$ for all $k \in K$, then:

Theorem 4.2.1 Given a process P as defined in (*), the system $P(0,d_0) \mid\mid P(1,d_1) \mid\mid \cdots \mid\mid P(N-1,d_{N-1})$ is strongly bisimilar to $Q(dt)$, where dt is equal to $[d_0, d_1, ..., d_{N-1}]$ and Q is an LPE of the form:

$$\begin{aligned} & Q(dt : \mathit{DTable}) = \\ & \qquad \textstyle\sum_{i \in I} \sum_{k:Nat} \sum_{d:D} \sum_{e_i:E_i} a_i(f_i(k,d,e_i)). \\ & \qquad Q(\mathit{update}(k, g_i(k,d,e_i), d, dt)) \\ & \qquad \lhd \mathit{test}(k,d,dt) \wedge c_i(k,d,e_i) \wedge k < N \rhd \delta \end{aligned}$$

Proof: Let ξ denote the state vector $[d_0 \times \ldots \times d_{N-1}]$ of the processes $P(0,d_0) \mid\mid P(1,d_1) \mid\mid \cdots \mid\mid P(N-1,d_{N-1})$. We define the relation $\mathcal{R}$:

$$dt\ \mathcal{R}\ \xi \iff \forall\, k \in K.\ \ dt(k) = \xi[k]$$

Initially, is trivial to see that $[d_0, d_1, ..., d_{N-1}]\,\mathcal{R}\,[d_0 \times \ldots \times d_{N-1}]$. Then, to prove that the parallel composition of N processes (from now on S) is bisimilar to Q, we have to prove that if from a state ξ, S can do a step to ξ' and ξ is related to dt, then $Q(dt)$ can do the same action to dt' and $dt'\ \mathcal{R}\ \xi'$, and the other way around.

- Let us consider the first implication:
  1. If $S = P(0,\xi[0]) \mid\mid P(1,\xi[1]) \mid\mid \cdots \mid\mid P(N-1,\xi[N-1])$ can do an a-step (a step labelled with a) implies that there exists one k such that $P(k,\xi[k])$ can do an a-step.[2]
  2. If $P(k,\xi[k])$ does an a-step implies that there exists a summand i and an e such that $c_i(k,\xi[k],e) = \mathsf{T}$ and $a = a_i(f_i(k,\xi[k],e))$.
  3. If $P(k,\xi[k])$ performs the step then ξ' will be $\xi[\xi[k] := g_i(k,\xi[k],e)]$.
  4. $\xi\ \mathcal{R}\ dt$ implies that $dt(k) = \xi[k]$. Then, $\mathit{test}(k,\xi[k],dt) = \mathsf{T}$.
  5. The condition of the ith summand of Q is: $\mathit{test}(k,d,dt) \wedge c_i(k,d,e_i) \wedge k < N$. The condition is true for $d = \xi[k]$. Then:

[2]Remember, that we assume that processes do not communicate between each other

6. dt' will be equal to $update(k, g_i(k, \xi[k], e), d, dt)$, i.e., $dt' = dt[k := g_i(k, \xi[k], e)]$
7. Hence $\xi' \mathcal{R}\ dt'$ which proves the case.

- The other implication is similar

Hence, S and Q are bisimilar.

□ (Theorem)

Theorem 4.2.1 states that any parallel composition of uniform processes can be encoded using an LPE and a data type *DTable*. Instead of the condition *test*, Groote and van Wamel used a function *get* to access the state of the processes. Both approaches are equivalent for defining concrete systems, in which every process is in only one state. However, our approach minimises the extra non-determinism added by the abstractions that do not allow to determine the exact state of the processes.

The process Q as defined in Theorem 4.2.1 trivially satisfies the following invariant (I1):

$$\sum_{d \in D} \sum_{k \in K} (dt(k) = d) = N$$

The invariant states that the addition of the number of processes by state is equal to the number of processes. Basically, means that every process k is in one and only one state d. This invariant will be uses in future proofs.

Abstraction. Now we present an abstraction framework for an LPE in Theorem 4.2.1. It is composed by some definitions and requirements that any particular instance of abstraction must fulfil. To perform an abstraction it is needed to specify a mapping $\mathcal{H}$[3] from concrete tables *DTable* to abstract ones *abs_DTable*. Furthermore, the concrete linear form is symbolically abstracted to the following *Modal*-LPE (see Definition 2.2):

$$\begin{aligned}
&abs\_Q(abs\_dt : \mathcal{P}(abs\_DTable)) = \\
&\qquad \textstyle\sum_{i:I} \sum_{k:Nat} \sum_{d:D} \sum_{e_i:E_i} a_i(f_i(k, d, e_i)). \\
&\qquad abs\_Q(abs\_update(k, g_i(k, d, e_i), d, abs\_dt)) \\
&\qquad \lhd abs\_test(k, d, abs\_dt) \wedge c_i(k, d, e_i) \wedge k < N \rhd \delta
\end{aligned}$$

abs_Q is the abstract version of the process defined in Theorem 4.2.1, it gets as a parameter the abstract specification *abs_DTable*, that is initialised by the abstraction of the concrete initial table, and can be accessed with the functions *abs_update* and *abs_test*, which have the following signatures:

$$\begin{aligned}
abs\_update &: K \times D \times D \times abs\_DTable \to \mathcal{P}(abs\_DTable) \\
abs\_test &: K \times D \times abs\_DTable \to \mathcal{P}(Bool)
\end{aligned}$$

[3] In this chapter, we only consider abstractions by homomorphisms. We believe that it can be extended to the Galois Connection framework, without extra difficulties.

Recall that all the functions point-wisely apply to sets of values. The function symbols appearing in the data terms are: $abs\_test, \wedge, c_i, <, f_i, abs\_update$ and g_i, from which only *abs_test* and *abs_update* are abstracted. Therefore, in order to prove the correctness of an instance of abstraction, the following conditions have to hold: $\forall k \in K,\ d, e{:}D$ and $dt{:}DTable$

$$\begin{aligned} \mathcal{H}(update(k, d, e, dt)) \in{} & abs\_update(k, d, e, \mathcal{H}(dt)) \\ test(k, d, dt) \in{} & abs\_test(k, d, \mathcal{H}(dt)) \end{aligned}$$

The remaining functions appearing in the specification are not abstracted, so there is no *safety* requirement related with them, as we have seen in previous chapters. A direct consequence of the fulfilment of the requirements and by Theorem 2.4.2 is that the *Modal*-LTS generated from the abstract specification is a *safe* abstraction of the original system, therefore it can be used to prove the satisfaction and/or the refutation of *safety* and *liveness* properties.

We see that processes are abstracted using standard data abstraction. By linearizing we encode the behaviour of the processes with a table, then we use abstraction to reduce the range of values of the table. This abstraction can be used in combination with other kinds of abstractions. For example, we can abstract the data type D that represents the state of the processes to reduce even further the size of the system. In the following section we present some instances of the general abstraction framework.

4.3 Abstraction Patterns

The previous section presented the abstract linear form for replicated processes composed in parallel, now we give two abstract data definitions that satisfy the *safety* requirements.

4.3.1 Abstraction of the Processes State

Instead of storing the values d of every process we just save the number of processes that are in a certain state. See Figure 4.1 for as graphical explanation.

Let *Count* denote the set $\{0, \dots, N\}$. Let *Succ* be the successor function defined as $Succ(c) = c+1$ for $c \in \{0, ..., N-1\}$ and $Succ(N) = N$, and let *Pred* be the predecessor function defined as $Pred(c) = c - 1$ for $c \in \{1, ..., N\}$ and $Pred(0) = 0$. First, we give a function *match* from *Count* to $\mathcal{P}(Bool)$ with the defining equations as follows:

$$match(c) =_{def} \begin{cases} \{\mathsf{T}\} & \text{if } c = N \\ \{\mathsf{T}, \mathsf{F}\} & \text{if } 0 < c < N \\ \{\mathsf{F}\} & \text{if } c = 0 \end{cases}$$

match checks whether a process is in a given state. The result of the function is $\{\mathsf{T}\}$ when all the processes are in the given state; $\{\mathsf{F}\}$ when no process is in the state; otherwise, $\{\mathsf{T},\mathsf{F}\}$, since we do not know the exact answer. The last case will introduce non-determinism to the model of the system.

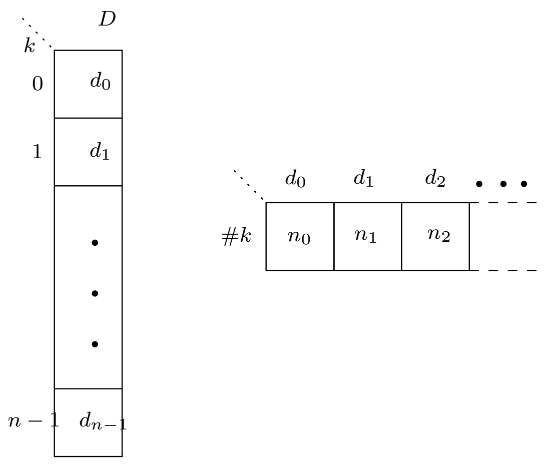


Figure 4.1: Abstract table 1: *by a counter*

Next, we specify *abs_DTable* as the type $D \to Count$. Each table *abs_dt* is a function from D to *Count*, *abs_dt(d)* expresses the number of processes that are in the state d. *abs_update* updates the number of processes in a certain state and *abs_test* is a function to check if a process is in a certain state. The definitions are:

$$\begin{aligned} abs\_test(k, d, abs\_dt) &=_{def} \quad match(abs\_dt(d)) \\ Succ(abs\_dt, d) &=_{def} \quad abs\_dt[d := Succ(abs\_dt(d))] \\ Pred(abs\_dt, d) &=_{def} \quad abs\_dt[d := Pred(abs\_dt(d))] \\ abs\_update(k, d, e, abs\_dt) &=_{def} \quad \{Succ(Pred(abs\_dt, e), d)\} \end{aligned}$$

If a process goes to one state, then we first decrement the counter of the previous state, and afterwards we increment the counter of the new state. The *abs_test* function does not depend on the index of the process, it only depends on the state. We define the abstraction function $\mathcal{H}_t$ from *DTable* to *abs_DTable* as follows:

$$\mathcal{H}_t(dt)(d) =_{def} \sum_{k \in K}(dt(k) = d)$$

which means that for every state d, we compute the number of processes k that are in d $(dt(k) = d)$.

Theorem 4.3.1 The mapping $\mathcal{H}_t$ and the data type *abs_DTable* with the functions *abs_update* and *abs_test* define a *safe* abstraction.

Proof: As we have seen in the general framework, it is enough to prove the following *safety conditions*:

$$\begin{aligned} \mathit{test}(k,d,dt) \in\ & \mathit{abs\_test}(k,d,\mathcal{H}_t(dt)) \\ \mathcal{H}_t(\mathit{update}(k,d,e,dt)) \in\ & \mathit{abs\_update}(k,d,e,\mathcal{H}_t(dt)) \end{aligned}$$

We prove the first condition:

By definition, we have:

- $\mathit{test}(k,d,dt)$ is equal to $(dt(k)=d)$
- $\mathcal{H}_t(dt)(d) = \sum_{k\in K}(dt(k)=d)$
- $\mathit{abs\_test}(k,d,\mathcal{H}_t(dt)) = \mathit{match}(\mathcal{H}_t(dt)(d))$

Then:

- Case $\mathit{test}(k,d,dt) = \mathsf{T}$.
  - We know that there is at least one process k in the state d hence $\sum_{k\in K}(dt(k)=d) \geq 1$. Therefore, $\mathcal{H}_t(dt)(d) \geq 1$
  - We distinguish several cases:
    1. if $\mathcal{H}_t(dt)(d) < N$ then $\mathit{match}(\mathcal{H}_t(dt)(d)) = \{\mathsf{T},\mathsf{F}\}$;
    2. if $\mathcal{H}_t(dt)(d) = N$ then $\mathit{match}(\mathcal{H}_t(dt)(d)) = \{\mathsf{T}\}$;
    3. by (I1) $\mathcal{H}_t(dt)(d) > N$ is not possible.
  - Hence the condition is true for this case.
- Case $\mathit{test}(k,d,dt) = \mathsf{F}$.
  - Since $\mathit{test}(k,d,dt) = \mathsf{F}$, it follows that not all the processes are in the state d, hence $\sum_{k\in K}(dt(k)=d) < N$. Therefore, $\mathcal{H}_t(dt)(d) < N$.
  - We distinguish the cases:
    1. if $\mathcal{H}_t(dt)(d) = 0$ then $\mathit{match}(\mathcal{H}_t(dt)(d)) = \{\mathsf{F}\}$;
    2. if $\mathcal{H}_t(dt)(d) > 0$ then $\mathit{match}(\mathcal{H}_t(dt)(d)) = \{\mathsf{T},\mathsf{F}\}$.
- In all cases, we obtain $\mathit{test}(k,d,dt) \in \mathit{abs\_test}(k,d,\mathcal{H}_t(dt))$, which proves the condition.

For the second condition, we see by the definition of the concrete linear form that *update(k,d,e,dt)* is only applied to e's that satisfy *test(k, e, dt)*, therefore, by definition of *test*, $dt(k) = e$. We will use this invariant of the system to facilitate the proof:

By definition, we have:

- $\mathit{update}(k,d,dt(k),dt)$ is equal to $dt\,[\,k := d\,]$
- $\mathcal{H}_t(dt)(d) = \sum_{k\in K}(dt(k)=d)$
- $\mathit{abs\_update}(k,d,dt(k),\mathcal{H}_t(dt)) = \{\mathit{Succ}(\mathit{Pred}(\mathcal{H}_t(dt),e),d)\}$

Then:

- Case $d = dt(k)$ (transition to the same state)
  1. By definition, $\mathit{update}(k, d, d, dt) = dt$
  2. Let c be $\sum_{k' \in K}(dt(k') = d)$. Then, $c > 0$ because there is at least one process in d.
  3. $\mathit{abs\_update}(k, d, d, \mathcal{H}_t(dt)) = \{\mathcal{H}_t(dt)\,[\,\mathcal{H}_t(dt)(d) := \mathit{Succ}(\mathit{Pred}(c))\,]\}$
  4. $\mathit{Succ}(\mathit{Pred}(c))$ for $c > 0$ equals c, hence $\mathit{abs\_update}(k, d, d, \mathcal{H}_t(dt)) = \{\mathcal{H}_t(dt)\}$
- Case $d \neq dt(k)$ (transition to a different state)
  1. Let:
     - $\mathit{update}(k, d, dt(k), dt)$ be dt'. (the table after the transition)
     - $\sum_{k' \in K}(dt(k') = dt(k))$ be c. (the number of processes in the source state d)
     - $\sum_{k' \in K}(dt(k') = d)$ be c'. (the number of processes in the destination state d)
  2. Trivially $c > 0$, and by (I1) $c' < N$
  3. $\sum_{k' \in K}(dt'(k') = dt(k)) = c - 1$
  4. $\sum_{k' \in K}(dt'(k') = d) = c' + 1$
  5. Let P be $\mathit{Pred}(\mathcal{H}_t(dt), dt(k))$ and S be $\mathit{Succ}(P, d)$.
  6. By definition $P = \mathcal{H}_t(dt(k))[c := \mathit{Pred}(c)]$. Then $c > 0$ implies $P = \mathcal{H}_t(dt(k))[c := c - 1]$
  7. By definition $S = P[c := \mathit{Succ}(c)]$. Then $c < N$ implies $S = P[c := c + 1]$
  8. Then by (3, 4) $S = \mathcal{H}_t(dt')$
  9. Considering $\mathit{abs\_update}(k, d, dt(k), \mathcal{H}_t(dt)) = \{S\}$, and by (8) it follows that $\mathit{abs\_update}(k, d, dt(k), \mathcal{H}_t(dt)) = \{\mathcal{H}_t(dt')\}$
- In both cases, we obtain:
  $\mathcal{H}_t(\mathit{update}(k, d, dt(k), dt)) \in \mathit{abs\_update}(k, d, dt(k), \mathcal{H}_t(dt))$

□ (Theorem)

This abstraction can be used for the verification of properties that do not depend on the exact process that executes an action, but only depends on whether there is a process that executes it or not.

4.3.2 Abstraction of the State Counter

We can generate a more abstract version of the system by abstracting the counter. Instead of storing the exact number of processes that are in a determined state we can just consider some specific cases, for example: (a) There is no process in a certain state. (b) All processes are in a certain state. (c) There are some (but not all) processes in a certain state (assuming $N > 1$).

To perform this abstraction, we define an abstract counter *abs_count* by specifying a new mapping $\mathcal{H}_c : Count \rightarrow abs\_count$. The sort *abs_count* has three values: *zero, some* and *all*. Together, we define two functions *abs_succ*, *abs_pred*: *abs_count*$\rightarrow \mathcal{P}(abs\_count)$ to increase and decrease an abstract counter.

$$\begin{aligned}
\mathcal{H}_c(c) &=_{def} \begin{cases} all & \text{if } c = N \\ some & \text{if } 0 < c < N \\ zero & \text{if } c = 0 \end{cases} \\
abs\_succ(zero) &=_{def} \{some\} \\
abs\_pred(zero) &=_{def} \{zero\} \\
abs\_succ(some) &=_{def} \{some, all\} \\
abs\_pred(some) &=_{def} \{zero, some\} \\
abs\_succ(all) &=_{def} \{all\} \\
abs\_pred(all) &=_{def} \{some\}
\end{aligned}$$

The function *abs_match* is an auxiliary function used to check whether a process in a certain state based on the information of the abstract counter. It corresponds to the function *match* for *Count*.

$$\begin{aligned}
abs\_match(zero) &=_{def} \{\mathsf{F}\} \\
abs\_match(some) &=_{def} \{\mathsf{T}, \mathsf{F}\} \\
abs\_match(all) &=_{def} \{\mathsf{T}\}
\end{aligned}$$

In our second instance of the general framework, we redefine *abs_DTable* as a data type with the signature $D \rightarrow abs\_count$. Accordingly, *abs_dt(d)* expresses the abstract number of processes which are in state *d*. *abs_update* is a function to update the number of processes in one state. The definitions of the new functions are as the ones defined in the previous section, the only difference is that we replace the concrete functions for the counter by abstract ones:

$$\begin{aligned}
abs\_test(k, d, abs\_dt) &=_{def} abs\_match(abs\_dt(d)) \\
Succ(abs\_dt, d) &=_{def} abs\_dt[d := abs\_succ(abs\_dt(d))] \\
Pred(abs\_dt, d) &=_{def} abs\_dt[d := abs\_pred(abs\_dt(d))] \\
abs\_update(k, d, e, abs\_dt) &=_{def} \{Succ(Pred(abs\_dt, e), d)\}
\end{aligned}$$

abs_test is used to check if a given process is in a certain state, it checks the state counter by using the auxiliary function *abs_match*. *Succ* and *Pred* compute respectively the abstract successor and predecessor of an state counter. *abs_update* updates the abstract table. The new table is a more abstract version of the previous one. The abstract mapping $\mathcal{H}_{t_c} : DTable \rightarrow abs\_DTable$, is the combination of the mappings $\mathcal{H}_t$ and $\mathcal{H}_c$:

$$\mathcal{H}_{t_c}(dt)(d) =_{def} \mathcal{H}_c(\mathcal{H}_t(dt)(d))$$

Theorem 4.3.2 The abstract table with abstract counters constructed using the mapping $\mathcal{H}_{t_c}$, defines a *safe* abstraction.

Proof: Considering the result of Theorem 4.3.1, the two *safety* requirements for the functions *abs_test* and *abs_update* reduce to proving that $\forall c : Count$ the following conditions hold:

$$\begin{array}{rl} \mathcal{H}_c(Succ(c)) \in & abs\_succ(\mathcal{H}_c(c)) \\ \mathcal{H}_c(Pred(c)) \in & abs\_pred(\mathcal{H}_c(c)) \\ match(c) \subseteq & abs\_match(\mathcal{H}_c(c)) \end{array}$$

We start with the first condition:

1. $c = 0$.
   (a) $Succ(c) = 1$ and $\mathcal{H}_c(1) = some$
   (b) $\mathcal{H}_c(0) = zero$ and $abs\_succ(zero) = \{some\}$
2. $0 < c < N$.
   (a) $Succ(c)$ is either N or $0 < Succ(c) < N$, therefore $\mathcal{H}_c(Succ(c)) = all$ or $some$.
   (b) $\mathcal{H}_c(c) = some$ and $abs\_succ(some) = \{some, all\}$
3. $c = N$.
   (a) $Succ(N) = N$ and $\mathcal{H}_c(N) = all$
   (b) $abs\_succ(all) = \{all\}$

The second condition for the predecessor function can be proved in the same way. For the third condition:

1. $c = 0$.
   (a) $match(0) = \{\mathsf{F}\}$
   (b) $\mathcal{H}_c(0) = zero$ and $abs\_match(zero) = \{\mathsf{F}\}$
2. $0 < c < N$.
   (a) $match(c) = \{\mathsf{T}, \mathsf{F}\}$
   (b) $\mathcal{H}_c(c) = some$ and $abs\_match(some) = \{\mathsf{T}, \mathsf{F}\}$
3. $c = N$.
   (a) $match(N) = \{\mathsf{T}\}$
   (b) $\mathcal{H}_c(N) = all$ and $abs\_match(all) = \{\mathsf{T}\}$

□ (Theorem)

This pattern is more abstract than the previous, therefore it will preserve less information. The next figure represents graphically the idea of the abstract pattern.

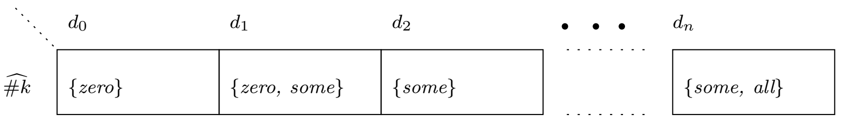


Figure 4.2: Abstract table 2: *by an abstract counter*

We remark, again, that the abstraction patterns are just examples of instances that match the general framework provided in section 4.2. Depending on the system other values for the abstract counter may be selected, for example the domains $\{zero, one, more\}$ or $\{zero, more, zero\_or\_more\}$ used in Ip and Dill's work [76] are easily embedded in our framework. This abstraction pattern may be also combined with the previous one by only abstracting the counters related to some specific states and leaving the others as natural counters.

4.4 Parallel Identical Processes

Linearization. Section 4.2 was dedicated to the linearization and abstraction of *uniform* processes. We have defined uniform processes as the ones that share the same specification, i.e., they are syntactically the same. Each process has assigned an unique identity. Even if two processes are syntactically the same, their behaviour may be different because of their identity.

We consider a particular case of *uniform* processes which are indistinguishable. We call this class of *identical* processes. The behaviour of each process does not depend on its own identity k. They share the following linear form:

$$\begin{array}{rl} P(d:D) = & \sum_{i\in I}\sum_{e_i:E_i} a_i(f_i(d,e_i)).P(g_i(d,e_i)) \\ & \triangleleft c_i(d,e_i) \triangleright \delta \end{array} \qquad (^{**})$$

Given two *identical* processes p_i and p_j that are in the same state d. If a condition c is true for p_i, then it is also true for p_j. Furthermore, they can execute the same action to end in the same new state.

If processes are identical then the mapping of section 4.3.1 does not lose information. Therefore, the concrete system is composed by a table that stores the number of processes that are in a certain state. We redefine the concrete table *DTable* of the signature $D \to Count$. $dt(d)$ yields the number of processes in a certain state d. Accordingly, we redefine the functions *update* and *test*.

$$\begin{array}{rl} update: & D \times D \times DTable \to DTable \\ test: & D \times DTable \to Bool \end{array}$$

The defining equations are:

$$\begin{aligned} test(d,dt) &=_{def} \begin{cases} \mathsf{T} & \text{if } dt(d) > 0 \\ \mathsf{F} & \text{if } dt(d) = 0 \end{cases} \\ Succ(dt,d) &=_{def} \quad dt[d := Succ(dt(d))] \\ Pred(dt,d) &=_{def} \quad dt[d := Pred(dt(d))] \\ update(d,e,dt) &=_{def} \quad Succ(Pred(dt,e),d) \end{aligned}$$

Then we have the following theorem:

Theorem 4.4.1 Given a process P as defined in (**), the system $P(d_0) \; || \; P(d_1) \; || \; \cdots \; || \; P(d_{N-1})$ is strongly bisimilar to $Q(dt)$, where dt is equal to $[d_0, d_1, ..., d_{N-1}]$ and Q is an LPE of the following form:

$$\begin{aligned} Q(dt : DTable) = \quad & \textstyle\sum_{i\in I} \sum_{d:D} \sum_{e_i:E_i} a_i(f_i(d,e_i)). \\ & Q(update(g_i(d,e_i),d,dt)) \\ & \lhd test(d,dt) \wedge c_i(d,e_i) \rhd \delta \end{aligned}$$

Proof: As in Theorem 4.2.1, ξ denotes the state vector $[d_0 \times \ldots \times d_{N-1}]$ of processes $P(d_0) \; || \; P(d_1) \; || \cdots || \; P(d_{N-1})$. Now, we define the relation $\mathcal{R}$:

$$dt \, \mathcal{R} \, \xi \iff \forall \, d \in D. \quad dt(d) = \textstyle\sum_{k\in K} (\xi[k] = d)$$

Initially, is trivial to see that $[d_0, d_1, ..., d_{N-1}] \, \mathcal{R} \, [d_0 \times \ldots \times d_{N-1}]$. Then, to prove that the parallel composition of N processes (from now on S) is bisimilar to Q, we have to prove that if from a state ξ, S can do a step to ξ' and ξ is related to dt, then $Q(dt)$ can do the same action to dt' and $\xi' \; \mathcal{R} \; dt'$, and the other way around. First, it is straightforward to verify the following invariant (I2) of Q:

$$test(d,dt) \iff \exists \, k \in K. \, d = \xi[k]$$

Then, we consider the first implication:

1. If $S = P(\xi[1]) \; || \; P(\xi[0]) \; || \cdots || \; P(\xi[N-1])$ can do a a-step (a step labelled with a) then there exists one k such that $P(\xi[k])$ can do an a-step.

2. If $P(\xi[k])$ does an a-step then there exists a summand i and an e such that $c_i(\xi[k], e) = \mathsf{T}$ and $a = a_i(f_i(\xi[k], e))$.

3. If $P(\xi[k])$ performs the step then ξ' will be $\xi[\xi[k] := g_i(\xi[k], e)]$.

4. Let c and c' be the number of processes in $\xi[k]$ before and after the step, and n and n' the number of processes in $g_i(\xi[k], e)$ before and after the step. Is trivial to see that:

   - If $\xi[k] \neq g_i(\xi[k], e)$ then $c' = c - 1$ and $n' = n + 1$.
   - If $\xi[k] = g_i(\xi[k], e)$ then $c' = c$.

5. By the (I2), $test(\xi[k], dt) = \mathsf{T}$. Hence, the condition of the ith summand of Q will be true.
6. dt' will be equal to $update(g_i(\xi[k], e), d, dt)$, in other words dt' will be equal to $Succ(Pred(dt, d), g_i(\xi[k], e))$.
7. By Definition of $Succ$ and $Pred$ and by (4), it follows that $\xi' \mathcal{R}\, dt'$ which proves the case.

The other implication is proved in the same way. Hence, S and Q are bisimilar.

□ (Theorem)

Instead of storing the state of every process, we have used a counter representing the number of processes that are in a certain state and we have proved that both representations are equivalent (are strongly bisimilar).

This representation of the composition of processes basically removes all the symmetrical paths that may be generated by the interleaving of the behaviour of the different processes.

Abstraction. In order to abstract the system with *identical* processes, we can trivially adapt the definitions given for the case of *uniform* processes. In this case the abstraction would consist of abstraction of the counters, therefore, one may use, for example, the pattern provided in section 4.3.2, in which the counter is abstracted to some symbolic values that determine the abstract number of processes that are in a certain state.

4.5 General Form

We have assumed that the uniform processes in parallel do not communicate between each other but only with external processes. In order to generalise the framework to any kind of uniform processes, we have to extend the definitions by allowing direct communication via an internal action $\gamma(a_{i_1}, a_{i_2})$ parameterized with data. The following equation (GF) generalises definition 4.2.

The abstraction framework should be extended to handle the new form. A similar definition can be given for the case of identical processes.

4.6 Applications

In this section, we shortly introduce a case study that uses the results of the chapter. Another case study is included in Chapter 8.

$$
\begin{aligned}
& Q(dt : DTable) = \\
& \quad \textstyle\sum_{i \in I} \sum_{k:Nat} \sum_{d:D} \sum_{e_i:E_i} a_i(f_i(k,d,e_i)). \\
& \quad Q(update(k, g_i(k,d,e_i), d, dt)) \\
& \quad \lhd test(k,d,dt) \wedge c_i(k,d,e_i) \wedge k < N \rhd \delta + \\
& \quad \textstyle\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1:K} \sum_{k_2:K} \sum_{d_1:D} \sum_{d_2:D} \sum_{e_1:E_i} \sum_{e_2:E_i} \\
& \qquad \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1,d_1,e_1)). \\
& \quad Q(update(k_2, g_{i_2}(k_2,d_2,e_2), d_2, update(k_1, g_{i_1}(k_1,d_1,e_1), d_1, dt))) \\
& \quad \lhd test(k_1,d_1,dt) \wedge c_{i_1}(k_1,d_1,e_1) \wedge k_1 < N \wedge \\
& \qquad test(k_2,d_2,dt) \wedge c_{i_2}(k_2,d_2,e_2) \wedge k_1 > k_2 \wedge \\
& \qquad f_{i_1}(k_1,d_1,e_1) = f_{i_2}(k_2,d_2,e_2) \rhd \delta
\end{aligned}
\tag{GF}
$$

4.6.1 A Distributed System for Lifting Trucks

A real-life distributed system for lifting trucks (lorries, railway carriages, buses and other vehicles), which was designed and implemented by a Dutch company, was analysed in μCRL together with CADP by Groote et al. [60].

The system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own micro-controller. On each lift there are some buttons that control its movement. The micro-controllers of the different lifts belonging to a system are connected to a ‘cyclic’ CAN (Controller Area Network). The formal analysis of the system discovered some errors in the original specification and helped to build a refined version of the incorrect implementation. The new specification could be proved correct for small instances of the system (at most 5 lifts). In [100] we have extended the analysis of some requirements to an arbitrary number of lifts. Basically, we first give a simplified version of the original specification that removes some details about the initialisation phase, then we apply the previously introduced techniques (see section 4.3.1 and section 4.3.2).

If the *up* button of a certain lift is pressed, all the lifts of the system should go up. The system has to assure that all lifts move simultaneously to the same direction. Lifts are programmed in such a way that during normal operation, they take turns to claim the bus. To achieve this orderly usage of the CAN bus, each lift must know its position in the network. Furthermore, in order to be able to find out whether all lifts are in the same state, each lift must know how many lifts there are in the network.

Initially, all lifts are in a *standby* state. The state of a lift is changed if its *up* button is pressed, then it will send an *up* message to the bus. Other lifts change their state according to the messages they receive, and when it is their turn to use the bus they broadcast a message according to their state. These messages are received by all the other lifts, and the lift where a button is pressed will count them. When it counts enough state messages, it will broadcast a *move* message when it gets the turn to use the bus, after which all the lifts will synchronously move. The state of each lift is the vector composed by: the identifier of the lift, which determines the order to claim the bus, the current state of the lift and a counter for synchronised lifts which ranges from 0 to N. The behaviour of the lifts also depends on the messages passed around, which are composed by: the identity, the state of the sender of the last message and a boolean specifying when the up-button was pressed. The complete system is composed by the parallel composition of the N lifts and the process that models the CAN bus.

In order to create an abstraction that proves properties for an arbitrary number of lifts, we have to combine the abstraction of processes as proposed in this paper with classical data abstraction. Therefore, we first abstract all the parameters and local variables that depend on N. Then, we see that the behaviour of the lifts depends on the process identifier so we have to use the abstraction pattern for *uniform* processes. Therefore, we construct the abstract table with abstract counters for every state using the abstraction pattern in section 4.3.2.

We have built such abstraction using the abstraction assistant for μCRL specifications presented in the previous chapter. The result of the abstraction is a *Modal*-LTS. We see below the comparison between the abstract result and some concrete instances.

System	States
Crt 5 Lifts	2,751
Crt 6 Lifts	10,011
Crt 7 Lifts	33,031
Crt 8 Lifts	101,255
Abs N Lifts	1341

Formally, we express a correctness criterion by the following *safety* property:

(P2): $[\ '.\ *_{\mathsf{may}}\ .*'*.\ '\mathsf{move}_{\mathsf{may}}'\ .\neg\ ('\mathsf{up}_{\mathsf{may}}')*\ .\ '\mathsf{move}_{\mathsf{may}}'\ ]\ \mathsf{F}$

Basically, the formula states that after a movement of the lift system, a button should be pressed in order to let the system move again. The *safety* formula is satisfied by the abstract system for an arbitrary number of processes N, therefore we can infer its satisfaction to all instances of the concrete system.

In this example we have seen that in some cases the abstraction patterns together with regular data abstraction may be used to generalise the model checking problem to an arbitrary number of uniform components. In order to prove properties for N processes, we had to abstract from quite a lot of information. Consequently, we were not able to verify interesting *liveness* properties for the abstract specification.

4.7 Conclusion

In this chapter, we have developed a framework to abstract a subclass of systems consisting of a set of (in)distinguishable replicated processes composed in parallel. First, we give a linear form, which differs from the standard form proposed for μCRL in [61, 117], to capture the behaviour of the full system. The special linear form encodes the state of replicated processes by using a table, in other words processes are specified using data. Once we have the suitable linear form we can apply the results of abstract interpretation described in the two first chapters:

- We transform the concrete linear form to the corresponding *Modal*-LPE, abstracting and lifting the functions that manipulate the table.
- We generate the safety conditions associated to the abstracted functions.
- We give two abstract definitions of the table that satisfy the safety conditions and, therefore, generate correct *Modal*-LTS of a given concrete system.

We believe that identifying classes of systems and providing suitable abstractions for them is a desirable way to help the system designer to integrate the abstract interpretation theories in his verification methodology. The patterns described in this chapter will not help in all cases, but they do apply in some practical cases.

Chapter 5

Accelerated Modal Abstractions

This chapter is dedicated to explaining an extension to the abstraction framework presented in the first chapter. We enhance *Modal Labelled Transitions Systems* by allowing transitions to be labelled with regular expressions. This permits to represent that a process can reach a state by executing a sequence of actions, abstracting the intermediate states. We show how using, what we call, *accelerated*-transitions we can improve the expressiveness of the abstractions being able to prove more *liveness* properties.

5.1 Introduction

In the first chapter, we have reasoned about the benefits of extending the classical frameworks of abstraction by allowing the abstraction of action labels. The improvements allowed to generate more expressive abstractions and to deal with infinitely branching systems. Actions were abstracted by mapping (or relating) concrete instances to abstract ones. In this chapter we are going to consider a more powerful way of abstracting actions. The idea is to abstract complete sequences of transitions labelled by concrete actions to single transitions labelled by abstract sequences of actions.

Abstraction has been successfully used to prove mainly safety properties. Even though some frameworks allow the inference of liveness properties their verification remains one of the major challenges of abstraction theories [102]. The problem comes from the extra behaviours added by the abstractions. We start by giving an example to motivate the extension.

Figure 5.1 represents the classical abstraction of a decreasing counter. It represents an over-approximation in which the values greater or equal to 1 are collapsed to a single abstract state: '+':

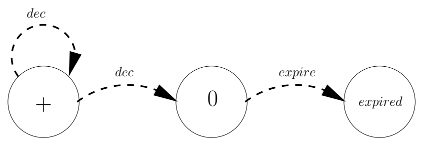


Figure 5.1: Abstraction of a *decreasing* counter.

From the abstract system we cannot infer that, in all traces, the action *expire* is *eventually* executed. This property is trivially true for the concrete system for any initial value of the counter, because the only possible action is the one that decreases the value of the counter. The reason why the property is not true in the abstract system, is the existence of abstract loop: $+ \rightarrow +$, that does not correspond to an infinite concrete computation.

In general, abstraction introduces non-determinism to the system which causes the loss of information. The problem of how to improve the expressiveness of such type of abstractions has already been addressed by other authors. An interesting approach is, for example, the one proposed by Pnueli, in [79, 41]. His idea is to impose fairness constraints to the abstract system in order to remove undesirable behaviours. The fairness constraints are extracted from the knowledge we have of the concrete system. We know that the concrete system does not contain any infinite decreasing trace, hence the abstract should not have it either. In the previous example, we can infer the following constraint:

"For any fair trace, if the transition $+ \rightarrow 0$ *is infinitely often enabled then it should be infinitely often taken."*

Any fair computation of the abstract system will not contain an infinite loop $+ \rightarrow +$. Therefore, under the constraint, we are able to prove that *expired* is reached. This approach is valid to remove non-progressing traces. It has been used to infer properties coded in LTL by Pnueli and also recently by Bosnacki et al. [18]. In the latter approach the authors proved that in some specific cases, such as the counter abstraction, strong fairness constraints can be reduced to weak fairness which are more efficiently handled by model checkers.

Here, we propose an alternative approach motivated by the following example. Let us consider a slightly more complex system: a *resettable* counter (which Pnueli's approach cannot handle). The following figure represents an over-approximation of a counter that might be reset to its initial value (that is bigger than 0), in case the counter is not expired:

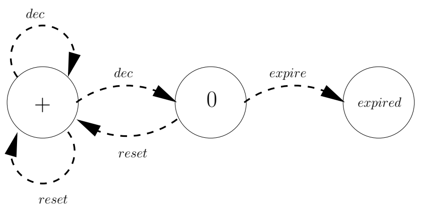


Figure 5.2: Abstraction of a *resettable* counter.

In this case the property *"in all traces,* expire *is eventually executed"* is true neither in the abstract system nor in the concrete. However, we can formulate an interesting property of the system, such as, *"for any reachable non-*expired *state there is path that leads to* expire*"*. This property cannot be expressed in LTL, we need some logic that captures the branching behaviour of the system. Furthermore, to reason about the abstract system we cannot impose the above presented fairness constraint because the abstract loop: $+ \rightarrow +$, in this case, does correspond to some concrete loop. To deal with such systems, we are going to use a different approach. The idea we propose in this chapter is to include a new kind of transitions that represent finite computations, which will allow to infer stronger progress properties.

We recall that to model abstractions we have used *Modal Labelled Transition Systems* which have two modalities: *may* and *must*. On the one hand, the *may* part corresponds to an over-approximation that preserves *safety* properties of the concrete instance and on the other hand the *must* part under-approximates the model and reflects *progress* properties. Here, we enhance MLTSs by allowing *must*-transitions to match sequences of actions, which captures the idea that in a finite computation a state can be reached from a given one.

We define a new type of structure *accelerated-Modal Labelled Transition Systems* in which *must*-transitions are labelled with regular expressions built over the action labels and the usual operators. We present this idea in the next section. Then, we present how to generate abstractions with *accelerations* and we give the preservation results. We prove that the framework is sound and complete for a logic based on the Propositional Dynamic Logic (PDL [68]) which is a branching logic, in the style of HML [71] with regular expressions, less expressive than μ-calculus [114].

Section 5.5 is dedicated to practical issues, defining a model checking algorithm for *accelerated*-MLTS and PDL properties. This chapter is mainly focused on the semantic level, nevertheless at the end we give some hints about how to extract *accelerated* abstraction from specifications.

5.2 Accelerated Transition Systems

Section 1.2, we have introduced two different structures to model the behaviours of systems: LTSs and *Modal*-LTSs. The first one was used to describe concrete semantics and the second one abstract semantics. We recall the definition of the latter:

Definition 5.2.1 A *Modal Labelled Transition System* (MLTS) is a tuple *(S, Act,* $\rightarrow_\Diamond, \rightarrow_\Box, s_0$) where S is a non-empty set of states, *Act* is a non-empty set of action labels and s_0 is the initial state and $\rightarrow_\Diamond, \rightarrow_\Box$ are possibly infinite sets of (may or must) transitions of the form $s \xrightarrow{a}_x s'$ with s,$s' \in S$, $a \in Act$ and $x \in \{\Diamond, \Box\}$. We require that every *must*-transition is a *may*-transition ($\xrightarrow{a}_\Box \subseteq \xrightarrow{a}_\Diamond$).

We enhance *Modal*-LTSs by changing the definition of *must*-transitions. We call *accelerated must*-transitions, those transitions that condense a sequence of steps in a single one. *accelerated must*-transitions will be labelled by regular expressions σ built over the alphabet *Act* and the usual operators $\cdot, *$ and $|$. It can be also used $\sigma+$, which is an abbreviation for $\sigma \cdot \sigma*$. Let us see the definition:

Definition 5.2.2 An *Accelerated Modal Labelled Transition System* (*accModal*-LTS) is a tuple $(S, Act, \rightarrow_\Diamond, \rightarrow_\boxplus, s_0)$ where S, *Act* and s_0, $\rightarrow_\Diamond$ are as in the previous definition, and $\rightarrow_\boxplus$ is a possibly infinite set of *accelerated*-must transitions of the form $s \xrightarrow{\sigma}_\boxplus s'$ with s,$s' \in S$, and σ is a regular expression. Furthermore, we require:

- Every *accelerated must*-transition corresponds to a finite sequence of *may*-transitions, i.e.:

$$s \xrightarrow{\sigma}_\boxplus s' \implies \exists\, a_0, ..., a_i.\, s \xrightarrow{a_0}_\Diamond \, ... \, \xrightarrow{a_i}_\Diamond s' \wedge [a_0 \cdots a_i] \in [\![\sigma]\!]^1$$

Basically, the new definition allows *must*-transitions to be labelled with arbitrary regular expressions. Examples of correct *accelerated* transitions are:

- $s \xrightarrow{a+}_\boxplus s'$

^{1}We denote by $[\![\sigma]\!]$ the language generated from σ, i.e. $\mathcal{L}(\sigma)$

- $s \xrightarrow{a|b}_{\boxplus} s'$
- $s \xrightarrow{a.b*.a}_{\boxplus} s'$

A trivial result is that every *Modal*-LTS is an *accModal*-LTS. It follows from the fact that every *must* transition is an *accelerated must*-transition in which σ is equal to a single action label. The condition that every *must*-transition corresponds to a sequence of *may*-transitions, is similar to the one imposed in the MLTS and, as we will see, it will help to define the logical characterisation of the abstractions.

The next figure[2] presents a modal abstraction of the *resettable* counter with *accelerated* transitions. Note that the difference with the abstraction that we could have generated using the theory of the first chapters is the presence of the accelerated transitons $+ \xrightarrow{dec+}_{\boxplus} 0$. The next section is dedicated to explaining how such an abstraction can be defined from a concrete system.

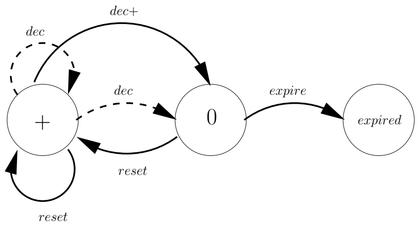


Figure 5.3: Resettable counter with one *accelerated must*-transition

5.3 Accelerated Modal Abstractions

This section is dedicated to explaining how to define abstract approximations of concrete transition systems. From a concrete system described by an LTS, we can generate an abstraction by relating concrete states with abstract states. Pnueli and Bosnacki et al. use function mappings to define abstractions. As we have seen, this approach was suggested by Clarke and Long [26, 36]. An alternative approach suggested by Cousot and Cousot [29] is based on Galois Connections, which allow concrete states to be related to more than one abstract state. The Galois Connection framework gives, in general, more accurate abstractions than the homomorphic approach. In this chapter, we are going to work with simple mappings, but we believe that the extension to more complicated relations can be done following the ideas of Chapter 1 . Furthermore, we do not consider, in this chapter, the abstraction of action labels.

[2]We use the following representation in the figures: *may* transitions (dashed lines), *accelerated must*-transitions (solid lines). The *may* transitions corresponding to *must*-transitions are not drawn.

Let us assume a set of abstract values $\widehat{S}$ and an abstraction function $h : S \rightarrow \widehat{S}$ that is total and surjective. An abstract value $\widehat{s}$ corresponds to all the states s for which $h(s) = \widehat{s}$. We define the way of generating an abstraction from a concrete system, as follows:

Definition 5.3.1 Given a *concrete* LTS, $\mathcal{M}$ $(S, Act, \rightarrow, s_0)$ and a mapping $h : S \rightarrow \widehat{S}$, we say that the *accModal*-LTS, $\widehat{\mathcal{M}}$ defined by $(\widehat{S}, Act, \rightarrow_{\diamond}, \rightarrow_{\boxplus}, \widehat{s}_0)$ is the minimal abstraction by h (denoted by $\widehat{\mathcal{M}} = min_h(\mathcal{M})$) if and only if $h(s_0) = \widehat{s}_0$ and the following conditions hold:

- $\widehat{s} \xrightarrow{a}_{\diamond} \widehat{r} \iff \exists\, s, r.\, h(s) = \widehat{s} \wedge h(r) = \widehat{r} \wedge s \xrightarrow{a} r$
- $\widehat{s} \xrightarrow{\sigma}_{\boxplus} \widehat{r} \iff (\forall\, s.h(s) = \widehat{s} \implies \exists\, r, a_0, ..., a_i.\, h(r) = \widehat{r} \wedge s \xrightarrow{a_0} ... \xrightarrow{a_i} r \wedge [a_0 \cdots a_i] \in [\![\sigma]\!])$

$[a_0 \cdots a_i] \in [\![\sigma]\!]$ means that the concatenation of $a_0, ..., a_i$ belongs to the language generated by the regular expresion σ.

A transition $\widehat{s} \xrightarrow{\sigma}_{\boxplus} \widehat{r}$ means that for all s mapped to $\widehat{s}$ there exists an r mapped to $\widehat{r}$ such that we can go from s to r by a word contained in the language generated from σ.

This definition gives the most accurate abstraction of a concrete system for a given homomorphism h, in other words the one that preserves as much information as possible of the original system. Now, we define the notion of approximation to characterise the non-minimal abstractions:

Definition 5.3.2 Given two *accModal*-LTSs, $\widehat{\mathcal{M}}$ $(\widehat{S}, Act, \rightarrow_{\diamond}, \rightarrow_{\boxplus}, \widehat{s}_0)$ and $\widehat{\mathcal{N}}$ $(\widehat{S}, Act, \rightarrowtriangle_{\diamond}, \rightarrowtriangle_{\boxplus}, \widehat{s}_0)$ built over the same sets of states and actions; $\widehat{\mathcal{N}}$ is an approximation of $\widehat{\mathcal{M}}$, denoted by $\widehat{\mathcal{M}} \sqsubseteq \widehat{\mathcal{N}}$, if the following conditions hold:

- $\widehat{s} \xrightarrow{a}_{\diamond} \widehat{r} \implies \widehat{s} \overset{a}{\rightarrowtriangle}_{\diamond} \widehat{r}$
- $\widehat{s} \overset{\sigma'}{\rightarrowtriangle}_{\boxplus} \widehat{r} \implies \exists\, \sigma_0, ..., \sigma_i.\, \widehat{s} \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} \widehat{r} \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\sigma']\!]$

The relation $\sqsubseteq$ characterises an order on the precision of the abstractions. $\widehat{\mathcal{M}}$ is more precise than $\widehat{\mathcal{N}}$ because it has less (or the same number of) *may*-transitions and more *accelerated must*-transitions or more precise ones. For example, considering only one *accelerated must*-transition $\widehat{s} \xrightarrow{\sigma}_{\boxplus} \widehat{r}$, we have:

$$\widehat{s} \xrightarrow{a}_{\boxplus} \widehat{r} \sqsubseteq \widehat{s} \xrightarrow{a+}_{\boxplus} \widehat{r} \sqsubseteq \widehat{s} \xrightarrow{a*}_{\boxplus} \widehat{r} \sqsubseteq \widehat{s} \xrightarrow{a*|b}_{\boxplus} \widehat{r} \sqsubseteq \widehat{s} \not\rightarrow_{\boxplus} \widehat{r}$$

Note, that the last case means that there is not any *must* transition between $\widehat{s}$ and $\widehat{r}$. It is a correct abstraction even if it does not contain a lot of information. Another simple example would be:

$$\widehat{s} \xrightarrow{a}_{\boxplus} \widehat{t} \xrightarrow{b}_{\boxplus} \widehat{r} \sqsubseteq \widehat{s} \xrightarrow{a \cdot b}_{\boxplus} \widehat{r}$$

So far, we have discussed the approximation relation between a concrete system modeled by an LTS and an abstraction modeled by *accelerated Modal*-LTS. Now, we are going to present some results about the preservation of properties between them.

5.4 Logical Characterisation

We are going to use a logic based on the propositional dynamic logic (PDL) to characterise the properties that can be inferred from abstract systems to concrete ones. We consider three types of formulas, action (α), regular (β) and state formulas (φ), expressed by the following grammars:

$$\begin{aligned}
&\alpha ::= \mathsf{T} \mid \mathsf{F} \mid \neg\,\alpha \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid a\\
&\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1 | \beta_2 \mid \beta *\\
&\varphi ::= \mathsf{T} \mid \mathsf{F} \mid \neg\,\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\beta]\varphi \mid \langle\beta\rangle\varphi
\end{aligned}$$

Note that this logic is as a subset of the one presented in Chapter 3, section 3.5. We recall the meaning of the formulas: a stands for an action label, it matches transitions with the same action label. T matches all actions, $\neg\,\alpha$ matches all actions but the ones matched by α. F matches no action, it could have been expressed by $\neg\,\mathsf{T}$.

Regular formulas match sequences of actions; '$\cdot$' stands for the concatenation operator, '$|$' is the choice operator, '$*$' is the transitive and reflexive closure operator. Note that α is used to represent both a regular formula with only one action and an action formula.

The semantics of the state formulas is standard. $[\beta]\varphi$ holds in a state in which all continuations by sequences matching β end in a state satisfying φ. $\langle\beta\rangle\varphi$ holds in a state in which exists at least one β sequence to a state satisfying φ.

We have chosen this logic because of the fact that it is also built over regular formulas, which makes it easier to manage in the abstraction framework we have presented.

As in the first chapter, a state formula is interpreted dually over *accModal*-LTSs, i.e. there will be two sets of states that satisfy it. A set of states that *necessarily* satisfy the formula and a set of states that *possibly* satisfy it. Thus, the semantics of the formulas is given by $[\![\varphi]\!] \in 2^S \times 2^S$ and the projections $[\![\varphi]\!]^{nec}$ and $[\![\varphi]\!]^{pos}$ give the first and the second component, respectively.

First, we present the semantics for *necessary* interpretation. We start with the state formulas in which basically the modality is pushed inwards in all the operators, and inverted for the negation:

$$\begin{aligned}
[\![\mathsf{T}]\!]^{nec} &= S \\
[\![\mathsf{F}]\!]^{nec} &= \emptyset \\
[\![\neg\,\varphi]\!]^{nec} &= S \setminus [\![\varphi]\!]^{pos} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{nec} &= [\![\varphi_1]\!]^{nec} \cap [\![\varphi_2]\!]^{nec} \\
[\![\varphi_1 \vee \varphi_2]\!]^{nec} &= [\![\varphi_1]\!]^{nec} \cup [\![\varphi_2]\!]^{nec} \\
[\![[\beta]\varphi]\!]^{nec} &= \{s \mid \forall\, r, a_0, ..., a_i.\, s \xrightarrow{a_0}_{\Diamond} \cdots \xrightarrow{a_i}_{\Diamond} r \wedge \\
&\qquad\qquad [a_0 \cdots a_i] \in [\![\beta]\!] \implies r \in [\![\varphi]\!]^{nec}\} \\
[\![\langle\beta\rangle\varphi]\!]^{nec} &= \{s \mid \exists\, r, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \wedge \\
&\qquad\qquad [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \,\wedge\, r \in [\![\varphi]\!]^{nec}\}
\end{aligned}$$

By the definition of negation if a system *necessarily* satisfies a negated property then it does not *possibly* satisfy the property (in positive form), i.e., $[\![\neg\varphi]\!]^{nec} = \neg[\![\varphi]\!]^{pos}$. The *possibly* semantics is dual, we just present it for the box and diamond operators (note the swap of the modalities):

$$\begin{aligned}
[\![[\beta]\varphi]\!]^{pos} &= \{s \mid \forall\, r\,, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \wedge \\
&\qquad\qquad [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \implies r \in [\![\varphi]\!]^{pos}\} \\
[\![\langle\beta\rangle\varphi]\!]^{pos} &= \{s \mid \exists\, r, a_0, ..., a_i.\, s \xrightarrow{a_0}_{\Diamond} \cdots \xrightarrow{a_i}_{\Diamond} r \wedge \\
&\qquad\qquad [a_0 \cdots a_i] \in [\![\beta]\!] \,\wedge\, r \in [\![\varphi]\!]^{pos}\}
\end{aligned}$$

Now, we give the semantics for the regular formulas:

$$\begin{aligned}
[\![\alpha]\!] &= \{[a] \mid a \in [\![\alpha]\!]\} \\
[\![\beta_1 \cdot \beta_2]\!] &= [\![\beta_1]\!] \circ [\![\beta_2]\!] \\
[\![\beta_1 | \beta_2]\!] &= [\![\beta_1]\!] \cup [\![\beta_2]\!] \\
[\![\beta *]\!] &= [\![\beta]\!] *
\end{aligned}$$

Now, we give the semantics for the action formulas $[\![\alpha]\!]$:

$$\begin{aligned}
[\![a]\!] &= \{a\} \\
[\![\mathsf{T}]\!] &= Act \\
[\![\mathsf{F}]\!] &= [\![\neg\,\mathsf{T}]\!] \\
[\![\alpha_1 \wedge \alpha_2]\!] &= [\![\alpha_1]\!] \cap [\![\alpha_2]\!] \\
[\![\alpha_1 \vee \alpha_2]\!] &= [\![\alpha_1]\!] \cup [\![\alpha_2]\!] \\
[\![\neg\alpha]\!] &= Act \setminus [\![\alpha]\!]
\end{aligned}$$

Lemma 5.4.1 We have the classical equivalences on state formulas:

- $\mathsf{T} = \neg\mathsf{F}$

- $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $[\![[\beta]\varphi]\!] = [\![\neg\langle\beta\rangle\neg\varphi]\!]$

We remark that the *necessary* interpretation is consistent, i.e., we cannot prove at the same time that one formula and its negation are *necessarily* satisfied. Furthermore, the *possible* semantics are complete, we can either prove a property or its negation. These two properties follow from the fact that $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$. We formally state these results:

Lemma 5.4.2 $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$

Lemma 5.4.3 The *necessary* interpretation is consistent, i.e., $[\![\varphi \wedge \neg\varphi]\!]^{nec} = \emptyset$

Lemma 5.4.4 The *possible* interpretation is complete, i.e., $[\![\varphi \vee \neg\varphi]\!]^{pos} = S$.

Now, we present the preservation results:

Theorem 5.4.5 Given two *accModal*-LTSs, $\widehat{M}$ and $\widehat{N}$, over the same sets of states and labels $\widehat{S}$ and *Act*, with $\widehat{M} \sqsubseteq \widehat{N}$ for all $\widehat{s}$ in $\widehat{S}$ and for all formula φ, we have:

- $\widehat{N}, \widehat{s} \models^{nec} \varphi \implies \widehat{M}, \widehat{s} \models^{nec} \varphi$
- $\widehat{N}, \widehat{s} \not\models^{pos} \varphi \implies \widehat{M}, \widehat{s} \not\models^{pos} \varphi$

Theorem 5.4.6 Let $\mathcal{M}$ be the LTS, $(S, Act, \rightarrow, s_0)$, h be mapping between S and $\widehat{S}$ and let the *accModal*-LTS, $\widehat{\mathcal{M}}$ $(\widehat{S}, Act, \rightarrow_{\diamond}, \rightarrow_{\boxplus}, \widehat{S}_0)$ be the minimal abstraction of ($\widehat{\mathcal{M}} = min_h(\mathcal{M})$). Then for every φ, and s and $\widehat{s}$ such that $h(s) = \widehat{s}$, we have:

- $\widehat{\mathcal{M}}, \widehat{s} \models^{nec} \varphi \implies \mathcal{M}, s \models \varphi$
- $\widehat{\mathcal{M}}, \widehat{s} \not\models^{pos} \varphi \implies \mathcal{M}, s \not\models \varphi$

The first Theorem defines the inference between different abstractions, and the second Theorem between a concrete system and its minimal abstraction. Together they can be used to infer properties from an abstract approximation to the concrete. The proofs of the Theorems are included at the end of the chapter.

If we have to prove a property φ on a concrete system $\mathcal{M}$, it is enough that an abstract approximation $\widehat{\mathcal{M}}$ *necessarily* satisfies it. If we want to refute a property, we prove that $\widehat{\mathcal{M}}$ does not *possibly* satisfies it Let us consider again the example of Figure 5.3, the property we want to prove is:

(P): $[\ (\neg\ \text{“}\mathsf{expire}\text{”})^*\ ]\ \langle\ \mathsf{T}^*.\ \text{“}\mathsf{expire}\text{”}\ \rangle\ \mathsf{T}$

which is read as: *after any sequence of actions different to* expire *there is a path that leads to* expire. We want to prove that this property is *necessarily* satisfied in the abstract system given above. We recall that the universal modality is interpreted over *may* actions and the existential over *must* ones. Hence:

- From the initial state $+$, the states that *may* be reached by ($\neg$ “expire”*) are $\{+, 0\}$. Then,
- from $+$ there is the path $\overset{\mathsf{dec+}}{\rightarrow}_{\boxplus}\overset{\mathsf{expire}}{\rightarrow}_{\boxplus}$ and $[\![\mathsf{dec+}\cdot\mathsf{expire}]\!] \subset [\![\mathsf{T}*\cdot\mathsf{expire}]\!]$, therefore $+$ satisfies $\langle\ \mathsf{T}^*.\ \text{“}\mathsf{expire}\text{”}\ \rangle\ \mathsf{T}$.
- From 0 we have the transition $\overset{\mathsf{expire}}{\rightarrow}_{\boxplus}$, and $[\![\mathsf{expire}]\!] \subset [\![\mathsf{T}*\cdot\mathsf{expire}]\!]$, therefore 0 also satisfies the $\langle\ \mathsf{T}^*.\ \text{“}\mathsf{expire}\text{”}\ \rangle\ \mathsf{T}$.
- Hence, the formula is *necessarily* satisfied in the abstraction and we can infer the satisfaction on the concrete system.

We remark that the formula cannot be proved using only the abstraction framework presented in the first chapter because there will not be a *must*-transition between $+$ and 0.

This section included the main preservation results about the logical characterisation and an intuitive example about how to apply the results to infer properties. Next section is dedicated to describe a decidable model checking algorithm that implements the semantics given above. This shows that the model checking problem for *accelerated*-MLTSs is still decidable.

5.5 Model Checking

The model checking problem is solved by two functions *eval_must* and *eval_may* both defined from *formulas* to $\mathcal{P}(S)$. They work by analysing the subformula components of the original. They are derived from the semantics presented in section 5.4. We only present *eval_must* which returns the set of states that *necessary* satisfy a formula. *eval_may* returns the set of states that *possibly* satisfy a formula and it is simpler than the other because there are no *accelerated* transitions involved.

```
function $eval\_must(\varphi)$ {
    if $\varphi = \mathsf{F}$ then return $\emptyset$;
    if $\varphi = \neg\varphi_1$ then return $S \setminus eval\_may(\varphi_1)$;
    if $\varphi = \varphi_1 \vee \varphi_2$ then return $eval\_must(\varphi_1) \cup eval\_must(\varphi_2)$;
    if $\varphi = \langle\beta\rangle\varphi_1$ then return $exists\_must(\beta, eval\_must(\varphi_1))$;
}
```

exists_must(β, R) returns the set of states that can reach a state in R by performing a sequence of actions such that the language generated by the concatenation of the action labels is included in the language generated by β. We first give an algorithm to compute this function, and then provide an explanation for it. In the algorithm we use the following notation: Given an automaton B, $B_{(i,J)}$ denotes the automaton that is obtained from B by changing the initial state to i and the final states to J.

```
function exists_must($\beta, R$){
1   $B := \mathrm{DFA}(\beta)$;
    $b_0 := \mathrm{START}(B)$;
    $F := \mathrm{FINAL}(B)$;
2   for all $\sigma$ such that $\exists\, s, r \in S.\, s \xrightarrow{\sigma}_{\boxplus} r$ do
        $R_\sigma := \{(i, J) \mid [\![\sigma]\!] \subseteq [\![B_{(i,J)}]\!]\}$
3   for all $s, r \in S$ do
        $R_{(s,r)} := \cup\{R_\sigma \mid s \xrightarrow{\sigma}_{\boxplus} r\}$
4   do {
        for all $s,\, r \in S$ do
            $R'_{(s,r)} := R_{(s,r)}$
        for all $s, r, t \in S$ do
            $R_{(s,t)} := R_{(s,t)} \cup$
                    $\{(h, J) \mid \exists\, I.\, (h, I) \in R_{(s,r)} \wedge \forall i \in I.(i, J) \in R_{(r,t)}\}$
    } while $(\exists\, s, r \in S.\, R'_{(s,r)} \neq R_{(s,r)})$
   return $\{s \mid \exists\, r \in R\, \exists\, (b_0, J) \in R_{(s,r)}.\, J \subseteq F\}$;
}
```

eval_must analyses recursively the given formula, computing for every subformula the set of states that satisfy it. *exists_must* computes the part referring the *accelerations*, let us see how the last algorithm works:

1. First, the algorithm computes a deterministic automaton (DFA) corresponding to the regular expression β. B denotes this automaton, b_0 its initial state and F the set of final states.

2. Then, for every regular expression σ of the transition system, we compute R_σ which consists of the pairs (i, J) of B, such that the language generated by σ is included in the language accepted by the automaton $B_{(i,J)}$. If (i, J) is in R_σ then all pairs (i, J') with $J \subset J'$ are also in R_σ

3. In the third step, for every pair of states (s, r) of the transition system, we take the union of the sets associated to the transitions from s to r. That is, $(i, J) \in R_{(s,r)}$ implies that there exists a regular expression σ such that

there is a transition from s to r labelled with σ, i.e, $s \xrightarrow{\sigma}_{\boxplus} r$, and the language of σ is included in the language accepted by $B_{(i,J)}$.

4. Then, for every pair of states, we compute the closure of the sets. The computation is done until the fixpoint is reached. If $(h, J) \in R_{(s,t)}$ then there exists a sequence of transitions from s to t, i.e., $s \xrightarrow{\sigma_0}_{\boxplus} ... \xrightarrow{\sigma_n}_{\boxplus} t$ such that the language of the concatenation of $\sigma_0, ..., \sigma_n$ is included in the language accepted by $B_{(h,J)}$.

   We can prove that this invariant is satisfied at any step of the computation by applying induction on the number of *while-do* loops. After step 3, it is trivially satisfied considering only sequences of length one. Then, let us assume that the invariant is true after the mth *while-do* loop. If we perform the next step of the fixpoint computation, for any three states s, r and t such that:

   - If there is a pair (h, I) in $R_{(s,r)}$ and for all i in I, there is a pair (i, J) in $R_{(r,t)}$, (h, J) will be added to $R_{(s,t)}$.
   - By I.H., we know that there is a sequence $\sigma_0, ..., \sigma_n$ such that $s \xrightarrow{\sigma_0}_{\boxplus} ... \xrightarrow{\sigma_n}_{\boxplus} r$, and the language $[\![\sigma_0 \cdots \sigma_n]\!]$ is included in the language accepted by $B_{(h,I)}$ and
   - for all $i \in I$, there is a sequence $\sigma'_0, ..., \sigma'_n$ such that $r \xrightarrow{\sigma'_0}_{\boxplus} ... \xrightarrow{\sigma'_n}_{\boxplus} t$, and the language $[\![\sigma'_0 \cdots \sigma'_n]\!]$ is included in the language accepted by $B_{(i,J)}$. Then
   - for all $i \in I$ there will be a sequence $s \xrightarrow{\sigma_0}_{\boxplus} ... \xrightarrow{\sigma_n}_{\boxplus} \xrightarrow{\sigma'_0}_{\boxplus} ... \xrightarrow{\sigma'_n}_{\boxplus} t$ such that the language $[\![\sigma_0 \cdots \sigma_n]\!] \circ [\![\sigma'_0 \cdots \sigma'_n]\!]$ is included in the language accepted by $B_{(h,J)}$, which implies
   - there exists a sequence $s \xrightarrow{\sigma_0}_{\boxplus} ... \xrightarrow{\sigma_n}_{\boxplus} \xrightarrow{\sigma'_0}_{\boxplus} ... \xrightarrow{\sigma'_n}_{\boxplus} t$ such that the language $[\![\sigma_0 \cdots \sigma_n]\!] \circ [\![\sigma'_0 \cdots \sigma'_n]\!]$ is included in the language accepted by $B_{(h,J)}$, hence $[\![\sigma_0 \cdots \sigma_n \cdot \sigma'_0 \cdots \sigma'_n]\!] \subseteq B_{(h,J)}$

5. Finally, the algorithm returns the states s that are related with a target state $r \in R$, such that the relation $R_{(s,r)}$ contains a pair (b_0, J) where b_0 is the initial state of B and J only contains final states of B.

   From $J \subseteq F$ follows that the language accepted by $B_{(b_0,J)}$ is included in the language accepted by B. And, by the invariant of the previous loop (step 4), we see that there exists a sequence of regular expressions $\sigma_0, ..., \sigma_n$ such that there is a sequence of transitions from s to r labelled with $\sigma_0, ..., \sigma_n$, i.e., $s \xrightarrow{\sigma_0}_{\boxplus} ... \xrightarrow{\sigma_n}_{\boxplus} r$ and the language of the concatenation of the sigmas is included in the language accepted by $B_{(b_0,J)}$, so also in the language accepted by B.

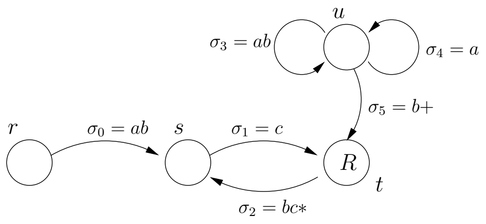
We can easily see that the function *exists_must* terminates, this is due to the fact that for every pair (s, r) the relation $R_{(s,r)}$ will contain elements in

$(B, \mathcal{P}(B))$, which are finite because the number of states B of the finite automaton generated from the input β is finite. Furthermore, the fixpoint computation is monotonic which implies that the algorithm will finish. We have sketched how to prove that the results of the algorithm are sound. It is still to be proved whether it finds all the states that satisfy the property or not.

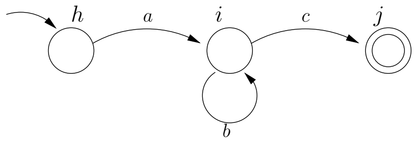
The use of regular expressions adds more computational complexity to the *normal* model checking algorithm. The algorithm is exponential on the size of the automaton corresponding to β and the automata of the transitions, and on the size of the transition systems. Even though the complexity is very high, in practise the regular expressions that will appear will be rather simple, so we expect that it will not cause a significant slow down of the normal model checking algorithms.

Let us now show two examples about *exists_must*.

Example 5.1: Considering the following transition system:



We want to compute *exists_must*(β, R) of it, with β is equal to $ab * c$ and $R = \{t\}$. The first step is to transform β to a deterministic automaton:



Then:

- $R_{\sigma_0} = R_{\sigma_3} = R_{\sigma_4} = \{(h, \{i\}), (h, \{i, j\}), (h, \{i, h\}), (h, \{i, j, h\})\}$
- $R_{\sigma_1} = \{(i, \{j\}), (i, \{j, i\}), (i, \{j, h\}), (i, \{j, i, h\})\}$
- $R_{\sigma_2} = \emptyset$
- $R_{\sigma_5} = \{(i, \{i\}), (i, \{i, j\}), (i, \{i, h\}), (i, \{i, j, h\})\}$.

Now, in step 3 we compute the relations between states of the transition system:

- $R_{(r,s)} = R_{\sigma_0}$

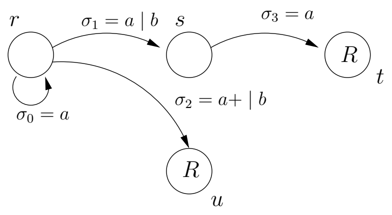
- $R_{(s,t)} = R_{\sigma_1}$
- $R_{(u,u)} = R_{\sigma_3} \cup R_{\sigma_4}$
- $R_{(u,t)} = R_{\sigma_5}$
- The sets for the rest of the pairs of states are empty.

We close the relations under concatenation, which adds:

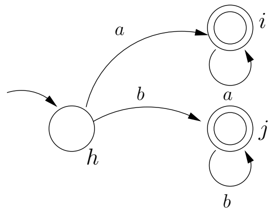
- $R_{(r,t)} = \{(h, \{j\}), (h, \{j, i\}), (h, \{j, h\}), (h, \{j, i, h\})\}$

Finally, we see that $R_{(r,t)}$ contains the pair $(h, \{j\})$ which is the initial state of β and $\{j\} = F$. Therefore, there is a path from r to t such the language of the concatenation of the labels is included in β. Hence, the result of the function will be $\{r\}$.

Example 5.2: The next example is included to motivate why we use pairs (i, J) with J being a set of states instead of pairs (i, j). The latter will not work for cases as the following. Let us consider the following transition system:



We want to compute *exists_must*(β, R) of it, with β is equal to $a+ \mid b+$ and $R = \{u, t\}$. The first step is to transform β to a deterministic automaton:



Then, $R_{\sigma_0} = R_{\sigma_3} = \{(h, \{i\}), ..., (i, \{i\}, ...)\}$[3], $R_{\sigma_1} = \{(i, \{i, j\}), ...\}$, $R_{\sigma_2} = \{(h, \{i, j\}), ...\}$. Now, in step 3 we compute the relations between states of the transition system: $R_{(r,r)} = R_{\sigma_0}$, $R_{(r,s)} = R_{\sigma_1}$, $R_{(s,t)} = R_{\sigma_3}$ and $R_{(r,u)} = R_{\sigma_2}$. The sets for the rest of the pairs of states are empty. We close the relations under concatenation, then we obtain:

[3]Note that we do not close the set upwards.

- $R_{(r,s)} = \{(h, \{i,j\}), ...\}$
- $R_{(r,t)} = \emptyset$
- $R_{(s,t)} = \{(h, \{i\}), ..., (i, \{i\}), ...\}$
- $R_{(r,u)} = \{(h, \{i,j\}), ...\}$

Then, the final result will be $\{r, s\}$.

5.6 Adding Accelerated Transitions.

So far, we have described how to capture semantically a transition that represents a set of computations. A different problem is, given a specification of the system, how to add *sound accelerated*-transitions. We give here some ideas about this.

In some cases a transition of the type $s \xrightarrow{a*}_{\boxplus} r$ corresponds to a loop in the original specification. The transition expresses that the loop executes a number of actions a and then terminates. We do not know how many cycles it contains, but it ends at some point. Therefore to add such a transition to the abstract labelled transition system, we will have to prove termination of the concrete loop. Proving termination of sequential programs has been investigated for many years, we believe that many of the results of this field can be applied to our framework.

One of the common ways to prove termination is by checking that the computation progresses in a given well-founded domain. For example, in order to infer fairness constrains, Pnueli ([79]) uses a monitor process composed in parallel with the modelled system. The monitor controls the progress in the domain of the naturals. In some cases it is trivial to find the domain, for example for the decreasing counter, but this is not always the case. There are other techniques and tools for proving termination, see for instance [119, 54].

A transition like $s \xrightarrow{(b \cdot a* \cdot c)+}_{\boxplus} r$ can capture a nested loop, whose termination can be proved separately for each loop. Another common source of missing *must*-transitions in abstractions appears in *if-then-else* constructions, when the condition is abstracted and it cannot be computed whether it is true or false. In that case, the abstraction will contain two *may*-transitions, one to the *then* branch ($s \xrightarrow{a}_{\diamond} r$) and the other to the *else* one ($s \xrightarrow{b}_{\diamond} r$). There we are sure that either one or the other is taken, therefore we can add the transition $s \xrightarrow{a|b}_{\boxplus} r$.

Let us now do a simple analysis of the bounded retransmission protocol, to describe a possible methodology to add and use *accelerations*.

5.6.1 ***The Bounded Retransmission Protocol*** Revisited

We refer to the *Bounded Retransmission Protocol* explained in section 2.5. We have seen that the property **C4** was *possibly* satisfied but not *necessarily*, therefore we could not infer its satisfaction or refutation to the concrete. The property stated that there is a path, after a positive notification of the receiver, in which the sender notifies *I don't know*. The sender delivers dk when he is not sure that the last packet has been delivered or not, because this packet or its acknowledgements were lost and the number of retransmissions has expired.

(C4): $[\ \mathsf{T}^* \ .\ \text{'}\mathsf{R}(.*,\mathsf{I}_{\mathsf{ok}})\text{'}\ ]\ \langle\ \mathsf{T}^*\ .\ \text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ \rangle\ \mathsf{T}$

We recall that, to prove the property, we have to find all *may*-paths leading to $\text{'}\mathsf{R}(.*,\mathsf{I}_{\mathsf{ok}})\text{'}$, and then, to check that they are followed by a *must*-path leading to $\text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}$. The property should be true in the concrete, but the abstraction that we have selected removes the real value of the retransmissions, so the sender is not sure if the number has decreased until zero.

We are going to do an *ad hoc* analysis of the system in order to enrich the model to be able to prove that the property is *necessarily* satisfied. If we examine the counter-example given for the *necessary* evaluation of formula **C4**, we see that there is an abstract *may*-loop from which a *may* dk transition can be reached. This loop contains the following actions:

1. The sender transmits the last packet.
2. The sender sets the timer and decreases the counter.
3. The channel delivers the packet.
4. The receiver notifies ok.
5. The receiver sends the acknowledgement.
6. The channel loses the acknowledgement.
7. The timer interrupts because the timeout expires.

We see the counter that controls the number of retransmissions is only changed in action 2 where it is decreased. In the concrete specification the counter is decreased by one and, trivially, at some point it will reach zero, but in the abstract we have lost the value of the counter. We know that the counter is bigger than zero but we do not know the exact value. Therefore, we can add an *accelerated* transition capturing the computation of the loop $+ \xrightarrow{\sigma}_{\boxplus} 0$, where:

- $+$ is the state in which we start to transmit a packet. The counter is bigger than zero.
- 0 is the state is which the counter is zero, therefore from there the sender can notify *I don't know*.
- The label σ is equal to $(1{\cdot}2{\cdot}3{\cdot}4{\cdot}5{\cdot}6{\cdot}7)+$

By adding this new transition we can infer the satisfaction of the property which could not be inferred otherwise. We see that the language generated by the new transition σ concatenated with $\text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}$ is included in $[\![\mathsf{T}* \cdot \text{'}\mathsf{S}(\mathsf{I}_{\mathsf{dk}})\text{'}\ ]\!]$.

5.7 Conclusion

In this chapter we have introduced an extension of the abstraction of action labels which allows to encode more expressive abstractions. The framework we have defined can be improved by using more complicated relations between concrete and abstract domains, such as Galois Connections, instead of just homomorphisms.

Even though the model checking algorithm we have defined is in general very inefficient the regular expressions used to define *accelerations* will usually be very simple, which should control the extra computational costs.

In the previous section, we have done a *manual* analysis of the communication protocol from which we can extract a general methodology for adding *accelerations*. Let us survey the steps:

1. First, we try to prove a property using the simple abstraction framework. If the property cannot be inferred or refuted, then
2. we analyse the counter-example looking for the abstract behaviours that do not correspond to any concrete behaviour.
3. If a counter-example is a spurious *may* loop, we try to prove termination of it. Here, we can reuse many of the existing techniques and algorithms for termination.
4. We add the corresponding *accelerated* transition.
5. We try to prove the property again.

The methodology is based on classical counter-example guided refinement (see, for example [82]) with the inclusion of the termination proofs and the addition of *accelerations*. Work has to be done to implement the ideas into the toolset described in Chapter 3.

The use of *accelerations* can be combined with fairness constraints à la Pnueli. In our framework, on the one hand *accelerations* add more *necessary* behaviours which helps to prove more liveness properties. On the other hand fairness constraints remove *possible* behaviours which help to prove more safety properties.

Related work is so-called *regular model checking* [19, 1]. In that theory systems are fully specified using regular expressions, then the verification is done by comparing the language generated by the system with some property. Regular model checking has been successfully used to verify simple examples, in which the behaviours present a lot of regularity [44]. In our framework, also some parts of the system are describe by regular expressions, which allows more flexibility. Furthermore, our technique is integrated inside the abstract interpretation framework.

5.8 Proofs

Proof Lemma(5.4.1): We only include the proof for the modal operators.

Case: *Necessary Universal*

1. We have to prove $[\![[\beta]\varphi]\!]^{nec} = [\![\neg\langle\beta\rangle\neg\varphi]\!]^{nec}$
2. $[\![[\beta]\varphi]\!]^{nec}$ equals:
$$\{s \mid \forall r, a_0, ..., a_i.\, s \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} r \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \implies r \in [\![\varphi]\!]^{nec}\}$$
3. $[\![\neg\langle\beta\rangle\neg\varphi]\!]^{nec} = \neg[\![\langle\beta\rangle\neg\varphi]\!]^{pos}$, which equals:
$$\{s \mid \nexists r, a_0, ..., a_i.\, s \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} r \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \wedge r \in [\![\neg\varphi]\!]^{pos}\}$$
4. which is equivalent to:
$$\{s \mid \forall r, a_0, ..., a_i.\, s \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} r \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \implies r \notin [\![\neg\varphi]\!]^{pos}\}$$
5. By the semantics of negation $r \notin [\![\neg\varphi]\!]^{pos}$ is equal to $r \in [\![\varphi]\!]^{nec}$ which proves the case.

□ (Case)

Case: *Possible Universal*

1. We have to prove $[\![[\beta]\varphi]\!]^{pos} = [\![\neg\langle\beta\rangle\neg\varphi]\!]^{pos}$
2. $[\![[\beta]\varphi]\!]^{pos}$ equals:
$$\{s \mid \forall r, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \implies r \in [\![\varphi]\!]^{pos}\}$$
3. $[\![\neg\langle\beta\rangle\neg\varphi]\!]^{pos} = \neg[\![\langle\beta\rangle\neg\varphi]\!]^{nec}$, which equals:
$$\{s \mid \nexists r, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \wedge r \in [\![\neg\varphi]\!]^{nec}\}$$
4. which is equivalent to:
$$\{s \mid \forall r, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \implies r \notin [\![\neg\varphi]\!]^{nec}\}$$
5. By the semantics of the negation $r \notin [\![\neg\varphi]\!]^{nec}$ is equal to $r \in [\![\varphi]\!]^{pos}$ which proves the case.

□ (Case)

□ (Lemma)

Proof: (Lemma 5.4.2) We apply induction over the structure of φ. The cases $\mathsf{T}, \mathsf{F}, \neg, \wedge$ and $\vee$ are trivial. We consider $[\![\langle\beta\rangle\varphi]\!]^{nec} \subseteq [\![\langle\beta\rangle\varphi]\!]^{pos}$, the other case is similar:

1. By Induction Hypothesis we have: $[\![\varphi]\!]^{nec} \subseteq [\![\varphi]\!]^{pos}$
2. $s \in [\![\langle\beta\rangle\varphi]\!]^{nec}$ implies:
$$\exists\, r, \sigma_0, ..., \sigma_i.\, s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \;\wedge\; [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \;\wedge\; r \in [\![\varphi]\!]^{nec}$$
3. By definition of *accMLTS*, $s \xrightarrow{\sigma}_{\boxplus} t$ implies $s \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} t$ and $[a_0 \cdots a_j] \in [\![\sigma]\!]$, which implies:
$$s \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} r \implies s \xrightarrow{a_{0,0}}_{\diamond} \cdots \xrightarrow{a_{0,j}}_{\diamond} \cdots \xrightarrow{a_{i,0}}_{\diamond} \cdots \xrightarrow{a_{i,j}}_{\diamond} r \,\wedge$$
$$[a_{0,0} \cdots a_{0,j} \cdots a_{i,0} \cdots a_{i,j}] \in [\![\sigma_0 \cdots \sigma_i]\!]$$
4. Therefore, assuming $s \in [\![\langle\beta\rangle\varphi]\!]^{nec}$ implies:
$$\exists\, r, a_{0,0} \ldots a_{0,j} \ldots a_{i,0} \ldots a_{i,j}.\, s \xrightarrow{a_{0,0}}_{\diamond} \cdots \xrightarrow{a_{0,j}}_{\diamond} \cdots \xrightarrow{a_{i,0}}_{\diamond} \cdots \xrightarrow{a_{i,j}}_{\diamond} r \,\wedge$$
$$[a_{0,0} \cdots a_{0,j} \cdots a_{i,0} \cdots a_{i,j}] \in [\![\beta]\!] \text{ and by I.H. } r \in [\![\varphi]\!]^{pos}$$

   hence, $s \in [\![\langle\beta\rangle\varphi]\!]^{pos}$, which proves the Lemma.

$\square$(*Lemma*)

Proof: (Lemmas 5.4.3 and 5.4.4) These proofs can be found at the end of Chapter 1

$\square$(*Lemmas*)

Let us now proceed with the proofs of the Theorems.

Proof (Theorem 5.4.5): We apply induction over the structure of φ, on the *necessary* and *possible* semantics at the same time. We skip the trivial cases.

Case: *Necessary Existential*

1. We have to prove $\widehat{N}, \widehat{s} \models^{nec} \langle\beta\rangle\varphi$ implies $\widehat{M}, \widehat{s} \models^{nec} \langle\beta\rangle\varphi$
2. By Induction Hypothesis, we know $\widehat{N}, \widehat{s} \models^{nec} \varphi \implies \widehat{M}, \widehat{s} \models^{nec} \varphi$
3. $\widehat{N}, \widehat{s} \models^{nec} \langle\beta\rangle\varphi$ implies:
$$\exists\, \widehat{r}, \sigma'_0, ..., \sigma'_i.\, \widehat{s} \xrightarrow{\sigma'_0}_{\boxplus} \cdots \xrightarrow{\sigma'_i}_{\boxplus} \widehat{r} \;\wedge\; [\![\sigma'_0 \cdots \sigma'_i]\!] \subseteq [\![\beta]\!] \;\wedge\; \widehat{N}, \widehat{r} \models^{nec} \varphi$$

4. By definition of approximation, $\widehat{s} \overset{\sigma'}{\rightarrowtriangle}_{\boxplus} \widehat{r} \implies \exists\, \sigma_0, ..., \sigma_j .\, \widehat{s} \overset{\sigma_0}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_i}{\longrightarrow}_{\boxplus} \widehat{r} \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\sigma']\!]$, therefore:

$$\begin{aligned}&\widehat{s} \overset{\sigma'_0}{\rightarrowtriangle}_{\boxplus} \cdots \overset{\sigma'_i}{\rightarrowtriangle}_{\boxplus} \widehat{r} \implies \exists\, \sigma_{0,0}, ..., \sigma_{0,j}, \ldots, \sigma_{i,0}, ..., \sigma_{i,j} .\\ &\widehat{s} \overset{\sigma_{0,0}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{0,j}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{i,0}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{i,j}}{\longrightarrow}_{\boxplus} \widehat{r} \wedge \\ &[\![\sigma_{0,0} \cdots \sigma_{0,j} \cdots \sigma_{i,0} \cdots \sigma_{i,j} .]\!] \subseteq [\![\sigma'_0 \cdots \sigma'_i]\!]\end{aligned}$$

5. Assuming $\widehat{N}, \widehat{s} \models^{nec} \langle\beta\rangle\varphi$, follows:

$$\begin{aligned}&\exists\, \widehat{r}, \sigma_{0,0}, ..., \sigma_{0,j}, \ldots, \sigma_{i,0}, ..., \sigma_{i,j} . \widehat{s} \overset{\sigma_{0,0}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{0,j}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{i,0}}{\longrightarrow}_{\boxplus} \cdots \overset{\sigma_{i,j}}{\longrightarrow}_{\boxplus} \widehat{r} \wedge \\ &[\![\sigma_{0,0} \cdots \sigma_{0,j} \cdots \sigma_{i,0} \cdots \sigma_{i,j}]\!] \subseteq [\![\beta]\!] \text{ and by I.H. } \widehat{M}, \widehat{r} \models^{nec} \varphi\end{aligned}$$

   which proves the case.

$\square$ (Case)

Case: *Possible Existential*

1. We have to prove $\widehat{N}, \widehat{s} \not\models^{pos} \langle\beta\rangle\varphi$ implies $\widehat{M}, \widehat{s} \not\models^{pos} \langle\beta\rangle\varphi$ or what is the same $\widehat{M}, \widehat{s} \models^{pos} \langle\beta\rangle\varphi$ implies $\widehat{s} \models^{pos} \langle\beta\rangle\varphi$
2. By Induction Hypothesis, we know $\widehat{M}, \widehat{s} \models^{pos} \varphi \implies \widehat{N}, \widehat{s} \models^{pos} \varphi$
3. $\widehat{M}, \widehat{s} \models^{pos} \langle\beta\rangle\varphi$ implies:

$$\exists\, \widehat{r}, a_0, ..., a_i .. \widehat{s} \overset{a_0}{\longrightarrow}_{\diamond} \cdots \overset{a_i}{\longrightarrow}_{\diamond} \widehat{r} \wedge\ [a_0 \cdots a_i] \in [\![\beta]\!] \wedge \widehat{M}, \widehat{r} \models^{pos} \varphi$$

4. By definition of approximation, $\widehat{s} \overset{\sigma}{\longrightarrow}_{\diamond} \widehat{r} \implies \widehat{s} \overset{a}{\rightarrowtriangle}_{\diamond} \widehat{r}$, therefore:

$$\widehat{s} \overset{a_0}{\longrightarrow}_{\diamond} \cdots \overset{a_i}{\longrightarrow}_{\diamond} \widehat{r} \implies \widehat{s} \overset{a_0}{\rightarrowtriangle}_{\diamond} \cdots \overset{a_i}{\rightarrowtriangle}_{\diamond} \widehat{r}$$

5. Assuming $\widehat{M}, \widehat{s} \models^{pos} \langle\beta\rangle\varphi$, follows:

$$\exists\, \widehat{r}, a_0, ..., a_i . \widehat{s} \overset{a_0}{\rightarrowtriangle}_{\diamond} \cdots \overset{a_i}{\rightarrowtriangle}_{\diamond} \widehat{r} \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \text{ and by I.H. } \widehat{N}, \widehat{r} \models^{pos} \varphi$$

   which proves the case.

$\square$ (Case)

$\square$ (Theorem)

Proof (Theorem 5.4.6): As in the previous Theorem, we apply induction over the structure of φ[4]. We skip the trivial cases.

Case: *Necessary Existential*

[4]In this case, we do not explicitly represent the model to which formulas refer.

1. We have to prove, $\widehat{s} \models^{nec} \langle\beta\rangle\varphi$ implies $s \models \langle\beta\rangle\varphi$
2. By Induction Hypothesis, we know $\widehat{s} \models^{nec} \varphi \wedge h(s) = \widehat{s} \implies s \models \varphi$
3. $\widehat{s} \models^{nec} \langle\beta\rangle\varphi$ implies:
$$\exists\, \widehat{r}, \sigma_0, ..., \sigma_i.\, \widehat{s} \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} \widehat{r} \wedge [\![\sigma_0 \cdots \sigma_i]\!] \subseteq [\![\beta]\!] \wedge \widehat{r} \models^{nec} \varphi$$
4. By definition of minimal abstraction, $\widehat{s} \xrightarrow{\sigma}_{\boxplus} \widehat{r}$ implies $\forall\, s.h(s) \in \widehat{s} \implies (\exists\, r, a_0, ..., a_j.\, h(r) = \widehat{r} \wedge s \xrightarrow{a_0} ... \xrightarrow{a_j} r \wedge [a_0 \cdots a_j] \in [\![\sigma]\!])$, therefore:
$\widehat{s} \xrightarrow{\sigma_0}_{\boxplus} \cdots \xrightarrow{\sigma_i}_{\boxplus} \widehat{r} \implies$
$(\forall\, s.h(s) \in \widehat{s} \implies \exists\, r, a_{0,0}, ..., a_{0,j}, ..., a_{i,0}, ..., a_{0,j}.\, h(r) = \widehat{r} \wedge$
$s \xrightarrow{a_{0,0}} \cdots \xrightarrow{a_{0,j}} \cdots \xrightarrow{a_{i,0}} \cdots \xrightarrow{a_{i,j}} r \wedge [a_{0,0} \cdots a_{0,j} \cdots a_{i,0} \cdots a_{i,j}] \in [\![\sigma_0 \cdots \sigma_i]\!])$
5. Assuming $\widehat{s} \models^{nec} \langle\beta\rangle\varphi$, follows:
$$\exists\, r, a_{0,0}, ..., a_{0,j}, ..., a_{i,0}, ..., a_{i,j}.\, s \xrightarrow{a_{0,0}} \cdots \xrightarrow{a_{0,j}} \cdots \xrightarrow{a_{i,j}} \cdots \xrightarrow{a_{i,j}} r \wedge$$
$$[a_{0,0} \cdots a_{0,j} \cdots a_{i,0} \cdots a_{i,j}] \in [\![\beta]\!] \text{ and by I.H. } r \models \varphi$$
which proves the case.

□ (Case)

Case: *Possible Existential*

1. We have to prove $\widehat{s} \models^{pos} \langle\beta\rangle\varphi$ implies $s \not\models \langle\beta\rangle\varphi$ or what is the same $s \models \langle\beta\rangle\varphi$ implies $\widehat{s} \models^{pos} \langle\beta\rangle\varphi$
2. By Induction Hypothesis, we know $s \models \varphi \wedge h(s) = \widehat{s} \implies \widehat{s} \models^{pos} \varphi$
3. $s \models \langle\beta\rangle\varphi$ implies:
$$\exists\, r, a_0, ..., a_i.\, s \xrightarrow{a_0} \cdots \xrightarrow{a_i} r \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \wedge r \models \varphi$$
4. By definition of minimal abstraction, $h(s) \xrightarrow{a} r \implies \widehat{s} \xrightarrow{a} h(r)$, therefore:
$$s \xrightarrow{a_0} \cdots \xrightarrow{a_i} r \implies \widehat{s} \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} h(r)$$
5. Assuming $s \models \langle\beta\rangle\varphi$, follows:
$$\exists\, \widehat{r}, a_0, ..., a_i.\, \widehat{s} \xrightarrow{a_0}_{\diamond} \cdots \xrightarrow{a_i}_{\diamond} \widehat{r} \wedge [a_0 \cdots a_i] \in [\![\beta]\!] \text{ and by I.H. } \widehat{r} \models^{pos} \varphi$$
which proves the case.

□ (Case)

□ (Theorem)

Part II

On Coordination...

Chapter 6

Formal Model of JavaSpaces

This chapter introduces a formal model of JavaSpaces. JavaSpaces is a shared data space architecture that provides a virtual repository allowing entities, like clients and servers, to communicate by sharing objects. The formal model, written in μCRL, captures the main features of the architecture, such as, communication primitives, transactions, leasing and the notification mechanism. The main purpose of the proposed formalism is to allow the verification of distributed applications built on top of JavaSpaces. The next chapter is dedicated to illustrating the use of the model by verifying different applications.

6.1 Introduction

It is well known that the design of reliable distributed systems can be an extremely arduous task. The parallel composition of processes with a simple behaviour can produce a wildly complicated system. A distributed application has to face some important challenges: it has to facilitate communication and synchronisation between processes across heterogeneous networks, dealing with latencies, partial failures and system incompatibilities. The use of coordination architectures is a suitable way to manage the complexity of specifying and programming large distributed applications.

Re-usability is one of the most important issues of coordination architectures. Once the architecture has been implemented on a distributed network, different applications can be built according to the requirements without any extra adaptation. Programmers implement their systems using the interface provided by the architecture, which consists of a set of primitives or operators.

In this chapter, we study the JavaSpaces^{TM} [91] technology that is a Sun Microsystems, Inc. architecture based on the Linda coordination language [25]. JavaSpaces is a Jini^{TM} [92] service that provides a platform for designing distributed computing systems. It gives support to the communication and synchronisation of external processes by setting up a common shared space. JavaSpaces is both an application program interface (API) and a distributed programming model. The coordination of applications built under this technology is modeled as a flow of objects. The communication is different from traditional paradigms based on message passing or method invocation models. Components of applications built under the JavaSpaces model are "loosely coupled", they do not communicate with each other directly but by sharing information via the common repository.

Several remote processes can interact simultaneously with the shared repository, the space handles the details of concurrent access. The interface provided by JavaSpaces is essentially composed by insertion and lookup primitives. Furthermore, it give support for transactions, leasing and it implements a notification mechanism. The API provided by Sun Microsystems is:

```
public interface JavaSpace {
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    public final long NO_WAIT = 0; // don't wait at all
    Entry read(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry readIfExists(Entry tmpl, Transaction txn,
                       long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry take(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
              RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmpl, Transaction txn,
                       long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    EventRegistration notify(Entry tmpl, Transaction txn,
              RemoteEventListener listener, long lease,
              MarshalledObject handback)
        throws RemoteException, TransactionException;
    Entry snapshot(Entry e) throws RemoteException;
}
```

A *write* operation places a copy of an entry into the space. Entries are granted for a fixed period of time (*lease*). The space automatically removes the objects when their lease expires. Two different entries have the same type if and only if they are of the same class. They should implement the following interface:

```
public interface Entry extends java.io.Serializable { }
```

Entries can be located by "associative lookup" implemented by matching *templates*. Processes find the entries they are interested in by expressing constraints about their contents without having any information about the object identification, owner or location. Match operations use entry objects of a given type, whose fields can either have values (references to objects) or wildcards (null references). When considering a template T as a potential match against an entry E, fields with values in T must be matched exactly by the value in the same field of E. Wildcards in T match any value in the same field of E.

A *read* request returns a copy of an object from the space that matches the provided *template*. Read requests are blocking in principle, but processes can limit the amount of time they are willing to wait for a matching entry. If this time expires without finding a matching entry, a *null* entry is returned. *ReadIfExists* is similar to *read*, but it only blocks if there exist matching objects in the space that are involved in some transaction (see below). *Take* and *takeIfExists* are the *destructive* versions of *read*: once an object has been returned, it is removed from the space.

Transactions ensure that a set of grouped operations are performed on the space atomically, in such a way that either all of them complete or none are executed. After the creation of a transaction, a process can either *abort* it, or *commit*. Transactions are also subject to leasing. If the lease expires the space automatically aborts the transactions and raises an *exception*. Transactions affect the behaviour of the primitives. E.g., an object written within a transaction is not externally accessible until the transaction commits, if the transaction aborts the insertion will never be visible. Transactions provide a means for enforcing consistency. JavaSpaces' transactions are claimed to preserve the ACID properties: Atomicity, Consistency, Isolation and Durability. The transactional model, the leasing mechanism and the event notification mechanism are supported by the JINI architecture. We see below an extract of the Jini specification of transactions:

```
public interface Transaction {
    public static class Created implements Serializable {
        public final Transaction transaction;
        public final Lease lease;
        Created(Transaction transaction, Lease lease) {...}
    }
    void commit()
     throws UnknownTransactionException, ...;
    void abort()
        throws UnknownTransactionException, ...;
    ...
}
```

Transactions are create by means of an external entity called Transaction Manager, we see part of its interface:

```
public interface TransactionManager
    extends Remote, TransactionConstants
    {
    public static class Created implements Serializable {
        public final long id;
        public final Lease lease;
        public Created(long id, Lease lease) {...}
    }
    Created create(long leaseFor)
        throws LeaseDeniedException, ...;
    int getState(long id)
        throws UnknownTransactionException, ...;
    ...
}
```

The transaction manager is in charge of creating new transactions, and performing the commits, aborts, et cetera... Apart from transactions, the space also handles some distributed events, in particular: a process can inform the space about its interest in future incoming entries by using the *notify* primitive. The space will notify by an event when a matching object arrives into the space. This feature will be explained in more detail in the next section.

Figure 6.1 presents an overview of the JavaSpaces architecture. Below, we present a small example of a function that renames entries that match a given template to a different type. The renaming is encapsulated in a transaction.

```
renamer(JavaSpace space,
      TransactionManager trcManager, MyEntry typeA){
   while(true){
```

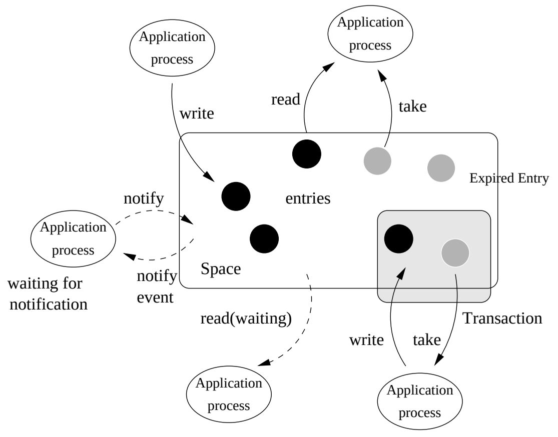


Figure 6.1: JavaSpaces architecture overview

```
      try{
         Transaction trc =
            (Transaction) trcManager.create(TIMEOUT);
         MyEntry e =
            (myEntry) space.take(typeA, trc, long.MAX_INT);
         space.write(e.rename(), trc, Lease.FOREVER);
         trc.commit();
      } catch (Exception ex){
        break; // exit the loop
      }
   }
}
```

The function first creates a transaction by requesting it to a transaction manager. Then, the process retrieves one entry of *typeA* by doing a blocking *take*. The entry is selected by using a *template* that is provided in the arguments of the function. A *template* is an entry in which some fields have values an the rest are null. The matching is performed by comparing the entries of the space with the non-null values of the *template*. After the *take*, the function puts the renamed version into the space and commits the transaction. If the timeout of the transaction expires the space sends an exception which is caught by the process, then the process finishes.

A full description of the JavaSpaces and Jini architectures can be found at:

- `http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html`
- `http://www.jini.org/nonav/standards/davis/doc/specs/html/coreTOC.html`

6.1.1 Modelling JavaSpaces

This chapter is dedicated to explaining the formal specification of the JavaSpaces architecture. One of the main difficulties of the specification resides in the lack of orthogonality of the different features of the architecture. Some of the features interfere with others. For instance, the results of the *write* operations depends on whether they are executed inside a transaction or not. This fact inflates the number of possible combinations of behaviours. We need to take into account all the possibilities, which increases the difficulty of the modelling task. The model has been built taking into account the following considerations:

1. The model should support the most important parts of the JavaSpaces specification introduced above. Even though it has to be as compliant as possible with the informal specification provided by Sun [91], some concepts are left out to keep the model simple and suitable to be used for automatic verification. For example, we have left away the snapshot operation which is basically mend to improve performance of reading the same entry several times.

2. The main goal of the model is to be used for verification by model checking. In general, to apply model checking, we need to generate the full state space of the system and in order to generate it we need an executable and finite specification.

3. Therefore, we need to select the correct level of abstraction in order to have enough details of the model to allow the *execution* of it. However, we do not specify the internal behaviours of the algorithms used by JavaSpaces, we only need to represent the external actions that they perform. For example, we have not modelled in fully detail the transaction mechanism, with the transaction manager, the protocols, et cetera... Instead, we have modelled the external behaviour of the transactions.

4. An executable specification allows to use simulation to get the first insight into the analysed systems and to quickly check the behaviour of them, as a step prior to the formal verification.

5. If we want finite specifications, we are forced to constrain the instances of the systems by limiting the resources that the space and applications can access. We are going to define a set of constants that bound the different sources of infiniteness of the systems.

6. The behaviour of JavaSpaces applications strongly depend on the data structures managed by the space. μCRL is a suitable language to model this system because it allows an elegant integration of data and processes.

According with these considerations and the language selected to do the specification, we enumerate below some of the important modelling choices:

- The different classes of the JavaSpaces architecture are specified using μCRL sorts. In section A.1 we give a full list of all the sorts.
- The use of sorts does not allow to implement the same matching procedure as it is done in the JavaSpaces specification. We have modelled it by using queries instead of by matching templates, see section 6.2.
- The main entities of the space (Entries, Transactions, Leases,...) are identified by using natural numbers. These numbers are used as pointers to the objects.
- In the JavaSpaces specification, resources are limited, for instance, there is maximum number of entries that can be stored in the space. These limits depend on the available memory of the system. To model this feature, we use costants to bound the identifiers of the resources. We modelled also a mechanism to assign fresh identifiers to every created resource, when possible.
- JavaSpaces primitives may raise exceptions due to, for example, network failures or external interruptions. For simplicity, we have modelled the exception mechanism only in case of expiration of transactions. All the primitives (but look-up operations) are considered atomic.
- Look-up operations (*take*, *read, takeIfExists* and *readIfExists* are modelled by two actions. One used by the external process to do the query and another used by the space to return the required values, see section 6.3.1.
- μCRL does not have a mechanism to handle exceptions. To model exceptions, in case of transaction expiration, we have used an extra action that should be executable in all the scope of the transaction, see section 6.2.1.
- The time is modelled using a centralised clock operated by the space, see section 6.3.4. There is no synchronisation between the clocks of external applications and the central time.
- We have considered that the space always grants the requested leases and timeouts. In the JavaSpaces specification, the space in some cases is allowed to grant smaller times.
- JavaSpaces implementations are multi-threaded hence many different operations can be done in parallel. Instead of having different processes, we modelled the space as a single process that can handle multiple requests by using interleaving.

The modelling choices are described with more detail in the next sections. After this short introduction about the JavaSpaces architecture, and the main goals and considerations about the formal model, the rest of the chapter is organised as follows: first, we introduce the specification from the external application point of view, describing the JavaSpaces interface. Then, we present the model of the implementation details from the space point of view. The appendix includes the full μCRL specification and a guide with the description of all the data structure included in the specification.

6.2 Application Point of View

The space is modeled as a single process called *javaspace*. User applications are implemented as external processes executed in parallel with the space. External applications exchange data between them by transferring entries through the shared space. The communication between the *javaspace* process and the external applications is done by means of a set of synchronous actions, derived from the JavaSpaces API, presented in the previous section. A JavaSpaces system is specified in μCRL as follows[1]:

$$\mathsf{System} = \tau_I \partial_H(\mathsf{javaspace(...)} \parallel \mathsf{external\_p_0(...)} \parallel \mathsf{external\_p_1(...)} \parallel ...)$$

The arguments of the *javaspace* process represent the current state of the space. They are composed by: stored objects, pending look-up operations, active transactions, et cetera... These arguments are explained in detail in section 6.3. External processes are anonymous and they cannot communicate with each other but through the space. We recall that τ_I is use to hide a set of actions and ∂_H to force communication of some specific actions, we explain the specification in more detail in section 6.4.

In the JavaSpaces specification, an entry corresponds to a serialisable Java object which implements the public interface Entry. In our model, entries are represented by a *sort*. Users can define their own data structure according to the application requirements. Entries are composed by data fields from standard sorts (naturals, booleans,...) or new sorts, and operators. The sort *Entry* must include the equality (*eq*) function, and the constructor *entryNull*, both are necessary to perform the look-up operations. The following code presents the definition of a simple type of entries:

```
sort  Entry
func  entryNull:→Entry
      A,B: →Entry
map   eq: Entry×Entry→Bool
      rename:Entry→Entry
var   e: Entry
rew   eq(e, e) = T
      eq(entryNull, A) = F     eq(A, entryNull) = F
      eq(entryNull, B) = F     eq(B, entryNull) = F
      eq(A, B) = F             eq(B, A) = F
      rename(A) = B            rename(B) = A
```

[1]We refer to section 2.2 for an introduction to μCRL.

The insertion of an entry into the space is done by means of the *write* action. The μCRL definition is derived from the following fragment of the JavaSpaces API:

```
Lease write(Entry e, Transaction txn, long lease)
     throws RemoteException, TransactionException;
```

The arguments of the action *write* are: the entry to insert, and the transaction identifier to which the action is attached. The space returns the reference to the requested lease. The μCRL signature of the operation is:

sort Nat, Entry, Time
act write: `Entry`$\times$`Nat`$\times$`Time`$\times$`Nat`

The first three arguments are provided by the external application, the fourth is the return value. The inserted data have to be of the *sort Entry*, as defined above. The behaviour of the action depends on whether it is executed under a transaction or not. If it is not joined to any transaction, meaning that the *transaction id* parameter is equal to *NULL*, then the insertion is instantaneously updated in the space. The use of transactions is explained in more detail further in the present section. In our model, *Write* actions are executed atomically. We do not allow exceptions during operations (the same assumption is made for all the operations, the only exceptions allowed are the ones caused by expiration of transactions). Once a *write* has been executed the entry is successfully inserted. Different *write* invocations will place different objects in the repository, even if the data values are equal.

Users can associate a lease to every inserted entry. An entry is automatically removed from the space when its lease expires. Leases are of sort *Time* which basically consists of natural numbers together with a special constant *FOREVER*. The null value (0) means that the entry is deleted in the same time unit that it is placed in the space. The *FOREVER* value says the entry will never be removed. For simplicity, we assume that the requested leases are always granted (in the real JavaSpaces the granted lease may differ from the requested one). The space returns the reference to the lease, which is a natural number (the fourth argument of *write*). The lease can be renewed by using the following action, in which the process passes the lease identifier and the new requested timeout to the space:

act renew: `Nat`$\times$`Time`

We have implemented another version of the *write* action with three arguments (write: `Entry`$\times$`Nat`$\times$`Time`) in which the space does not return the lease's reference. This version is used when the external process does not have future interest in modifying the lease's timeout. We give an example of *write* invocation in which the application process inserts an entry A not attached to any transaction and leased, first, for one time unit and then upgraded to two time units:

```
proc  p =
      ...
      .$\sum_{\text{lease:Nat}}$ write(A, NULL, tt(1), lease)
        .renew(lease, tt(2))
      ...
```

Look-up primitives could be classified as: *destructive* and *non-destructive*, depending on whether the item is removed or not after the execution of the action, and in *blocking* and *non-blocking* depending on whether the process waits until it receives the requested item. We can invoke destructive look-ups (*take*) or non-destructive (*read*), setting up the time during which the action blocks.

The JavaSpaces specification says that a look-up request searches in the space for an Entry that matches the template provided in the action. If the match is found, a reference to a copy of the matching entry is returned. If no match is found, *null* is returned. We implement the matching operation by adding to every invocation one predicate, as argument, which determines whether an entry matches the action or not. This predicate belongs to the *sort Query*, defined using the sort Entry. The sort *Query* must include the operator *match* used to perform the matching.

Let's see an example of Query *sort* that has three possible queries: *any* that matches any entry in the space and *isTypeA* and *isTypeB* that match *A* and *B* entries respectively:

```
sort  Query
func  any: →Query
      isTypeA, isTypeB: →Query
map   match: Query×Entry→Bool
var   e: Entry   q: Query
rew   match(q, entryNull) = F    match(any, e) = ¬ eq(e, entryNull)
      match(isTypeA, A) = T      match(isTypeA, B) = F
      match(isTypeB, A) = F      match(isTypeB, B) = T
```

An entry of the space will match a look-up action if it satisfies the associated query, as indicated by the *match* predicate. We have implemented four look-up primitives: *read*, *take*, *readIfExists* and *takeIfExists*. All of them take the following arguments: channel identifier, transaction identifier number, timeout and query.

The execution of a look-up primitive is done by means of two atomic actions. First the external process invokes the primitive (*read*, *take*, . . .), then the space communicates the result of the requests by returning a matching entry if the operation was successfully performed or an *entryNull* if the timeout has expired and no objects satisfied the query. A well-formed specification of an external application has to have a *return* operation immediately after a look-up operation. This constraint avoids that a process executes two look-up operations without waiting for the result. The external processes provides an identifier pointing to the channel where it is expecting the answer. The returned entry will be send by the space to this channel. The signatures are[2]:

sort Nat, Entry, Time, Query
act read, take, readIfExists,takeIfExists: $\texttt{Nat}\times\texttt{Nat}\times\texttt{Time}\times\texttt{Query}$
ReadReturn, TakeReturn,
ReadIEReturn, TakeIEReturn: $\texttt{Nat}\times\texttt{Entry}$

The return operation uses the chanel identifier passed as argument in the look-up invocation, to send the returned entry. Let's see an example program using the *blocking take* which requests an entry of type A:

```
proc  p(id:Nat) =
      ...
      .take(id, NULL, tt(1), isTypeA)
      .$\sum_{e:Entry}$ TakeReturn(id, e)
         ....$\lhd$ eq(e, entryNull) $\rhd$...
      ...
```

The *ifExists* operations perform a test of presence before blocking, i.e., they check whether there are matching entries in the space which may be locked by some transactions. If there are no (locked or unlocked) entries the *ifExists* operations will not block, i.e., its timeout will be set to 0, and an *entryNull* will be sent as soon as the space executes the return operation.

6.2.1 Transactions

In our model the instantiation of a transaction is done by the action *create*, which has as arguments: the created transaction (provided by the space), and the requested lease. The space allocates a new resource and returns to the user the identification number of the created transaction and the reference to the granted lease, both are encapsulated in the *TransactionCreated* structure.

sort TransactionCreated
func trcCreated: $\texttt{Nat}\times\texttt{Nat}\rightarrow\texttt{TransactionCreated}$

Once the transaction has been created, operations join to it by passing its *id* number to the corresponding primitives. A transaction can complete by the explicit actions *commit* and *abort*, or by being automatically aborted when its lease expires. If the last case happens, the space sends an "exception". We model the exceptions by a communication action called *exception* parameterized with the *id* of the transaction. If a process creates a transaction, it has to add the possibility of receiving exceptions to all the actions executed until the commitment (or abortion) of the transaction. The signatures of the actions involved in the transaction mechanism are:

[2]We used the following name convention on the whole specification: we write with lower case the actions that are actively performed by a process, with upper case the ones that are reactive and by the initial letter the result of the communication. For example, take is actively performed by an external process, the space will react by doing Take (with upper case). Both actions will communicate and result in an action T (see section 6.4).

```
sort  Nat, Time
act   create: TransactionCreated×Time
      commit, abort, Exception: Nat
```

Transactions make changes on the semantics of the primitives, e.g. when a *write* action is performed under a transaction, the entry will be externally visible only after the transaction commits, and if the transaction is aborted no changes will be updated in the space. If a process inserts an entry under a transaction, and meanwhile another process executes a *readIfExists*, the second process will block to wait for the commitment of the transaction (or for the timeout).

We have presented, so far, enough ingredients of the μCRL interface to be able to specify the small example we have shown in the introduction. The example is modeled as a recursive process which renames entries of type A:

```
proc  renamer(id: Nat) =
      $\sum_{trc:TransactionCreated}$ create(trc, TIMEOUT)
          .(take(id, trcId(trc), FOREVER, isTypeA)
                         + Exception(trcId(trc)).$\delta$)
          .$\sum_{e:Entry}$ (TakeReturn(id, e)
                         + Exception(trcId(trc)).$\delta$)
            .(write(rename(e), trcId(trc), FOREVER)
                         + Exception(trcId(trc)).$\delta$)
          .(commit(trcId(trc)) + Exception(trcId(id)).$\delta$)
      .$\delta$
```

The transactional mechanism has been simplified, for example our model doesn't support nested transactions or transactions over multiple processes. The inclusion of these features will need a deeper insight on the Jini architecture from which JavaSpaces inherits the mechanism.

6.2.2 Notifications

Several entities are involved in the notification mechanism: The external processes that register their interest in future incoming entries. The space which fires events when a new incoming entry matches the *queries* associated to the registrations. The destinations of the events, called listeners, wait for the arrival of events and "react" to them and the network. Events travel through a network which is assumed to be unreliable. Notifications can be lost, duplicated, unordered,... note that it is not the same for takes, returns, et cetera which are performed reliably and never be lost.

The registration is done by the synchronous action *notify* based on the JavaSpaces specification [50].

```
EventRegistration notify(Entry tmpl, Transaction txn,
      RemoteEventListener listener, long lease,
      MarshalledObject handback)
   throws RemoteException, TransactionException;
```

The primitive gets, as arguments, the template to match entries, the reference to a transaction, the reference to the remote event listener, the lease and a handback used to pass information from the registered application to the listeners. If the transaction aborts all the registrations associated to the transaction are cancelled.

After a notify registration, the space returns an *eventRegistration* object, which includes the registration identification number (the space assigns a identification number to any new registration), the source of the events (in our case this will be always the space), the granted lease and the initial sequence number for events generated from the *notify* registration.

```
public class EventRegistration implements java.io.Serializable
{
    public EventRegistration(long eventID,
                             Object eventSource,
                             Lease eventLease,
                             long seqNum) {...};
    public long getID() {...};
    public Object getSource() {...};
    public Lease getLease() {...};
    public long getSequenceNumber() {...};
}
```

Every matching entry will increase by one the sequence number of the registrations. If the registration is associated to a transaction, only *written* entries inside the same transaction are counted. Newly generated events will contain a sequence number greater than the previous one. Remote listeners are modelled based on the interface below.

```
public interface RemoteEventListener extends Remote,
    java.util.EventListener
{
    void notify(RemoteEvent theEvent)
        throws UnknownEventException, ...;
}
```

In the μCRL model, we have abstracted away the handback of the notify registration. The template is replaced by a query, as in the previous operations. For simplicity, we assume the registration is done atomically, thus no exceptions can be fired between the beginning of the registration and its return. It would not have been difficult to model the operation with exceptions as we did for the transactions. Therefore, the initial sequence number of events will be zero. Due to these simplifications, the space will return an *eventRegistration* containing only the registration identification number and the lease identifier. The registration is performed reliably so it does not throw any exception when executing. The μCRL signature is:

sort Nat, Query, Time, EventRegistration
act notify: $\texttt{Nat}\times\texttt{Time}\times\texttt{Query}\times\texttt{EventRegistration}$

The arguments are: the transaction identification number, the lease, the query and the event registration (provided by the space).

Events sent by the space include the registration identification number to which they are directed and the sequence number, which can be used by listeners to know the number of events that occurred since the last notification.

An event listener is a process that reacts to the reception of an event and that may be running remotely. Every listener contains a method (*notify* [3]) invoked whenever it receives a notification event. According to the JavaSpaces specification the *notify* call is synchronous so the space waits for a listener until the call finishes, but the JavaSpaces implementations are multi-threaded hence many different notify calls can be done concurrently. We have modeled the *notify* operation with our single *javaspace* process, assuming that we have an implementation with enough threads to manage all the notifications of the system. We model the delivery in an asynchronous way, i.e., the μCRL space delivers the event and doesn't wait until the end of the method call of the listener. This policy will not be admissible if there are too many listener registrations or if the *notify* methods are very slow (as this would block the space for long periods) or never return.

μCRL does not allow dynamic creation of processes at runtime so listeners have to be defined at the beginning, according to the needs of the application. A listener has the following structure:

```
proc  listener(id: Nat) =
          $\sum_{\text{regId:Nat}}$ receiveRegId(id, regId).
          listenerActive(id, regId)

proc  listenerActive(id:Nat, regId: Nat, ...) =
          $\sum_{\text{seq:Event}}$ _Notify(regId, seq)
          .do_work
          .listenerActive(id, regId, ...)
```

The listener gets active when it receives from a process the registration number to which it is associated. Then, by the action *_Notify* receives events from the space. The *do_work* operation may be composed of any computation or any communication with other processes or with the space. The listener cannot be interrupted while it is performing some task. An example of a process that registers its listener for events is:

[3]Not to be confused with the registration notify method.

```
proc  p(...,listenerId:Nat,...) =
        ...
        $\sum$evReg:EventRegistration
          notify(NULL, FOREVER, isTypeA, evReg)
          sendRegId(listenerId, regId(evReg))
        ...
```

Events go through a unreliable network. The network can always lose, duplicate and reorder the events. We model the network by a new process composed in parallel with the space, the application processes and the listeners. More details about the network process are given in the next section. We use the auxiliary communicating actions *sendRegId* and *receiveRegId* to pass information between the listener and the process.

So far, we have presented the basic features of the JavaSpaces architecture from the external point of view. All the methods contained in the JavaSpaces API have been implemented but the *snapshot*. We consider that the later is not essential for verification purposes because it is only used to increase performance when the same entry is read several times. The next section is dedicated to introducing the implementation details.

6.3 Implementation Point of View

The *javaspace* process handles the concurrent access of the external applications to the common repository. It manages a data base storing the following data (the full specification of the space is given in the appendices):

- The shared entries (M).
- The active transactions (*Trc*).
- Pending look-up operations (PA).
- Notify registrations (*Reg*).

The process executes non-deterministically the actions which are enabled at any moment. The *javaspace*, following the requests of the external processes, can at any time add new entries, start look-up operations, return values to the the look-up actions, create, commit and abort transactions, decrement the time-outs, create registrations, collect expired entries, renew leases, ... The actions synchronise with an external process and update the data base of the space. All the operations are done atomically, and do not raise exceptions.

Entries are internally encapsulated in a sort called *Object*, which includes the entry, its lease and an identification number. The space automatically assigns a fresh *id* to every new entry. The *id* is used to manage the locks induced by the read operations performed under transactions. The signature of the *Object* sort is defined as follows:

```
sort  Object
func  object: Nat×Entry×Lease→Object
map   eq: Object×Object→Bool
      id:Object→Nat
      entry:Object→Entry
      lease:Object→Lease
      decT:Object→Object
      isExpired:Object→Bool
```

The function *decT* is used to decrement the lease of the Object. If the lease's timeout is *FOREVER* then it will never be decremented. *isExpired* returns true if the lease timeout is equal to 0.

Objects are stored in a data base (M) that has the structure of a multiset. The size of M is bounded by a constant (*maxObjects*). The entries are organised without any order, so when the space executes a search action, all of the matching entries have the same possibility to be selected.

When the space receives an external invocation of a *write*, it creates a new *object* and it inserts it into the set. The following fragment of code corresponds to a *write* action not attached to any transaction:

```
∑e:Entry, trcId:Nat, timeout:Time, objectId: Nat, leaseId: Nat
     Write(e, trcId, timeout)
     .javaspace(in(object(objectId, e, lease(leaseId, timeout)), M),
          PA, Trc, match(e, trcId, Reg))
     ◁ eq(trcId, NULL) ∧ ¬ eq(e, entryNull) ∧
       (lt(timeout, maxTime) ∨ eq(timeout, FOREVER)) ∧
       eq(objectId, freshId(U(M, UWsets(Trc)))) ∧
       lt(objectId, maxObjects) ∧
       eq(leaseId, freshId(U(leases(M), U(leases(Trc), leases(Reg))))) ∧
       lt(leaseId, maxLeases) ▷ δ
```

We see that the *write* action can be done under the following conditions:

- The *id* of the transaction is equal to *NULL*. As we said before, the code corresponds to the *write* without transaction. There is another summand of the *javaspace* process in charge of handling the cases under transaction.
- The entry to write is not *NULL*.
- The lease's timeout is either *FOREVER* or smaller than a given bound. There is a constant fixing the maximum timeout that can be granted (*maxTime*). This restriction applies to all timeouts appearing in the system.
- There is at least one fresh object *id* available. If the space is full, all the *ids* are assigned to some entries therefore no more entries may be inserted. The function *freshId* returns an unused *id* or *maxObjects*. The second

possibility implies that the space is full and the entry cannot be inserted. The objects are stored in M.

- The space checks if there are fresh *ids* in the union of all the places that can store objects: in M and in special write sets of the transactions.

- There is at least one fresh lease *id* available. The search for leases *ids* is done in the union all entities that can have associated leases.

After the execution of the action, the space updates the object set and checks if there are some notify registrations that match the new entry, in order to send a notification event. The matching of registrations is done by the function *match(e, trcId, Reg)*.

There is another summand handling the writes without transaction that also returns the lease *id* to the external process. In this case the code is exactly the same, the only difference is the action invocation which will include the lease's *id*: *Write(e, trcId, timeout, leaseId)*.

6.3.1 Look-up Operations

Regarding the look-up primitives: when the space receives a search request first it creates a *pending action*. A *pending action* includes: the type of action (*read*, *take*, ...), the transaction *id* to which it is associated, the query, the original value of the timeout and the current timeout. *Pending actions* are stored in a set *PA* of the sort *ActionSet*, we can see the full specification in the appendix A. The following code presents a *take* request without transaction:

```
Σprocld:Nat, trcld:Nat, timeout:Time, query:Query
        Take(procld, trcld, timeout, query)
        .javaspace(M, in(action(takeA, trcld,
                procld, timeout, query), PA), Trc, Reg)
        ◁ eq(trcld, NULL) ∧
          (lt(timeout, maxTime) ∨ eq(timeout, FOREVER)) ∧
          lt(len(PA), maxActions) ▷ δ
```

The action set *PA* has the same structure as the object set. It is also bounded by the constant *maxActions*. *Pending actions* do not have any identifier. If there is an object that matches one of the pending actions then the space returns the entry to the corresponding external process by means of the *return* action. An entry matches an action if the execution of the *match* operation of the associated query returns true. The return operation is not urgent, the space can postpone the return even if there are some matching entries. If there is a pending action with an expired timeout the space returns the *entryNull*. We give the code corresponding to a successful return of matching entry for a *take* invocation.

```
Σa:Action, o:Object
```

takeReturn(procId(a), entry(o))
.javaspace(rem(o, M), rem(a, PA), Trc, Reg)
$\lhd$ get(PA, a) $\wedge$ get(M, o) $\wedge$
eq(type(a), takeA) $\wedge$ eq(trcId(a), NULL) $\wedge$
match(query(a), entry(o)) $\wedge$ $\neg$ readLocked(o, Trc) $\rhd$ δ

The first three arguments of the return action are the ones that were provided in the look-up invocation. After the *take* the object is removed from the space, this will not be the case for the *read* returns. Before returning the entry, the space checks if the object is locked by some transaction, see section 6.3.2. We explain later the locking mechanism. Figure 6.2 graphically outlines the *look-up* mechanism:

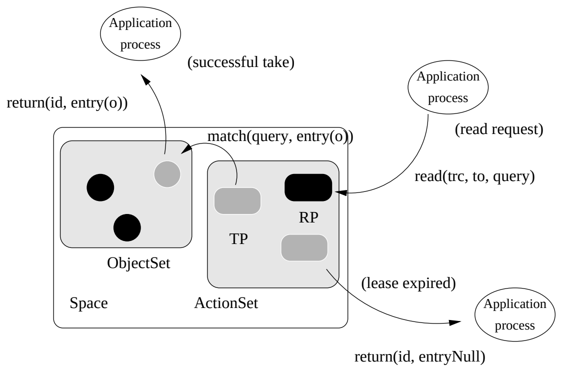


Figure 6.2: Look-up mechanism

6.3.2 Transactions

We continue introducing the signature of the sort transaction:

sort `Transaction`
func transaction: `Nat`$\times$`Lease`$\times$
`ObjectSet`$\times$`ObjectSet`$\times$`ObjectSet`$\rightarrow$`Transaction`

When created, every transaction receives a fresh identification number, 0 is reserved for the *NULL* transaction. Apart from the *id* and the associated lease, transactions have three object sets. The sets are used to trace the changes performed by the operations joined to the transaction:

- *Wset*: stores the entries written under a transaction. After a commit the objects are placed in the data base *M*.
- *Tset*: after a successful *take*, the matched object is removed from the space and is placed into *Tset*. If the transaction commits the object contained

in *Tset* are deleted. However, if the transaction is aborted the objects are put back in the space.

- *Rset*: stores the entries read under the transaction. When an object in *Rset* is read-locked, it cannot be taken outside the transaction. An object can be read-locked by several transactions.

We now present the creation operation:

```
∑timeout: Time, trcId: Nat, leaseId:Nat
        Create(trcCreated(trcId, leaseId), timeout)
        .javaspace(M, PA,
                in(transaction(trcId, lease(leaseId, timeout), emO, emO, emO),
                        Trc), Reg)
        ◁ eq(trcId, freshId(Trc)) ∧ lt(trcId, maxTrc)) ∧
          (lt(timeout, maxTime) ∨ eq(timeout, FOREVER)) ∧
          eq(leaseId, freshId(U(leases(M), U(leases(Trc), leases(Reg)))))) ∧
          lt(leaseId, maxLeases) ▷ δ
```

Initially, the sets associated to the transaction are set to empty. As in the previous cases the space first tries to allocate the needed resources by searching fresh *ids*. The following μCRL code corresponds to the commit action:

```
∑trcId: Nat, trc: Transaction
        Commit(trcId)
        .javaspace(U(M, Wset(trc)), PA),
                rem(trc, Trc), remRegs(id(trc), Reg))
        ◁ get(Trc, trc) ∧ eq(trcId, id(trc))▷ δ
```

When the transaction commits the new object set of the space will be the union of M with the write set of the transaction. After the commitment the entities associated with the transaction (*pending actions* and *registrations*) are removed. The abort operation is as follows:

```
∑trcId: Nat, trc: Transaction
        Abort(trcId)
        .javaspace(U(M, Tset(trc)), PA),
                rem(trc, Trc), remRegs(id(trc), Reg))
        ◁ get(Trc, trc) ∧ eq(trcId, id(trc))▷ δ
```

Note that the only difference with the commit is that the new set of objects is the old set plus the objects stored in the *take* set of the transactions. If a process executes a *readIfExists* or *takeIfExists* and there is no matching object in the space, we check in the *Wsets* and *Tsets* of the other transactions to decide whether the process has to block or not.

Exceptions are thrown when the timeout of a transaction expires. The exception is a synchronous action that communicates with the process that holds the transaction identifier. In case of an exception, the changes performed under the transaction are *rolled back*. Let us see the definition:

```
∑trc: Transaction
      exception(id(trc))
      .javaspace(U(M, Tset(trc)),
            remPAs(id(trc), PA), rem(trc, Trc), remRegs(id(trc), Reg))
      ◁ get(Trc, trc) ∧ isExpired(trc) ▷ δ
```

Figure 6.3 summarises the mechanism of transactions.

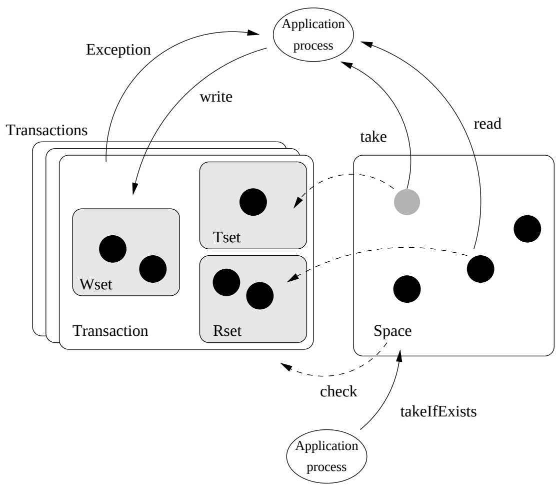


Figure 6.3: Transactional model

The specification of the rest of the operations under transactions, such as *write* under a transaction, can be found in the appendix of the thesis.

6.3.3 Notifications

Notify registrations are defined as follows:

```
sort   Registration
func   registration: Nat×Nat×Lease×
                     Query×Nat×Bool→Registration
```

The fields of the registration are: the identification number, the listener *id* to which the events are directed, the lease, the query that match the incoming entries, the sequence number of matching entries, and a boolean used to know whether the space has to send new events or not. The following code presents registration:

```
∑timeout: Time, trcId: Nat, leaseId:Nat, regId:Nat, query:Query
        Notify(trcId, timeout, query, evRew(regId, leaseId))
        .javaspace(M, PA, Trc,
                in(registration(regId, trcId, lease(leaseId, timeout),
                        query, 0, F), Reg))
        ◁ eq(regId, freshId(Reg)) ∧ lt(len(Reg), maxRegistrations)) ∧
          (lt(timeout, maxTime) ∨ eq(timeout, FOREVER)) ∧
          eq(leaseId, freshId(U(leases(M), U(leases(Trc), leases(Reg)))))) ∧
          lt(leaseId, maxLeases) ▷ δ
```

After every successful insertion of an entry the space increments the counters associated with the notify registrations. The sending of events can be postponed. At any moment the space can decide to send a notification event to the corresponding process. Events are model by the following sort:

```
sort   Event
func   event: Nat×Nat →Event
```

An event contains the reference to the registration identifier to which it is directed and the value of the counter of notifications. Messages travel over the network from the space to the event destination (the listener); they are not delivered instantaneously nor reliably. Hence events may be lost and never reach their destinations. We model the network with the following process:

```
proc   Network(E:EventList) =
       ∑e:Event _Notify(e).Network(in(e, E))
       ◁ lt(len(E), maxEvents) ▷ δ+
       ∑e:Event _notify(registrationId(e), seq(e))
       .Network(rem(e, E))
       ◁ get(E,e) ▷ δ+
       duplicate.Network(in(e, E))
       ◁ get(E,e) ∧ lt(len(E), maxEvents) ∧ lt(0, len(E)) ▷ δ+
       lose.Network(tail(E))
       ◁ lt(0, len(E)) ▷ δ
```

The network is implemented by a process that manipulates a finite list of events (see implementation in section A.3. The process can: receive an event from the space (*_Notify*), deliver an event to a listener (*_notify*), duplicate or lose the first event of the list. Note that the network can deliver the events in any order. We have used the following conventions on action names, in section 6.4 one can find how the actions communicate:

- *notify*, is the action performed by a process to register for events.

- *Notify*, is the action performed by the space to register for events.
- *_notify*, is the action performed by the space to send an event.
- *__Notify*, is the action performed by the network to receive an event.
- *__notify*, is the action performed by the network to deliver an event.
- *_Notify*, is the action performed by a process to receive an event.

Applications using the notification mechanism are composed by the parallel composition of the javaspace process, the application processes, the listeners and the network. Figure 6.4 illustrates the notification mechanism.

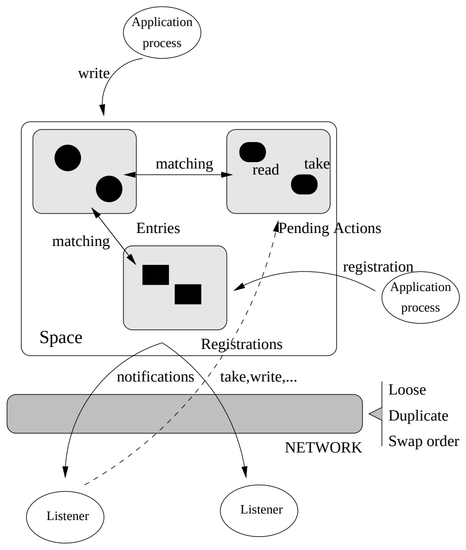


Figure 6.4: Notification architecture

We also added leasing to the registration mechanism. We proceed in the same way as for the look-up primitives. The application process passes the requested lease to the space which includes this value in a data field of the registration object. When the registration lease expires the space automatically removes the registration from the data base without further communications. Listeners can receive events even if the registration has been removed, because the messages may be delayed on the network.

Notify can also be joined to a transaction. The space will send events when a matching entry is written under the same transaction of the registration or under the *null* transaction. If a transaction expires the joined registrations will be removed.

6.3.4 Leasing

The leasing mechanism as the transactions and event notifications, is part of the Jini architecture. Leases are defined by the following sort:

```
sort  Lease
func  lease: Nat×Time→Lease
map   eq: Lease×Lease→Bool
      id:Lease→Nat
      timeout:Lease→Time
      decT:Lease→Lease
      isExpired:Lease→Bool
      hasTimeout:Lease→Bool
```

A new lease is created after the insertion of an entry, a notify registration and the creation of a new transaction. All leases have a different identification number which allows external agents to renew their timeout. *hasTimeout* returns true when the lease's timeout is different from *FOREVER*. The rest of the functions are identical to the ones of the *Object* set. The following fragment of code implements the action of renewing the timeout of a lease which is associated to an *Object*:

```
∑leaseId:Nat, timeout:Nat, o:Object
        Renew(leaseId, timeout)
        .javaspace(in(object(id(o), entry(o), lease(leaseId, timeout)), rem(o,M)),
                PA, Trc, Reg)
        ◁ get(M, o) ∧ eq(id(lease(o)), leaseId) ▷ δ
```

Note that the renewal is always granted by the space, and the new timeout will be the one requested by the external process. We can do the same to leases associated to transactions or registrations. In fact we could have implement a set of leases as we did for pending actions, objects or registrations.

The space manages a centralised discrete clock. Under some constraints, the space can tick which cause the decrement of all timeouts. Between two time *ticks* many actions may happen. Externally several actions can be performed in parallel (in the sense of interleaving). Here is the code that describes the behaviour of the clock:

```
tick.javaspace(decT(M), decT(PA), decT(Trc), decT(Reg))
◁ (areTimeouts(M) ∨ areTimeouts(PA) ∨
areTimeouts(Trc) ∨ areTimeouts(Reg)) ∧
(¬ areExpired(M) ∨ ¬ areExpired(PA) ∨
¬ areExpired(Trc) ∨ ¬ areExpired(Reg)) ▷ δ
```

The action *tick* can only happen if there are some entities in the space with an active timeout (greater than 0 and not equal to *FOREVER*). This is done by including the condition *areTimeouts*, the idea is avoid self *tick*-loops, which can hide deadlocks in the system. If there are some expired timeouts the space has to first perform the operations that correspond to the expiration.

6.4 Putting All Together

In summary, the *javaspace* process can at any time:

- Receive request of services: look-ups or insertions.
- Match entries with pending actions, sending the result to the external processes.
- Create, commit or abort transactions.
- Decrement timeouts.
- Renew leases.
- Accept notify registrations. And send events to some listener through the network.
- Perform actions related to the timeout expirations: delete expired entries or registrations, abort transactions by sending the corresponding exception and return null entries for the unmatched look-up operations.

We have presented separately the javaspaces specification from two different point of view: from the side of the external applications and from the side of the implementation. In order to build a complete system and to apply automated verification to it, we have to consider some more issues.

As we have seen in section 6.2, a complete system is composed by the parallel composition of the javaspace process with the external applications, as follows:

$$\begin{aligned}\mathsf{System} = \tau_I \partial_H(&\mathsf{javaspace}(...) \parallel \mathsf{external\_p_0}(...) \parallel \mathsf{external\_p_1}(...) \parallel ...\\ &\mathsf{NetWork(emptyList)} \parallel \mathsf{listener_0}(...) \parallel \mathsf{listener_1}(...) \parallel ...)\end{aligned}$$

∂_H forces communication of some specific actions. The network and listeners are put in parallel when needed. Let us, first, show the full declaration of actions (some of them have been described in the previous sections):

```
act  write, Write, W : Entry×Nat×Time
     write, Write, W : Entry×Nat×Time×Nat
     take, Take, T : Nat×Nat×Time×Query
     read, Read, R : Nat×Nat×Time×Query
     readIfExists, ReadIfExists, RIE : Nat×Nat×Time×Query
     takeIfExists, TakeIfExists, TIE : Nat×Nat×Time×Query
     takeReturn, TakeReturn, TReturn :Nat×Entry
     readReturn, ReadReturn, RReturn : Nat×Entry
     readIfExistsReturn, ReadIfExistsReturn, RIEReturn : Nat×Entry
```

takeIfExistsReturn, TakeIfExistsReturn, TIEReturn : Nat×Entry
create, Create, C : TransactionCreated×Time
commit, Commit, Cm : Nat
abort, Abort, A : Nat
exception, Exception, E : Nat
notify, Notify, N: Nat×Time×Query×EventRegistration
_notify, __Notify, _N: Event
__notify, _Notify, __N: Nat×Nat
tick
duplicate lose
gc: Entry
gc: Registration
renew,Renew,Rnew: Nat×Time

We recall that we have used the following convention: we write with lower case the actions that are actively performed by a process, with upper case the ones that are reactive and by the initial letter the result of the communication. The definition of the communication is as follows:

comm write | Write = W
take | Take = T
read | Read = R
readIfExists | ReadIfExists = RIE
takeIfExists | TakeIfExists = TIE
takeReturn | TakeReturn = TReturn
readReturn | ReadReturn = RReturn
readIfExistsReturn | ReadIfExistsReturn = RIEReturn
takeIfExistsReturn | TakeIfExistsReturn = TIEReturn
create | Create = C
commit | Commit = Cm
abort | Abort = A
exception | Exception = E
notify | Notify = N
_notify | __Notify = _N
__notify | _Notify = __N
renew | Renew = Rnew

All the actions above (*write, Write, take, Take, read,...*) are forced to communicate, therefore they are included in the set H of ∂_H. Furthermore, the result of the communication (*W, T, R, ...*) is normally hidden. To hide actions we use the operator τ_I.

We have required that the system must be suitable to do model checking. To satisfy the requirement we need to generate finite state spaces. We have defined a set of constants that are used to define bounds on the different data structures. These constants are: *maxObjects, maxActions, maxEvents, maxTime, maxRegistrations, maxLeases, maxTransactions*. The definition of the constants can be seen in the appendix. The setting of these constants allows to manipulate different instances of the system.

6.5 Conclusion

For brevity, we have not explaining in full detail all the operations that the space can perform. However we include in the appendix A the full specification of the *javaspace* process. It has around 1000 lines of code (more than 1500 lines together with the data specification). Even if the model captures the most part of the functionalities of the architecture, there are a few interesting features that are not included, such as for example: nested transactions or use of multiple spaces. We believe that the later might be included without major problems, the former may need some deeper inside in the Jini's specification of transactions.

The model has been created from the informal specification provided by Sun. An important question is how to validate the correctness of the model with respect to the informal description. Validation has been done by means of the analysis of small examples whose behaviours are predictable [67], and by methodic simulation of all the features. Another possible way is to compare the specification with another formal model, such as the one presented in [22] and [24] (which only contains the basic operations for reading, taking and writting and the notification mechanism that is modelled separately). However our model contains more features which makes a comparison difficult. In any case, more effort should have dedicated for validation of the model by doing, for instance, exhaustive testing.

As we have said the main aim of the model it to verify applications implemented on top of JavaSpaces. The next chapter is dedicated to describing the use of the model for verification purposes. We will see that the model is suitable to verify interesting and non-trivial applications, which was one of the initial requirements. The model might be used also to verify meta-properties of the architecture, as done in [23]. In that case we would have to extract some correctness criteria from Sun's specification and to check these criteria against the μCRL model.

Chapter 7

Verification of JavaSpaces Applications

In the previous chapter, we have presented a formal model of the JavaSpaces architecture. This chapter is dedicated to introducing a verification methodology for JavaSpaces applications. In the first part of the chapter, we illustrate the usage of the framework by proving some properties on some simple examples. The last part is dedicated to the study of a non-trivial fault-tolerant application that solves a typical coordination problem. The problem consists of the computation of an extensive task, performed in parallel by splitting it into smaller and more manageable parts. This sort of problems is specially well-suited to be implemented using JavaSpaces.

7.1 Verification Methodology

The chapter is composed by a set of small JavaSpaces examples, and by an exhaustive analysis of a realistic application. The methodology we are going to use to verify the systems is:

1. First, we specify in μCRL the application we want to analyse. We will follow the premises presented in section 6.2 of the previous chapter.
2. Then, we compose the external components with the formal model of JavaSpaces.
3. We formally describe the properties that we want to check. The language used to write the properties is the regular alternation-free μ-calculus introduced in Chapter 3, section 3.5.
4. Using the μCRL tool set, we generate the state space corresponding to the full system.
5. Finally, using the CADP model checker [43], we prove the properties correct. If some property is not satisfied we analyse the counter-example to extract some information about the error. Note that (4) and (5) can be done on-the-fly without generating the full state space.

Model checking in general can only handle finite state spaces. Therefore, in some cases, we will have to restrict the system to a fixed number entries, processes, events... in order to obtain a finite system. These numbers are typically small. The next chapter is dedicated to showing how to generalise the model checking results to any parameter by using abstraction techniques.

7.2 Playing with JavaSpaces

The following examples are inspired by the classical arcade game Ping-Pong, in which two players throw one ball from one to the other. This example has been taken from Chapter 5 of the book "JavaSpaces^{TM} Principles, Patterns, and Practice" [50]. The players are modeled by two processes called Ping and Pong which communicate by means of an entry that encapsulates the ball. We, first, propose a very simple version of the game, later we will make some small changes to the game rules.

Simple Ping-Pong: In the first version, players can only catch and throw the ball. The Entry *sort* (ball) is defined as follows:

```
sort  Entry
func  entryNull:→Entry
      ball: Name→Entry
map   eq:Entry×Entry→Bool
      receiver:Entry→Name
```

```
var   e: Entry
      n: Name
      receiver(ball(n)) = n
```

The only field the entry has is the name of the player whom the ball is directed to. The name is from the sort *Name*, that has two constructors *Ping* and *Pong*, and one function (*other*) used to switch from one to the other (*other(Ping) = Pong and other(Pong) = Ping*). To get the ball from the space, a player uses a query:

```
sort  Query
func  forMe: Name→Query
      eq: Query×Query→Query
map   match: Query×Entry→Bool
var   e: Entry
      n': Name
rew   match(forMe(n), e) = eq(n, receiver(e))
      eq(forMe(n), forMe(n')) = eq(n, n')
```

The code of both players is the same:

```
proc  player(id:Nat, name:Name) =
      take(id, NULL, FOREVER, forMe(name))
      .Σ_{e:Entry} TakeReturn(id, e)
          .print(name)
          .write(ball(other(name)), NULL, FOREVER)
      .player(id, name)
```

Print is an external action used to communicate to the environment that a player has caught the ball and is going to throw it back. In the initial state the space includes a ball directed to *Ping*. The values of the other main data structures (TransactionSet, PendingActionSet,. . .) are initialised to empty. The system instantiation is as follows:

$$\begin{aligned}System = {} & \tau_{\{\text{W, E, Rt}\}}\partial_{\{\text{write, Write, take, Take, takeReturn, TakeReturn}\}}\\ & (javaspace(\text{in(object(0, ball(Ping), lease(0, FOREVER)), emO)},\\ & \quad \text{emT, emA, emR})\\ & \,||\, player(\text{0, Ping}) \,||\, player(1, \text{Pong}))\end{aligned}$$

We can prove, for example, a safety property expressing the prohibition of “bad” execution sequences, such as the player *Ping* cannot throw the ball twice in a row:

(A1): [T* . ”print(Ping)” . (¬ ”print(Pong)”)* . ”print(Ping)”] F

The following property expresses that after a *Ping*, the action *Pong* is eventually reached:

(A2): $[\ \mathsf{T}^* \ .\ \text{"print(Ping)"}]\langle \mathsf{T}^* \ .\ \text{"print(Pong)"}\ \rangle\ \mathsf{T}$

Both properties are satisfied by the model. The state space has only 23 states and 33 transitions.

Timed Ping-Pong: Now, we introduce a small change in the rules of the game. In this version, once a player has caught the ball, he has one time unit to put it back into the space, otherwise he loses the game. We model this approach by using transactions. After a player has performed *take*, he creates a transaction leased for one time unit. When the *write* operation is performed, the transaction can safely commit. Let us see the processes code:

```
proc  player(id: Nat, name:Name) =
      take(id, NULL, FOREVER, forMe(name))
      .$\sum_{e:Entry}$ TakeReturn(id, e)
         .$\sum_{trc:TransactionCreated}$ create(trc, tt(1))
          .(write(ball(other(name)), trcId(trc), FOREVER)
            + Exception(trcId(trc)).loser(name))
          .(print(name) + Exception(trcId(trc)).loser(name))
          .(commit(trcId(trc))) + Exception(trcId(trc)).loser(name))
      .player(id, name)

proc  loser(name:Name) = theLoser(name).$\delta$
```

The complete system contains 51 states and 95 transitions. The properties **A1** and **A2** are also satisfied by this system, however the following property that states that after a *Ping*, *Pong* is inevitably reached is not:

(A3): $[\ \mathsf{T}^* \ .\ \text{"print(Ping)"}]\ \mu X.\ (\langle\ \mathsf{T}\rangle\ \mathsf{T} \wedge [\neg\ \text{"print(Pong)"}]\ X)$

The property is not satisfied, due to the fact that after a *Ping* the player *Pong* can receive a timeout exception and therefore lose the game.

In the formula **A3**, for all traces that contain a *Ping* a fixpoint computation is started. The variable X represents the sets of states that satisfy the full formula on each iteration of the computation. The formula is satisfied if the initial state is in X when the fixpoint is reached. Starting from the empty set of states ($X = \emptyset$), the first iteration will actualise the variable X by adding all the states that have at least one outgoing transition ($\langle\ \mathsf{T}\rangle\ \mathsf{T}$) and all transitions different to `print(Pong)` go to the empty set, in order words the states that only have `print(Pong)` transitions. The following iterations will add the states that have all transitions going to the states belonging to the previous iteration.

Ping-Pong with events: Now, we give a different implementation of the game by using the notification mechanism. Each player subscribes to the entries that encapsulate balls directed to them. The associated listeners are in charge of collecting the ball and sending it back.

```
proc  player(name:Name) =
      $\sum_{\text{reg:EventRegistration}}$ notify(NULL, FOREVER, forMe(name), reg)
         .sendRegId(name, regId)
      .(write(ball(Ping), NULL, FOREVER) $\lhd$ eq(Pong, name) $\rhd$ $\delta$)
      .$\delta$
```

The player *Pong* writes the first ball into the space which is initially empty. The action *send* is used to activate the listener, whose code is:

```
proc  listener(name:Name) =
      $\sum_{\text{regId:Nat}}$ receiveRegId(name, regId)
         .listenerActive(name, regId)

proc  listenerActive(name:Name, regId: Nat) =
      $\sum_{\text{seq:Nat}}$ (_Notify(regId, seq) $\lhd$ lt(seq, MaxGame) $\rhd$ $\delta$)
      take(regId, NULL, FOREVER, forMe(name))
      .$\sum_{\text{e:Entry}}$ TakeReturn(regId, e)
         .print(name)
         .write(ball(other(name)), NULL, FOREVER)
      .listenerActive(name, regId)
```

Listeners only receive events with a sequence number below a maximum (this constraint is imposed to avoid that the sequence numbers can grow until infinity, generating an infinite state space). After the arrival of an event, the listener takes the corresponding ball and writes it back to the space.

In this case the full system is composed by the players the listeners and the network composed in parallel with the space. The system has 12448 states and 34093 transitions, for *MaxGame* equal 2.

$$\begin{aligned} System = {} & \tau_{\{\ldots\}}\partial_{\{\ldots\}}(javaspace(\mathsf{emO, emT, emA, emR}) \\ & \,||\, player(\mathsf{Ping}) \,||\, player(\mathsf{Pong}) \\ & \,||\, listener(\mathsf{Ping}) \,||\, listener(\mathsf{Pong}) \,||\, Network(\mathsf{emEv})) \end{aligned}$$

Formula **A1** is satisfied by the system, however **A2** is not. This is because listeners can get blocked. This will happen when the network duplicates an event and lets a listener think that there is no such a ball directed to itself when there is not. A combination of these behaviours leads to a deadlock in the system. **A3**, obviously, does not hold either.

7.3 Parallel Summation

Now, we address the problem of fault-tolerant parallel summation of a multiset of numbers [108, 59]. The main difficulty of this application is how to determine when the summation is completed. Among the different possibilities to solve the problem, we propose an algorithm consisting of a number of identical processes (*Workers*) that independently perform simple additions and a *Master*, that is a special worker, who is charged to publish the result when the complete sum is accomplished. First, we present a naive (and wrong) implementation of this idea. Next, we will impose on the system extra non-functional requirements and give a correct solution to the problem.

The following two fragments of Java code implement the wrongly terminating solution. Note that we do not show the definition of auxiliary classes and the initialisation of the system. The entries in the space are instances of the class *Number* which encapsulates a natural number:

```
while(true){
   e1 = (Number) space.take(anyNumber, NULL, 0);
   if(e1 == null){return;}
   e2 = (Number) space.take(anyNumber, NULL, 0);
   if(e2 == null){
      space.write(e1, NULL, Lease.FOREVER);
      return;
   }
   space.write(e1.plus(e2), NULL, Lease.FOREVER);
   println("Worker wrote: " + e1.plus(e2));
}
```

A *Worker* first tries to take two entries, one after the other, by performing non-blocking takes matching any number in the space. If he succeeds then he writes the addition and loops, otherwise he undoes the changes (if needed) and halts. The calls to the method *take* get three parameters: First, the template which is also an instance of the class *NUMBER*. Second, the reference to the transaction which is equal to *NULL* as the action is not performed within a transaction. And finally, the timeout of the action, *0* means that if there are no entries in the space the method will not wait. The method *write* receives the entry to be stored, the transaction reference and the lease. The constant *FOREVER* is used to place objects that will never be removed by the space. Let us, now, present the code of the *Master*:

```
while(true){
   e1 = (Number) space.take(anyNumber, NULL, 0);
   if(e1 == null){return;}
   e2 = (Number) space.take(anyNumber, NULL, 0);
   if(e2 == null){
      System.out.println("Master publish:" + e1);
      return;
```

```
        }
        space.write(e1.plus(e2), NULL, Lease.FOREVER);
        println("Master wrote: " + e1.plus(e2));
    }
```

The *Master* is similar to the *Workers*, but if the second take does not succeed the *Master* publishes the value of the first entry as the final result. In case both takes return an entry he performs the normal addition and continues. Note that this solution will lead to incorrect publications, for example: a *Worker* may take the last but one item, and while it is busy, the *Master* might take the last one and think that the algorithm has terminated. Before examining how the *Master* can be sure that he took the last element, let us consider another important issue.

We extend the system with faulty behaviour, i.e. any process may suddenly stop and restart. But if a worker crashes after taking a number, this number would be lost. This forces us to consider the use of transactions. To guarantee non-corruption of the data due to the failure of a process, the critical operations have to be encapsulated by a transaction. In case a *Worker* halts or fails in the middle of an operation, the space will automatically recover to a stable state, undoing the modifications.

Recall that transactions are subject to leasing. Now the lease on the *Workers'* transactions provides an upper bound on the duration of a simple summation operation. We choose this timeout (from now denoted t_{op}) to be sufficiently large to perform one addition. It can be approximated by the estimated duration of a simple addition plus the latencies of the coordination primitives. We can, now, propose a mechanism that guarantees the exclusive access of the *Master* to the data:

1. When the *Master* is willing to check termination, he starts a transaction and writes a special entry (*lock*) to prevent *Workers* to start new operations. Thus, *Workers* have to check the non-existence of the *lock* entry in the space before starting a new addition.
2. Then, the *Master* waits until the end of all possible active operations, i.e. he has to block longer than the upper bound of the *Workers'* operations (at least $\mathrm{t}_{op} + 1$ time units). We denote this timeout with t_{opM}.
3. Now, he is sure he has exclusive access, i.e., no *Worker* has an entry, and no *Worker* can take any of them. Then:
   - If there is only a single entry in the space the *Master* publishes it, commits the transaction and halts.
   - If there are two entries he performs a simple addition, puts the result in the space, removes the lock, commits the transaction and waits until he decides to restart the termination test again.

The *Master*'s operations have to be executed under a transaction to prevent problems such as for instance: the failure of the *Master* after having locked the space will forbid any more progress by *Workers*. The timeout of the transaction (t_{ma}) has to be sufficient to guarantee that the process can perform the steps of the protocol. t_{ma} can be underapproximated by: t_{opM} plus the estimated time to perform a simple sum, plus the latencies of the involved coordination primitives. Now, let us, first, see the body of the *Master* process and then the explanation of some details of the implementation:

```
while(true){
   try{
      Transaction txn = trcManager.create(t_ma);
      space.write(lock, txn, Lease.FOREVER);
      space.take(noEntry, txn, t_opM);
      e1 = (Number) space.take(anyNumber, txn, 0);
      if(e1 == null){
         txn.abort();
         return;
      }
      e2 = (Number) space.take(anyNumber, txn, 0);
      if(e2 == null){
         txn.commit();
         println("Master publish: " + e1);
         return;
      }
      space.write(e1.plus(e2), txn, Lease.FOREVER);
      space.take(anyLock, txn, 0);
      txn.commit();
      println("Master wrote: " + e1.plus(e2));
   } catch (Exception e){} //loop
   Thread.sleep(t_wait);
}
```

The *Master* first creates a transaction and locks the space by writing a *lock* entry. The *lock* is written inside the transaction, therefore it will not be externally visible except that it blocks *IfExists* actions. Then, the *Master* has to wait until the end of the active operations (t_{opM}). Since we make no assumptions on the relative speed of the clock of different processes, we cannot use local primitives as `Thread.sleep(T_opM)` to perform the wait. However, we can use synchronization between the *Master* process and the space, by reading with a template that matches nothing (*noEntry*). This operation will always block during t_{opM} time units. After the *null* return of this primitive, he has exclusive access and can test the completion of the algorithm.

The behaviour of the generic *Workers* is similar to the naive version. But now the operations are executed under a transaction, leased for t_{op} time units. In case a *Worker* receives an exception due to the expiration of the transaction's timeout, he just restarts the algorithm again. Another change is that before every addition, a worker has to check whether the space is locked or not by the *Master*. This test is done by means of a *ReadIfExists* primitive which only blocks if there are matching entries with conflicting transaction locks. Note that the *Master* is never going to free the *lock* before removing it, so, this operation can only result in a *null* return, which allows *Workers* to continue their tasks, or a transaction exception which will force them to restart.

```
while(true){
   try{
      Transaction txn = trcManager.create(t_op);
      space.readIfExists(lock, txn,Long.MAX_VALUE);
      e1 = (Number) space.take(anyNumber, txn, 0);
      if(e1 == null){
         txn.abort();
         continue; // loop
      }
      e2 = (Number) space.take(anyNumber, txn, 0);
      if(e2 == null){
         txn.abort();
         continue; //loop
      }
      space.write(e1.plus(e2), txn, Lease.FOREVER);
      space.take(anyLock, txn, 0);
      txn.commit();
      println("Worker wrote: " + e1.plus(e2));
   } catch (Exception e){} //loop
}
```

Since all the critical actions are encapsulated in transactions, all agents can arbitrarily fail and restart without corrupting the information of the system. However, to detect the completion of the sum one *Master* should be alive sufficiently long.

The application allows to have any number of running *Workers*. However, replication of the *Master* would lead to incorrectness. Nevertheless, it's possible to imagine a complete replicable application in which all the processes are equal and the role of *Master* or *Worker* is assigned by a special entry or token. In this case, if the actual *Master* dies the *Workers* will compete for the token. Note that this solution will require to manage two different transactions, one as in the previous processes and another to deal with the new token.

The proposed algorithm tries to maximise the efficiency of the computation by allowing as many operations in parallel as possible. Note that *Workers* do not compete between each other for any resource, so they run completely in parallel. But, performance of the system depends on the selection of accurate upper bounds on the simple additions, the rate of test for termination, which is given by the time the *Master* waits between consecutive loops (t_{wait}) and, of course, on the number of active *Workers* and the reliability of the processes. t_{wait} can be tuned according to the estimated total duration.

If we knew a priori the amount of numbers in the space, we might use a counter, storing this number and decreasing it after every successful operation, this solution will have a negative effect on the performance of the system due to the concurrent access to this shared entry. Other similar solutions based on a shared data structure will suffer from the same handicap.

Another possible approach to solve the problem could be based on the dispatch of notification events after successful additions which would allow processes to control the number of entries left in the space and determine termination. This solution presents difficulties due to the, by specification, unreliable distribution of events, i.e. events may be lost, duplicated or unordered, and the verification of the algorithm would be unfeasible.

Even if the basic idea of the algorithm is rather simple, the proposed solution deals with quite complicated features: mutual exclusion, transactions and relations between timeouts. Therefore we cannot immediately claim that the algorithm is correct, i.e., that the master publishes the expected result. In the following section we are going to use a formal procedure to prove correctness.

7.3.1 Verification

In order to verify the correctness of the proposed algorithm, we first translate it to μCRL. Process expressions are obtained by systematic manual translation of the Java code presented above. Let us first see the *Master*'s code:

```
proc  Master(id: Nat) =
      $\sum_{trc:Nat,\ leaseId:Nat}$ create(trcCreated(trc, leaseId), $t_{ma}$)
        .(write(lock, trc, FOREVER) + ExM(id, trc))
        .(take(id, trc, $t_{opM}$, noEntry) + ExM(id, trc))
        .(TakeReturn(id, entryNull) + ExM(id, trc))
        .(take(id, trc, tt(0), anyNumber) + ExM(id, trc))
        .($\sum_{e1:Entry}$ TakeReturn(id, e1)
          .(((take(id, trc, tt(0), anyNumber) + ExM(id, trc))
          .($\sum_{e2:Entry}$ TakeReturn(id, e2)
            .((commit(trc) + ExM(id, trc))
            .publish(value(e1)).$\delta$
            $\triangleleft$ eq(e2, entryNull) $\triangleright$
            (write(plus(value(e1), value(e2)), trc, FOREVER)
                  + ExM(id, trc))
            .(take(id, trc, tt(0), anyLock) + ExM(id, trc))
            .(TakeReturn(id, lock) + ExM(id, trc))
            .(commit(trc) + ExM(id, trc))
```

```
              .Master_wrote(plus(value(e1), value(e2))))
             + ExM(id, trc)))
             ◁ not(eq(e1, entryNull)) ▷
             abort(trc).δ)
           + ExM(id, trc))
         .Master(id)
```

ExM(trc) encodes the possibility of receiving an exception due to the timeout of the transaction. In that case, the *Master* restarts the algorithm:

```
proc  ExM(id:Nat, trc:Nat) = Exception(trc).Master(id)
```

Now, we present the *Workers*' code:

```
proc  Worker(id:Nat) =
      $\sum_{trc:Nat,\ leaseId:\ Nat}$ create(trcCreated(trc, leaseId), t_op)
        .(readIfExists(id, trc, FOREVER, anyLock) + ExW(id, trc))
        .($\sum_{l:Entry}$ ReadIfExistsReturn(id, l)
          .((take(id, trc, tt(0), anyNumber) + ExW(id, trc))
          .($\sum_{e1:Entry}$ TakeReturn(id, e1)
            .((take(id, trc, tt(0), anyNumber) + ExW(id, trc))
            .($\sum_{e2:Entry}$ TakeReturn(id, e2)
              ((write(plus(value(e1), value(e2)),trc,FOREVER)
                   + ExW(id, trc))
              .(Worker_wrote(plus(value(e1), value(e2)))
                   + ExW(id, trc))
              .commit(trc)
              ◁ not(eq(e2, entryNull)) ▷
              abort(trc).Worker)
            + ExW(id, trc))
            ◁ not(eq(e1, entryNull)) ▷
          abort(trc).Worker)
        + ExW(id, trc)
        ◁ eq(l, entryNull) ▷
        abort(trc).Worker)
      + ExW(id, trc))
      .Worker(id)

proc  ExW(id:Nat, trc:Nat) = Exception(trc).Worker(id)
```

Model checking techniques are (mostly) restricted to systems with finite (and small) state spaces. Therefore, we will have to limit our system by fixing the number of workers, and the amount of entries initially included in the repository. We will perform a model checking analysis for different instances of these values. Furthermore, we fix the following constants:

$$\mathsf{t_{ma}} = \mathsf{tt(2)} \quad \mathsf{t_{opM}} = \mathsf{tt(1)} \quad \mathsf{t_{op}} = \mathsf{tt(0)}$$

As we said in the previous section, t_{ma} has to be greater than t_{opM}, and the latter greater than t_{op}.

The correctness claim of the algorithm is that successful termination is detected, i.e. the *Master* publishes the correct result. Therefore, we would like to prove a property meaning that on all possible paths, starting from the initial state, the action `publish(n)` will be reached, with `n` representing the correct result. This is expressed by the "inevitable reachability" property, as follows:

(B1): μX. (⟨ T⟩ $\mathsf{T}\wedge$ [¬ 'publish(n)'] X)

The formula does not hold for the given specification. The reason is that processes may fail continuously which does not allow any progress in the computation. We can have an infinite loop in which the *Master* creates a transaction and, then, the transaction is aborted by an exception before the completion of the assigned task. What we can prove is that there exists at least one path that leads to the publication of the result, expressed by the following formula:

(B2): ⟨ T* . "publish(n)" ⟩ T

The formula is satisfied but is not strong enough in order to believe that the system acts correctly. The reason is that the system may contain deadlocks or livelocks that are not detected by the formula. We are going to prove a property that expresses that publish is reached in all fair traces.

(B3): [(¬ 'publish(n)')*]⟨(¬ 'publish(n)')* . 'publish(n)'⟩ T

The property says that for all possible traces (without publish) there exists a path that leads to *publish.* Another possibility to prove the stronger property "inevitable reachability" is based in the imposition of a constraint on the behaviour of the processes. We assume some sort of reliability of the process. For instance, the *Master* process may fail any number of times but at some time it becomes *safe* and does not fail anymore. To implement the idea we specify a new *Master* process which is a copy of the previous one but with the difference that the *safe Master*'s transaction cannot expire, i.e., it's leased *FOREVER.* When the *unsafe* process gets an exception, it chooses non-deterministically to become *safe* or to stay *unsafe.* We indicate the first choice by adding an external action to inform the environment about the change:

proc ExM(trc) = Exception(trc).(Master + safe.SafeMaster)

In this case we assume for the *Workers* that once they fail they do not restart again. This constraint removes all possible non-progressing loops produced by continuous failures of the *Workers*. The property that we are going to prove is:

(B4): $[\ (\neg\ \text{"safe"})^*.\ \text{"safe"}\ ]\ \mu X.\ (\langle\ \mathsf{T}\rangle\ \mathsf{T} \wedge [\ \neg\ \text{"publish(n)"}\ ]\ X)$

The formula expresses that once the *Master* becomes safe then the *publish* action is “inevitably reached”. The property is satisfied by the constrained system. The above properties reason about the termination of the algorithm once the *Master* becomes safe but we have to be sure also that all participants may contribute to the progress of the computation. For this purpose, we check also the following property, expressing that both unreliable *Master* and *Workers* may perform additions, writing the result into the space:

(B5): $\langle\ \mathsf{T}^*\ .\ \text{'M\_wrote.*'}\ \rangle\ \mathsf{T} \wedge \langle\ \mathsf{T}^*\ .\ \text{'W\_wrote.*'}\ \rangle\ \mathsf{T}$

We have checked the properties on several instances of the system. With different number of *Workers* and *entries*. The following table gives the size of the systems on which we have done the experiments:

	# data entries			
# workers	2	3	4	5
1	16206	193179	9195321	–
2	56436	691056	36791821	–
3	121256	1520550	–	–

Figure 7.1: Sizes of the generated state spaces

7.4 Conclusion

Through the analysis of a non-trivial example we validated our general framework to verify JavaSpaces applications. The studied problem is a representative example of coordination problems that JavaSpaces is aimed at. The μCRL language and its related tool sets have been shown to be suitable to model check this class of applications. We conclude that small but non-trivial JavaSpaces applications can be effectively verified using our technology. This technology could still be scaled up, by using abstraction and symmetry techniques. In the next chapter, we provide general patterns to abstract JavaSpaces applications.

We have seen in section 7.2, how we manually translate the Java code into μCRL. We think that the automatic translation of the code is very important from a methodological point of view and for the “industrial” application of the verification technique, however this process is completely orthogonal to our research and can be carried out by applying techniques implemented in the Bandera [69] or Loop Project's [7] tool.

The proposed algorithm for parallel summation can be generalised in order to solve other similar coordination problems. Furthermore, we can see this generalisation as an alternative fault-tolerant distributed implementation of the so-called chemical abstract machines [10]. The JavaSpaces will store the chemical molecules and the *Workers* will perform in parallel the chemical reactions.

The Gamma paradigm [95] is also based on the computation of simple operations over multisets of data according to some specific rules. The present case study can be transformed in such a way that it supports other operations different from summation.

Chapter 8

Abstracting JavaSpaces

This short chapter concludes the second part of the thesis. So far, we have presented the formal specification of JavaSpaces and we have analysed a non-trivial application, where we have seen the difficulties to handle realistic instances of the case study due to the state explosion problem. We start by presenting a small case study that illustrates how to apply the abstraction framework presented in the first part of the thesis to a specific application. Then, some basic guidelines will be given about how to apply the results about abstraction to generic JavaSpaces applications.

8.1 A Case Study On Replication

Abstraction, as we have discussed in the first part of the thesis, is a suitable technique to attack the state explosion problem that arises during the automatic analysis of complex systems. In section 7.3 of Chapter 7, we saw how we can hardly verify instances of the summation algorithm with more than 3 workers and 5 entries, these figures are relatively small comparing to what can be considered as a realistic system. Even if the verification of small instances gives extra confidence about the correctness of the system, we cannot formally infer that it behaves correctly for any instance.

The target of the concluding chapter is not to deeply study any concrete case study but to give the guidelines on how and where we can use abstraction techniques to verify JavaSpaces applications. We want to identify the parts of common JavaSpaces applications that are suitable to be abstracted and to provide examples of abstract patterns that can be used. For this purpose, we first analyse a simple application and then, in the next section, we generalise the conclusions of the analysis. Let us consider a simple example composed by three different types of components:

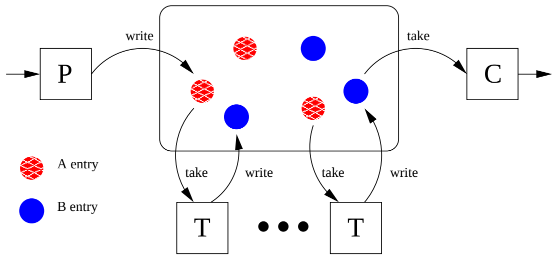


Figure 8.1: Producer-Transformer(s)-Consumer

- *Producer:* It writes new entries into a shared repository. We can think of the producer as an acquisition unit that generates a continuous flux of data that have to be processed. We mark the unprocessed entries as being of type A.
- *Transformer:* It retrieves entries of type A from the repository, performs some computation and writes back a transformed entry. The processed entry is of type B.
- *Consumer:* It takes the processed information and displays the result of the computations.

Unprocessed data becomes obsolete after some specific time. Therefore every A has an associated timeout. The repository will automatically remove the entries with expired timeouts. The complete system is defined by the parallel composition of the producer, the consumer, a process representing the repository and N *Transformers* process.

A real life example that can match our model is, for example, a radar-monitor system. The radar introduces packets of different measurements taken from an external moving agent. Transformer processes the measures by computing predictions of future moves of the investigated agent. The monitor displays the results of the process. In general, we would like to have several transformers making calculations at the same time in order to accelerate the display of the results. Figure 8.1 presents an overview of the system.

The implementation of such systems is straightforward using the JavaSpaces architecture. The μCRL code of the different components of the system, that are composed in parallel with the JavaSpaces process, is:

```
proc  Producer =
        write(A, NULL, timeout)
        .Producer


proc  Consumer =
        take(cC, typeB, NULL, FOREVER)
        .TakeReturn(cC, B)
        .Consumer


proc Transformer =
        take(cT, typeA, NULL, FOREVER)
        .TakeReturn(cT, A)
        .write(B, NULL, timeout)
        .Transformer
```

We considered that *Transformer* shared the same communication channel cT with the space. If two transformers are in the same state, waiting for an entry, they would compit for the returned entry. Actually we do not care about this fact because it is not important which of the *Transformers* performes the computation.

One basic requirement to check is that the system, no matter what happens, keeps progressing. In other words that it will not deadlock. If the capacity of the repository is bounded and the A entries are never eliminated (infinite timeout), the system may arrive to a deadlock state. Let us consider a finite instance of the system with one single *Transformer* and the size of the repository equals 2. The following sequence of steps: 1) Producer writes A, 2) Producer writes A, 3) transformer takes A, 4) Producer writes A, leads to a deadlock since the repository is full and the transformer cannot write a B entry, the producer cannot write any new entry either and the consumer cannot retrieve any B entry to free some space in the repository. If the repository is unbounded this problem will not arise since both producer and transformer can always write. But this solution is not realistic because we cannot assume to have infinite memory. The idea is to use the entries' timeout to allow the repository to free some place for new incoming entries. In principle, by using timeouts, there should not be any deadlock in the system, because in any state one of the following actions is possible:

- If it is not full then the producer can *Write*.
- If there is some A entry in the repository, *Transformers* that are waiting for an entry can *take* it (similar for B).
- If there is one expired entry, i.e., its timeout equals 0, and no process is requiring it, then the repository can remove the entry.
- If there is one entry with non-expired timeout, the repository can decrease the timeout of the entry.

Transformer's codes are *identical* (as they all used the same channel identifier) therefore we can use the pattern presented in Section 4.4. Moreover, we can abstract it by doing an abstraction to the counters of the number of processes that are in one state, as presented in pattern of Section 4.3.2. We can not abstract all state counters because it is important to capture the idea that every read request performed by a *Transformer* should be followed by exactly one answer from the repository. Therefore, we abstract all state counters but the one that determines the number of processes that are waiting for an entry.

The absence of deadlock may be expressed using the action-based μ-calculus with modalities as follows:

(P1): ν X. ($\langle$ '. $*$must .$*$' $\rangle$ T$\wedge$ ['. $*$may .$*$'] X)

The formula states that from every state that *may* be reached there is an out-going *must* transition. Below, we present a table with the sizes of the state spaces for different instances of the system. We compare three cases:

- The concrete system represented with the standard representation of parallel processes (denoted by Crt).
- The concrete system given in the linear form proposed in the equation of Theorem 4.4.1 (denoted by Crt Lin).
- The abstract version of the last case (denoted by Abs).

The table shows how the standard representation of parallel processes can not deal with big instances of the system. However our proposed format can, because it eliminates symmetries of the interleavings. Moreover, with the abstraction we can reduce even more the size of the systems. It is possible to handle instances with more than 100 parallel processes. In order to generalise the model checking problem to an arbitrary number of processes, we would have to abstract also some other parts of the system.

Crt	States	Crt Lin	States	Abs	States
5T	15,135	10T	4,663	10T	3,858
6T	49,560	20T	25,828	20T	12,093
7T	161,097	40T	267,302	40T	42,171
8T	520,494	80T	1,193,830	80T	156,759
				100T	241,269

This example shows that the proposed framework is suitable for verifying *liveness* properties.

8.2 Abstraction Guidelines

The simple example presented in the previous section allow us to point out the general *abstractable* parts of the JavaSpaces applications. Let us consider the different parts of the system that can be abstracted:

- **Entries:** The entries that are manipulated by the system are only of sorts A or B. Note that this already an abstraction of a realistic case in which the entries can contain an arbitrary value. They can be for example measurements of an external radar that transformers have to manipulate to produce digested values. When the correctness of the application does not depend on the real values of the entries (which is the case for the example) we can abstract away these values.

  In the summation algorithm of the previous chapter, we were interested in the publication of the correct value of the addition. If we abstract the value of the entries then we can relax the property. We might prove that something is published (independently of the value of the result). Furthermore, we can check that every path leading to a publication contains the desirable number of partial publications.

  Note that, one can also abstract queries according to the abstraction of the entries.

- **Entry Set:** Apart from the contents of the entries, we can abstract the repository to a counter describing the number of entries inside the repository. Furthermore, we may only consider some interesting symbolic values such as *empty*, *more* and $full$ as we have done for the buffer example of Chapter 1.

  If we are interested in proving termination, for example described as *at some point all entries are removed from the space*, we would have problems using some symbolic abstraction. To solve this problem we can use *accelerations*. We know that by performing the action *take* (more precisely *takeReturn*) the number of entries in the set decreases by one. Therefore, we cannot have a path containing an infinite number of *takeReturns*, unless it contains an infinite number of *writes*.

- **Timeouts and Leases:** They play a very important role in the process of implementing applications. Considering that the number of resources of the space is limited, timeouts and leases may be used to control the usage of them. We have that, in the example, the lease of the entries was necessary to guarantee the absence of deadlocks in the system.

  They are implemented by using decreasing counters. The space grants a value, and then it decreases the counter until zero where some action is performed, such as: exceptions, garbage collection or null returns. Trivially, counters can be abstracted by considering the needed symbolic values. Normally, we would have to distinguish whether the timeouts are expired or not. In some cases, we may need more values to capture the relations among different counters (for example in the case of the summation algorithm).

  As for the previous case, *accelerations* will help to verify expirations of timeouts. For leases, the pattern will coincide exactly with the resettable counter presented in Chapter 5, because leases can be renewed at any case.

- **Replicated Processes:** The example itself was used to show the application of the patterns to replicated processes. In fact, many of the typical JavaSpaces applications have a similar form. We have different kinds of processes interacting among each other through the space, some of the processes are identical. For example, in the summation case study we can replicate the *worker* process.

  For replicated processes we can abstract away from channel identifiers, as we have done for the Producer-Transformer(s)-Consumer. We can consider that replicated processes share the same communication channel. This fact allows the use of the pattern for *identical* processes presented in Section 4.4, because external processes are not identifiable. We show below the linear process that captures an arbitrary number of transformers. We consider that the process can be in three different states:

  1. *Taking.* Just before performing the *take* action.
  2. *Waiting.* for the return of a *take*.
  3. *Writing.* Inserting the new entry in the space.

```
proc  Transformers(dt: DTable) =
        $\sum_{d:\ D}$(
          take(cT, typeA, NULL, FOREVER)
          .Transformers(update(waiting, d, dt))
          $\lhd$ test(d, dt) $\wedge$ d = taking $\rhd \delta$ +
          TakeReturn(cT, A)
          .Transformers(update(writing, d, dt))
```

$$
\begin{aligned}
&\lhd \ \mathsf{test(d,\ dt)} \ \wedge \ \mathsf{d = waiting} \rhd \delta \ + \\
&\mathsf{write(B,\ NULL,\ timeout)} \\
&\mathsf{.Transformers(update(taking,\ d,\ dt))} \\
&\lhd \ \mathsf{test(d,\ dt)} \ \wedge \ \mathsf{d = writing} \rhd \delta)
\end{aligned}
$$

If we use symbolic counters to represent the number of processes that are in a given state, we will not be able to count down to *zero*. In that case, we can again use *accelerations* to describe the fact that at some point the number of processes in a state becomes *zero*.

- **Other parts:** The abstraction of the rest of the structures of the space is more complicated such as: *pending actions* or *transactions*. They play a crucial role in the well behaving of the system. For example, if we abstract away the contents of the pending actions the space will not know what he has to return and to whom.

8.3 Conclusion

This entire chapter is a conclusion for the second part of the thesis, it describes how to apply the verification framework by abstraction to JavaSpaces applications. The purpose of the chapter was not to apply the abstraction patterns to an illustrative example, but to give some guidelines about the general application of them.

As an exercise, it would be interesting to use the patterns to push forward the verification of the summation example. Some of the abstractions can be easily applied because they are fully automated by the abstraction toolset for μCRL presented in Chapter 3. However, other abstractions need extra effort to implement tools to support them, for example:

- To use the pattern for replicated processes, we first need to transform the processes to a special linear form. This transformation can be done manually for simple examples, but it is tedious for more complex systems.

- The inclusion of *accelerations*, so far, is a little bit more complicated. In Chapter 5, we have described the theory at the semantic level. It is still to be investigate how to add the accelerations directly to the specification. In any case for simple examples, we can use ad how reasoning, as we have done in section 5.6.1.

We have applied several abstraction techniques to JavaSpaces applications by prototype tools and sometimes by hand. This shows that the techniques are promising. To apply this routinely on a larger scale, some more implementation and integration would be needed.

Appendix A

JavaSpaces

This appendix includes the full JavaSpaces specification together with the source code of the summation algorithm. It is organized with the following order:

1. The different data structures that the space uses.
2. The definition of actions and communications.
3. The process *javaspace*.
4. The auxiliary constants and processes.
5. The summation system, presented in section 7.3.

A.1 Data Description

We start by describing all the sorts one by one, and then we provide their μCRL code:

- **Bool**: Represents the booleans. It contains two constructors *true* and *false*, and the basic operators for negation, conjunction, disjunction,...
- **Nat**: Represents the natural numbers. It contains two constructors *0* and successor, and the basic operators for addition, comparision, ...
- **Time**: Represents the time that is used by processes to request leases or timeouts. It contains two constructors *FOREVER* and *tt*. The first one represents the maximum, no other time value is bigger than *FOREVER*. By using *tt* we encapsulate a natural numbers. For example, $tt(1)$ represents the time value 1.

  The operations are basically the same as for the natural numbers. The differences are when applying the operators to the constant *FOREVER*. For instance, the result of decrementing *FOREVER* is *FOREVER*.

- **TypeAction**: Represents the different types of look-up operations, used to distinguish among the pending actions, see section 6.3.1. There are four different constants:
  - *takeA*, It denotes a take action.
  - *readA*. It denotes a read action.
  - *takeEA*. It denotes a takeIfExists action.
  - *readEA*. It denotes a readIfExists action.

The μCRL code of these four sorts is included in section A.2. Now, we continue with some other core data structures:

- **Object**: It is used to encapsulate Entries, see section 6.3. The constructor *object* has three arguments:
  - The object identifier, represented by a natural number. Identifiers are used to implement the blocking mechanism of transactions.
  - The entry. We recall that the sort *Entry* should be provided by the external application, in section A.8, we see an example of it.
  - The lease. It is the reference to the lease.

  The sort provides functions to access to the different components of the object, to decrement the lease and to check if the object has expired.

- **Action**: It is used to encapsulate the pending look-up operations, see section 6.3.1. The constructor *action* has the following arguments:
  - The type of action: *readA*, *takeA*, ...
  - The reference to the transaction to which it is associated. This field is a natural number.
  - The timeout of sort *Time*.
  - The channel identifier, to which the answer has to be sent. It is a natural number provided by the external process.
  - The query of sort *Query*. We recall that the query is used to implement the matching, see section 6.2. It has to be provided by the external application. We can see an example in section A.8.

  The sort provides functions to access to the different fields of the pending action, to decrement the timeout and to check if the action has expired.

- **Entry**: It is used to model Entries. It has to be provided by the external user. All entries have to contain a null constructor *entryNull*, used to implement the unsuccessful return of the look-up operations.

- **Query**: It is used to model the matching mechanism. It has to be provided by the external user. All sorts have to contain the *match* function.

The μCRL code of the first two sorts can be found in section A.2. An example of sorts *Entry* and *Query* can be found in section A.8

- **Transaction**: It models the transactions, see section 6.3.2. The constructor has the following arguments:
  - The transaction identifier, which is a natural number.
  - The reference to the lease.
  - The three different object sets used to implement the transaction mechanism, as we have seen in section 6.3.2.

  Apart from the accessors to the different fields of the transaction, to decrement the lease and to check if it has expired, the sort has some functions to add and remove objects to and from the objects sets.

- **TransactionCreated**: It is used as return value after the creation of a transaction. The constructor contains two fields: the transaction identifier and the lease identifier.

- **Lease**: Represents the lease. It contains two fields: the identifier which is a natural number, and the timeout. It provides functions to decrement the timeout, to check if it is expired and to check if it has a timeout different than *FOREVER*.

- **Registration**: It models the event registrations, as we have seen in section 6.3.3. The constructors contains the following fields:
  - The registration identifier.
  - The reference to the transaction to which it is associated.
  - The reference to the lease.
  - To query used to match incoming entries.
  - The sequence number of matching entries that have arrived.
  - A boolean (*notified*) denoting if there is some pending event to be sent.

  The sort provides functions to access to the different fields, to decrement the lease and to check if it has expired. Furthermore, it has a function *match* to increment the sequence number if an entry matches the query. And two functions to check and change the status of the registration (from *not notified* to *notified* and the other way around).

- **EventRegistration**: It is used as return value after the registration for incoming entries. The constructor contains two fields: the registration identifier and the lease identifier.

- **Event**: It is used to represents the notification events sent by the space to the listeners. Every event contains the registration identifier to which it is directed and the sequence number (number of matching entries that have arrived since the last notification).

All these sorts are presented in section A.2. Now we proceed explaining the data bases of the system.

- **ObjectSet**. It represents a set of objects. It has two constructors: the empty set, *emO* and the insertion function. It also provides the following operations:
  - Basic functions to add, remove and get entries from the set. And to make the union of two sets.
  - *areExpired*. It is a function to check if there are expired entries. There is also a function to decrement all active leases.
  - *areTimeouts*. It is a function to check if there are active leases (with value different than *FOREVER* and bigger than 0) in the set.
  - *matches*. It is a function to check if there are some objects matching a the query of a pending action.
  - *freshId*. It is a function to look for fresh identifiers. Identifiers are from a given range (0...*maxObjects*). When a new object is created a unused identifier is assigned to it. The function *isUsed* is used to check if an identifier is already assigned to an object.
  - *leases*. Gives the set of leases associated to the objects of the set.
- **ActionSet**. It is used to model the pending action set. The specification is very similar to the previous one. Apart from the basic functions, there is a function *match* to check if an object matches some action's query.
- **TransactionSet**. It is used to model the transaction set. Its specification is also very similar to the *ObjectSet*'s one. Apart from the previously cited functions, it provides:
  - *URsets*. It is function to make the union of all read sets of the active transactions. The same functions exists for the write and take sets. This functions are used to implement the blocking mechanism.
  - *readLocked*. Checks if an entry is locked by some active transaction.
- **LeaseSet**. It is simple set to store the leases. It just has the basic constructors, and the functions to look for fresh ids.
- **RegistrationSet**. It is set to store registrations. Its specification is also very similar to the *ObjectSet*'s one.
- **EventList**. It is simple list to store events. It is used by the implementation of the network between the space and the listeners, see section 6.3.3. It provides the basic functions to manipulate lists.

A.2 Basic Data Types

```
sort Bool
func T,F:->Bool
map  eq:Bool#Bool->Bool
     not:Bool->Bool
     and,or:Bool#Bool->Bool
     if:Bool#Bool#Bool->Bool
var  x,y:Bool
rew  eq(T,T)=T         eq(F,F)=T
     eq(T,F)=F         eq(F,T)=F
     and(T,x) = x      and(F,x) = F
     and(x,F) = F      and(x,T) = x
     or(T,x) = T       or(F,x) = x
     or(x, T) = T      or(x, F) = x
     not(T) = F        not(F) = T
     if(T,x,y) = x     if(F,x,y) = y
     not(not(x)) = x

sort Nat
func 0:->Nat
     S:Nat->Nat
map  eq:Nat#Nat->Bool
     plus:Nat#Nat->Nat
     P:Nat->Nat
     lt:Nat#Nat->Bool
     gt:Nat#Nat->Bool
     ge:Nat#Nat->Bool
     le:Nat#Nat->Bool
     if:Bool#Nat#Nat->Nat
var  x,y:Nat
rew  eq(x,x) = T                  eq(0,S(x)) = F
     eq(S(x),0) = F               eq(S(x),S(y)) = eq(x,y)
     eq(x,S(x)) = F               eq(S(x),x) = F
     plus(x,0) = x                plus(x,S(y)) = S(plus(x,y))
     ge(x,0) = T                  ge(0,S(x)) = F
     ge(S(x),S(y)) = ge(x,y)      gt(x,y) = ge(x,S(y))
     lt(x,y) = gt(y,x)            le(x,y) = ge(y,x)
     P(0) = 0                     P(S(x)) = x
     if(T,x,y) = x                if(F,x,y) = y

% Some frequently used constants

map  1,2,3,4,5,6:-> Nat
rew  1 = S(0)
     2 = S(S(0))
     3 = S(S(S(0)))
     4 = S(S(S(S(0))))
     5 = S(S(S(S(S(0)))))
     6 = S(S(S(S(S(S(0))))))
```

```
% Time: natural, use to specify timeouts, leases, ...

sort Time
func tt:Nat->Time
     FOREVER:->Time
map  eq:Time#Time->Bool
     plus:Time#Time->Time
     lt:Time#Time->Bool
     gt:Time#Time->Bool
     ge:Time#Time->Bool
     le:Time#Time->Bool
     S:Time->Time
     decT:Time->Time
var  x, x':Nat
     t1, t2: Time
rew  eq(FOREVER, FOREVER) = T
     eq(tt(x), FOREVER) = F
     eq(tt(x), tt(x')) = eq(x,x')
     eq(FOREVER, tt(x)) = F
     plus(FOREVER, FOREVER) = FOREVER
     plus(tt(x), FOREVER) = FOREVER
     plus(tt(x), tt(x')) = tt(plus(x,x'))
     plus(FOREVER, tt(x)) = FOREVER
     lt(FOREVER, FOREVER) = F
     lt(tt(x), FOREVER) = T
     lt(tt(x), tt(x')) = lt(x,x')
     lt(FOREVER, tt(x)) = F
     le(FOREVER, FOREVER) = T
     le(tt(x), FOREVER) = T
     le(tt(x), tt(x')) = le(x,x')
     le(FOREVER, tt(x)) = F
     gt(FOREVER, FOREVER) = F
     gt(tt(x), FOREVER) = F
     gt(tt(x), tt(x')) = gt(x,x')
     gt(FOREVER, tt(x)) = T
     ge(FOREVER, FOREVER) = T
     ge(tt(x), FOREVER) = F
     ge(tt(x), tt(x')) = ge(x,x')
     ge(FOREVER, tt(x)) = T
     S(FOREVER) = FOREVER
     S(tt(x)) = tt(S(x))
     decT(tt(x)) = tt(P(x))
     decT(FOREVER) = FOREVER

% TypeAction:

sort TypeAction
func takeA, readA, takeEA, readEA :->TypeAction
map  eq:TypeAction#TypeAction->Bool
```

```
var  ta1, ta2: TypeAction
rew  eq(takeA, takeA) = T       eq(readA, readA) = T
     eq(takeEA, takeEA) = T     eq(readEA, readEA) = T
     eq(takeA, readA) = F       eq(takeA, readEA) = F
     eq(takeA, takeEA) = F      eq(readA, takeA) = F
     eq(readA, readEA) = F      eq(readA, takeEA) = F
     eq(takeEA, readA) = F      eq(takeEA, readEA) = F
     eq(takeEA, takeA) = F      eq(readEA, readA) = F
     eq(readEA, takeA) = F      eq(readEA, takeEA) = F
```

A.3 System Data Structures

```
% Object:

sort Object
func object:Nat#Entry#Lease->Object
map  eq:Object#Object->Bool
     entry:Object->Entry
     lease:Object->Lease
     id:Object->Nat
     decT:Object->Object
     isExpired:Object->Bool
var  o,o’:Object
     e:Entry
     l:Lease
     id:Nat
rew  eq(o, o’) = and(and(
        eq(lease(o), lease(o’)),
        eq(id(o), id(o’))),
        eq(entry(o), entry(o’)))
     entry(object(id,e,l)) = e
     lease(object(id,e,l)) = l
     id(object(id,e,l)) = id
     decT(object(id,e,l)) = object(id,e,decT(l))
     isExpired(o) = isExpired(lease(o))

% Action:

sort Action
func action: TypeAction#Nat#Nat#Time#Query->Action
map  eq:Action#Action->Bool
     type:Action->TypeAction
     trcId:Action->Nat
     timeout:Action->Time
     procId:Action->Nat
     query:Action->Query
     decT:Action->Action
     isExpired:Action->Bool
var  a,a’:Action
```

```
     trc:Nat
     to:Time
     type:TypeAction
     q:Query
     pId:Nat
rew  eq(a,a’) = and(and(and(and(
        eq(type(a), type(a’)),
        eq(timeout(a), timeout(a’))),
        eq(procId(a), procId(a’))),
        eq(trcId(a), trcId(a’))),
        eq(query(a), query(a’)))
     type(action(type,trc,pId,to,q)) = type
     trcId(action(type,trc,pId,to,q)) = trc
     timeout(action(type,trc,pId,to,q)) = to
     procId(action(type,trc,pId,to,q)) = pId
     query(action(type,trc,pId,to,q)) = q
     decT(action(type,trc,pId,to,q)) =
        action(type,trc,pId,decT(to),q)
     isExpired(a) = eq(timeout(a), tt(0))

% Transaction:

sort Transaction
func transaction:Nat#Lease#
        ObjectSet#ObjectSet#ObjectSet->Transaction
map  eq:Transaction#Transaction->Bool
     id:Transaction->Nat
     lease:Transaction->Lease
     Wset:Transaction->ObjectSet
     Tset:Transaction->ObjectSet
     Rset:Transaction->ObjectSet
     decT:Transaction->Transaction
     isExpired:Transaction->Bool
     inW:Object#Transaction->Transaction
     inT:Object#Transaction->Transaction
     inR:Object#Transaction->Transaction
     remW:Object#Transaction->Transaction
     remT:Object#Transaction->Transaction
     remR:Object#Transaction->Transaction
     if:Bool#Transaction#Transaction->Transaction
var  t,t’:Transaction
     id:Nat
     l:Lease
     Ws,Ts,Rs:ObjectSet
     o:Object
rew  eq(t, t’) = and(and(and(and(eq(id(t), id(t’)),
        eq(lease(t), lease(t’))), eq(Wset(t),Wset(t’))),
        eq(Tset(t),Tset(t’))), eq(Rset(t),Rset(t’)))
     id(transaction(id, l, Ws, Ts, Rs)) = id
     lease(transaction(id, l, Ws, Ts, Rs)) = l
     Wset(transaction(id, l, Ws, Ts, Rs)) = Ws
     Tset(transaction(id, l, Ws, Ts, Rs)) = Ts
     Rset(transaction(id, l, Ws, Ts, Rs)) = Rs
     decT(transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, decT(l), Ws, Ts, Rs)
     isExpired(t) = isExpired(lease(t))
     inW(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, in(o, Ws), Ts, Rs)
     inT(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, Ws, in(o, Ts), Rs)
     inR(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, Ws, Ts, in(o, Rs))
     remW(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, rem(o, Ws), Ts, Rs)
     remT(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, Ws, rem(o, Ts), Rs)
     remR(o,transaction(id, l, Ws, Ts, Rs)) =
        transaction(id, l, Ws, Ts, rem(o, Rs))
     if(T,t,t’) = t
          if(F,t,t’) = t’

% TransactionCreated:
%    return data after the creation of a transaction

sort TransactionCreated
func trcCreated:Nat#Nat->TransactionCreated
map  eq:TransactionCreated#TransactionCreated->Bool
     trcId:TransactionCreated->Nat
     leaseId:TransactionCreated->Nat
var  t,t’:TransactionCreated
     id:Nat
     l:Nat
rew  eq(t, t’) = and(eq(trcId(t), trcId(t’)),
        eq(leaseId(t), leaseId(t’)))
     trcId(trcCreated(id, l)) = id
     leaseId(trcCreated(id, l)) = l

% Lease:

sort Lease
func lease:Nat#Time->Lease
map  eq:Lease#Lease->Bool
     id:Lease->Nat
     timeout:Lease->Time
     decT:Lease->Lease
     isExpired:Lease->Bool
     hasTimeout:Lease->Bool
var  l,l’:Lease
     to:Time
     id:Nat
```

```
rew  eq(l, l’) = and(
        eq(id(l), id(l’)),
        eq(timeout(l), timeout(l’)))
     id(lease(id,to)) = id
     timeout(lease(id,to)) = to
     decT(lease(id,to)) = lease(id,decT(to))
     isExpired(l) = eq(timeout(l),tt(0))
     hasTimeout(l) = not(eq(timeout(l), FOREVER))

% Registration

sort Registration
func registration: Nat#Nat#Lease#
         Query#Nat#Bool -> Registration
map  eq: Registration#Registration -> Bool
     id:Registration -> Nat
     trcId:Registration -> Nat
     query:Registration->Query
     lease:Registration -> Lease
     notified:Registration -> Bool
     seq:Registration -> Nat
     isExpired:Registration->Bool
     decT:Registration->Registration
     match:Entry#Nat#Registration->Registration
     notify:Registration->Registration
     if:Bool#Registration#Registration->Registration
var  no, no’:Registration
     id:Nat
     b:Bool
     q:Query
     e:Entry
     l:Lease
     s,trc,trcId:Nat
rew  eq(no, no’) = and(and(and(and(and(
        eq(id(no), id(no’)),
        eq(trcId(no), trcId(no’))),
        eq(lease(no), lease(no’))),
        eq(query(no), query(no’))),
        eq(notified(no), notified(no’))),
        eq(seq(no), seq(no’)))
     if(T,no,no’) = no
     if(F,no,no’) = no’
     id(registration(id,trc,l,q,s,b)) = id
     trcId(registration(id,trc,l,q,s,b)) = trc
     lease(registration(id,trc,l,q,s,b)) = l
     query(registration(id,trc,l,q,s,b)) = q
     seq(registration(id,trc,l,q,s,b)) = s
     notified(registration(id,trc,l,q,s,b))= b
     isExpired(no) = isExpired(lease(no))
     decT(registration(id,trc,l,q,s,b)) =
         registration(id,trc,decT(l),q,s,b)
      match(e,trcId,registration(id,trc,l,q,s,b)) =
         if(and(match(q, e),
               or(eq(trcId,trc), eq(trcId,NULL))),
            registration(id,trc,l,q,S(s),F),
            registration(id,trc,l,q,s,b))
      notify(registration(id,trc,l,q,s,b)) =
         registration(id,trc,l,q,s,T)

% EventRegistration:
%    return data after the registration for events

sort EventRegistration
func evReg:Nat#Nat->EventRegistration
map  eq:EventRegistration#EventRegistration->Bool
     regId:EventRegistration->Nat
     leaseId:EventRegistration->Nat
var  e,e':EventRegistration
     id:Nat
     l:Nat
rew  eq(e, e') = and(eq(regId(e), regId(e')),
        eq(leaseId(e), leaseId(e')))
     regId(evReg(id, l)) = id
     leaseId(evReg(id, l)) = l

% Event

sort Event
func event: Nat#Nat -> Event
map  eq: Event#Event -> Bool
     registrationId:Event -> Nat
     seq:Event -> Nat
var  ev, ev':Event
     id, s:Nat
rew  eq(ev, ev') = and(
        eq(seq(ev), seq(ev')),
        eq(registrationId(ev), registrationId(ev')))
     registrationId(event(id,s)) = id
     seq(event(id,s))= s
```

A.4 System Data Bases

```
% Object Set:

sort ObjectSet
func emO:->ObjectSet
     in:Object#ObjectSet->ObjectSet
map  eq:ObjectSet#ObjectSet->Bool
     if:Bool#ObjectSet#ObjectSet->ObjectSet
     rem:Object#ObjectSet->ObjectSet
     get:ObjectSet#Object->Bool
     U:ObjectSet#ObjectSet->ObjectSet
     areExpired:ObjectSet->Bool
     areTimeouts:ObjectSet->Bool
     matches:Query#ObjectSet->Bool
     decT:ObjectSet->ObjectSet
     freshId:ObjectSet->Nat
     freshId:ObjectSet#Nat->Nat
     isUsed:ObjectSet#Nat->Bool
     leases:ObjectSet->LeaseSet
     leases:ObjectSet#LeaseSet->LeaseSet
var  S,S’:ObjectSet
     o,o’:Object
     time:Time
     query:Query
     n:Nat
     L:LeaseSet
rew  eq(emO,emO) = T
     eq(emO,in(o,S)) = F
     eq(in(o,S),emO) = F
     eq(in(o,S),in(o’,S’)) = and(eq(o,o’),eq(S,S’))
     if(T,S,S’) = S
     if(F,S,S’) = S’
     rem(o,emO) = emO
     rem(o,in(o’,S)) = if(eq(o,o’),S,in(o’,rem(o,S)))
     get(emO,o) = F
     get(in(o,S),o’) = or(eq(o,o’),get(S,o’))
     U(S,emO)=S
     U(emO,S)=S
     U(S,in(o,S’))=U(in(o,S),S’)
     areExpired(emO)=F
     areExpired(in(o,S)) = or(isExpired(o),areExpired(S))
     areTimeouts(emO) = F
     areTimeouts(in(o, S)) =
       if(hasTimeout(lease(o)), T, areTimeouts(S))
     matches(query, emO) = F
     matches(query, in(o, S)) =
        if(match(query, entry(o)), T, matches(query, S))
     decT(emO) = emO
     decT(in(o, S)) = in(decT(o), decT(S))
     freshId(S) = freshId(S, P(maxObjects))
     freshId(S, 0) = if(isUsed(S, 0), maxObjects, 0)
     freshId(S, S(n)) =
        if(isUsed(S, S(n)), freshId(S, n), S(n))
     isUsed(emO, n) = F
     isUsed(in(o, S), n) = if(eq(id(o), n), T, isUsed(S,n))
     leases(S) = leases(S, emL)
     leases(emO, L) = L
     leases(in(o, S), L) = leases(S, in(lease(o), L))
```

```
% Action Set:

sort ActionSet
func emA:->ActionSet
     in:Action#ActionSet->ActionSet
map  eq:ActionSet#ActionSet->Bool
     if:Bool#ActionSet#ActionSet->ActionSet
     rem:Action#ActionSet->ActionSet
     remPAs:Nat#ActionSet->ActionSet
     U:ActionSet#ActionSet->ActionSet
     areExpired:ActionSet->Bool
     areTimeouts:ActionSet->Bool
     get:ActionSet#Action->Bool
     decT:ActionSet->ActionSet
     match:Object#ActionSet->Bool
     len:ActionSet->Nat
var  S,S’:ActionSet
     a,a’:Action
     time:Time
     o:Object
     id:Nat
rew  eq(emA,emA) = T
     eq(emA,in(a,S)) = F
     eq(in(a,S),emA) = F
     eq(in(a,S),in(a’,S’)) = and(eq(a,a’),eq(S,S’))
     if(T,S,S’) = S
     if(F,S,S’) = S’
     rem(a,emA) = emA
     rem(a,in(a’,S)) = if(eq(a,a’),S,in(a’,rem(a,S)))
     U(S,emA)=S
     U(emA,S)=S
     U(S,in(a,S’))=U(in(a,S),S’)
     areExpired(emA)=F
     areExpired(in(a,S)) = or(isExpired(a), areExpired(S))
     areTimeouts(emA) = F
     areTimeouts(in(a, S)) =
        if(not(eq(timeout(a), FOREVER)), T, areTimeouts(S))
     get(emA,a) = F
     get(in(a,S),a’) = or(eq(a,a’),get(S,a’))
     decT(emA) = emA
     decT(in(a, S)) = in(decT(a), decT(S))
     match(o, emA) = F
     match(o, in(a, S)) =
        if(match(query(a), entry(o)), T, match(o, S))
     len(emA) = 0
     len(in(a, S)) = S(len(S))
     remPAs(id,emA) = emA
     remPAs(id,in(a,S)) =
        if(eq(id,trcId(a)),
           remPAs(id, S), in(a,remPAs(id, S)))

 % Transaction Set

sort TransactionSet
func emT:->TransactionSet
     in:Transaction#TransactionSet->TransactionSet
map  eq:TransactionSet#TransactionSet->Bool
     if:Bool#TransactionSet#TransactionSet->TransactionSet
     rem:Transaction#TransactionSet->TransactionSet
     get:TransactionSet#Transaction->Bool
     getById:TransactionSet#Nat->Transaction
     areExpired:TransactionSet->Bool
     areTimeouts:TransactionSet->Bool
     decT:TransactionSet->TransactionSet
     freshId:TransactionSet->Nat
     freshId:TransactionSet#Nat->Nat
     isUsed:TransactionSet#Nat->Bool
     URsets:TransactionSet->ObjectSet
     UWsets:TransactionSet->ObjectSet
     UTsets:TransactionSet->ObjectSet
     readLocked:Object#TransactionSet->Bool
     leases:TransactionSet->LeaseSet
     leases:TransactionSet#LeaseSet->LeaseSet
var  S,S’:TransactionSet
     t,t’:Transaction
     time:Time
     n:Nat
     o:Object
     L:LeaseSet
rew  if(T,S,S’) = S
     if(F,S,S’) = S’
     rem(t,emT) = emT
     rem(t,in(t’,S)) = if(eq(t,t’),S,in(t’,rem(t,S)))
     get(emT,t) = F
     get(in(t,S),t’) = or(eq(t,t’),get(S,t’))
     getById(in(t,S),n) = if(eq(id(t),n), t, getById(S, n))
     eq(emT,emT) = T
     eq(emT,in(t,S)) = F
     eq(in(t,S),emT) = F
     eq(in(t,S),in(t’,S’)) = and(eq(t,t’),eq(S,S’))
     areTimeouts(emT) = F
     areTimeouts(in(t, S)) =
        if(hasTimeout(lease(t)), T, areTimeouts(S))
     areExpired(emT)=F
     areExpired(in(t,S)) = or(isExpired(t), areExpired(S))
     decT(emT) = emT
     decT(in(t, S)) = in(decT(t), decT(S))
     freshId(S) = freshId(S, P(maxTrc))
     freshId(S, 0) = maxTrc % NULL is reserved
     freshId(S, S(n)) =
        if(isUsed(S, S(n)), freshId(S, n), S(n))
     isUsed(emT, n) = F
     isUsed(in(t, S), n) =
        if(eq(id(t), n), T, isUsed(S,n))
     URsets(emT)=emO
     URsets(in(t,S))=U(Rset(t),URsets(S))
     UWsets(emT)=emO
     UWsets(in(t,S))=U(Wset(t),UWsets(S))
     UTsets(emT)=emO
     UTsets(in(t,S))=U(Tset(t),UTsets(S))
     readLocked(o, S) = isUsed(URsets(S), id(o))
     leases(S) = leases(S, emL)
     leases(emT, L) = L
     leases(in(t, S), L) = leases(S, in(lease(t), L))

% Lease Set

sort LeaseSet
func emL:->LeaseSet
     in:Lease#LeaseSet->LeaseSet
map  U:LeaseSet#LeaseSet->LeaseSet
     freshId:LeaseSet->Nat
     freshId:LeaseSet#Nat->Nat
     isUsed:LeaseSet#Nat->Bool
var  L, L':LeaseSet
     l: Lease
     n: Nat
rew  U(L,emL)=L
     U(emL,L)=L
     U(L,in(l,L'))=U(in(l,L),L')
     freshId(L) = freshId(L, P(maxLeases))
     freshId(L, 0) = if(isUsed(L, 0), maxLeases, 0)
     freshId(L, S(n)) =
        if(isUsed(L, S(n)), freshId(L, n), S(n))
     isUsed(emL, n) = F
     isUsed(in(l, L), n) =
        if(eq(id(l), n), T, isUsed(L,n))

% Registration Set:

sort RegistrationSet
func emR:->RegistrationSet
     in:Registration#RegistrationSet->RegistrationSet
map  eq:RegistrationSet#RegistrationSet->Bool
     if:Bool#RegistrationSet#
        RegistrationSet->RegistrationSet
     rem:Registration#RegistrationSet->RegistrationSet
     U:RegistrationSet#RegistrationSet->RegistrationSet
     areExpired:RegistrationSet->Bool
     areTimeouts:RegistrationSet->Bool
     get:RegistrationSet#Registration->Bool
     decT:RegistrationSet->RegistrationSet
     len:RegistrationSet->Nat
     freshId:RegistrationSet->Nat
     freshId:RegistrationSet#Nat->Nat
     isUsed:RegistrationSet#Nat->Bool
     match:Entry#Nat#RegistrationSet->RegistrationSet
     remRegs:Nat#RegistrationSet->RegistrationSet
     leases:RegistrationSet->LeaseSet
     leases:RegistrationSet#LeaseSet->LeaseSet
var  S,S’:RegistrationSet
     r,r’:Registration
     time:Time
     n:Nat
     e:Entry
     trcId:Nat
     L:LeaseSet
rew
     eq(emR,emR) = T
     eq(emR,in(r,S)) = F
     eq(in(r,S),emR) = F
     eq(in(r,S),in(r’,S’)) = and(eq(r,r’),eq(S,S’))
     if(T,S,S’) = S
     if(F,S,S’) = S’
     rem(r,emR) = emR
     rem(r,in(r’,S)) = if(eq(r,r’),S,in(r’,rem(r,S)))
     U(S,emR)=S
     U(emR,S)=S
     U(S,in(r,S’))=U(in(r,S),S’)
     areExpired(emR)=F
     areExpired(in(r,S)) = or(isExpired(r), areExpired(S))
     areTimeouts(emR) = F
     areTimeouts(in(r, S)) =
        if(hasTimeout(lease(r)), T, areTimeouts(S))
     get(emR,r) = F
     get(in(r,S),r’) = or(eq(r,r’),get(S,r’))
     decT(emR) = emR
     decT(in(r, S)) = in(decT(r), decT(S))
     len(emR) = 0
     len(in(r, S)) = S(len(S))
     freshId(S) = freshId(S, P(maxRegistrations))
     freshId(S, 0) = if(isUsed(S, 0), maxRegistrations, 0)
     freshId(S, S(n)) =
        if(isUsed(S, S(n)), freshId(S, n), S(n))
     isUsed(emR, n) = F
     isUsed(in(r, S), n) =
        if(eq(id(r), n), T, isUsed(S,n))
     match(e,trcId,emR) = emR
     match(e,trcId,in(r,S)) =
        in(match(e,trcId,r), match(e,trcId,S))
     remRegs(trcId,emR) = emR
     remRegs(trcId,in(r,S)) =
        if(eq(trcId,trcId(r)),
           remRegs(trcId, S), in(r,remRegs(trcId, S)))
     leases(S) = leases(S, emL)
     leases(emR, L) = L
     leases(in(r, S), L) = leases(S, in(lease(r), L))

% Event List:

sort EventList
func emEv:->EventList
     in:Event#EventList->EventList
map  first:EventList->Event
     tail:EventList->EventList
     get:EventList#Event->Bool
     rem:Event#EventList->EventList
     len:EventList->Nat
     if:Bool#EventList#EventList->EventList
var  e,e’:Event
     E,E’:EventList
rew  first(in(e, E)) = e
     tail(in(e, E)) = E
     len(emEv) = 0
     len(in(e, E)) = S(len(E))
     get(emEv,e) = F
     get(in(e,E),e’) = or(eq(e,e’),get(E,e’))
     rem(e,emEv) = emEv
     rem(e,in(e’,E)) = if(eq(e,e’),E,in(e’,rem(e,E)))
     if(T,E,E’) = E
     if(F,E,E’) = E’
```

A.5 Action Definitions

```
act  write, Write, W : Entry#Nat#Time
     write, Write, W : Entry#Nat#Time#Nat
     take, Take, T : Nat#Nat#Time#Query
     read, Read, R : Nat#Nat#Time#Query
     readIfExists, ReadIfExists, RIE : Nat#Nat#Time#Query
     takeIfExists, TakeIfExists, TIE : Nat#Nat#Time#Query
     takeReturn, TakeReturn, TReturn : Nat#Entry
     readReturn, ReadReturn, RReturn : Nat#Entry
     readIfExistsReturn, ReadIfExistsReturn,
        RIEReturn : Nat#Entry
     takeIfExistsReturn, TakeIfExistsReturn,
        TIEReturn : Nat#Entry
     create, Create, C : TransactionCreated#Time
     commit, Commit, Cm : Nat
     abort, Abort, A : Nat
     exception, Exception, E : Nat
     notify, Notify, N: Nat#Time#Query#EventRegistration
     _notify, __Notify, _N: Event
     __notify, _Notify, __N: Nat#Nat
     tick
     duplicate loose
     gc: Entry
     gc: Registration
     renew,Renew,Rnew: Nat#Time

comm write|Write = W
     take|Take = T
     read|Read = R
     readIfExists|ReadIfExists = RIE
     takeIfExists|TakeIfExists = TIE
     takeReturn|TakeReturn = TReturn
     readReturn|ReadReturn = RReturn
     readIfExistsReturn|ReadIfExistsReturn = RIEReturn
     takeIfExistsReturn|TakeIfExistsReturn = TIEReturn
     create|Create = C
     commit|Commit = Cm
     abort|Abort = A
     exception|Exception = E
     notify|Notify = N
     _notify|__Notify = _N
     __notify|_Notify = __N
     renew|Renew = Rnew
```

A.6 JavaSpace Process

```
proc javaspace(M:ObjectSet, PA:ActionSet,
               Trc:TransactionSet, Reg: RegistrationSet) =

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   WRITE WITHOUT TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   sum(e:Entry, sum(trcId:Nat, sum(timeout:Time,
      sum(objectId: Nat, sum(leaseId: Nat,
      Write(e, trcId, timeout)
      .javaspace(in(object(objectId, e,
         lease(leaseId, timeout)), M),
            PA, Trc, match(e, trcId, Reg))
   <| and(and(and(and(and(and(
      eq(trcId, NULL),
      not(eq(e, entryNull))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      eq(objectId, freshId(U(M, UWsets(Trc))))),
      lt(objectId, maxObjects)),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |>
   delta)))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   WRITE WITH TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(e:Entry, sum(trcId:Nat, sum(timeout:Time,
      sum(trc: Transaction, sum(objectId: Nat,
         sum(leaseId: Nat,
      Write(e, trcId, timeout)
      .javaspace(M, PA, in(inW(object(objectId, e,
         lease(leaseId, timeout)), trc),
            rem(trc, Trc)), match(e, trcId, Reg))
   <| and(and(and(and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      not(eq(e, entryNull))),
      or(lt(timeout, maxTime), eq(timeout, FOREVER))),
      eq(objectId, freshId(U(M, UWsets(Trc))))),
      lt(objectId, maxObjects)),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |>
   delta))))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   WRITE WITHOUT TRANSACTION (returning the lease)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(e:Entry, sum(trcId:Nat, sum(timeout:Time,
   sum(objectId: Nat, sum(leaseId: Nat,
      Write(e, trcId, timeout, leaseId)
      .javaspace(in(object(objectId, e,
         lease(leaseId, timeout)), M),
            PA, Trc, match(e, trcId, Reg))
   <| and(and(and(and(and(and(
      eq(trcId, NULL),
      not(eq(e, entryNull))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      eq(objectId, freshId(U(M, UWsets(Trc))))),
      lt(objectId, maxObjects)),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |>
   delta)))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   WRITE WITH TRANSACTION (returning the lease)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(e:Entry, sum(trcId:Nat, sum(timeout:Time,
      sum(trc: Transaction, sum(objectId: Nat,
         sum(leaseId: Nat,
      Write(e, trcId, timeout, leaseId)
      .javaspace(M, PA, in(inW(object(objectId, e,
         lease(leaseId, timeout)), trc),
            rem(trc, Trc)), match(e, trcId, Reg))
   <| and(and(and(and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      not(eq(e, entryNull))),
      or(lt(timeout, maxTime), eq(timeout, FOREVER))),
      eq(objectId, freshId(U(M, UWsets(Trc))))),
      lt(objectId, maxObjects)),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |>
   delta))))))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TAKE PRIMITIVE WITHOUT TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      Take(procId, trcId, timeout, query)
      .javaspace(M, in(action(takeA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions))
   |>
   delta))))

% NULL RETURN
   +
   sum(a:Action,
      takeReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      eq(type(a), takeA)),
      eq(trcId(a), NULL)),
      isExpired(a)),
      not(matches(query(a), M)))
   |>
   delta)

% NON-NULL RETURN
   +
   sum(a: Action, sum(o: Object,
      takeReturn(procId(a), entry(o))
      .javaspace(rem(o, M), rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(
      get(PA, a),
      get(M, o)),
      eq(type(a), takeA)),
      eq(trcId(a), NULL)),
      match(query(a), entry(o))),
      not(readLocked(o, Trc)))
         % Not read locked by any transaction
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   READ PRIMITIVE WITHOUT TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      Read(procId, trcId,  timeout, query)
      .javaspace(M, in(action(readA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions))
   |>
   delta))))

% NULL RETURN
   +
   sum(a:Action,
      readReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      eq(type(a), readA)),
      eq(trcId(a), NULL)),
      isExpired(a)),
      not(matches(query(a), M)))
   |>
   delta)

% NON-NULL RETURN
   +
   sum(a: Action, sum(o: Object,
      readReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      get(M, o)),
      eq(type(a), readA)),
      eq(trcId(a), NULL)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   READ_IF_EXISTS PRIMITIVE WITHOUT TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% There are matching entries in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      ReadIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(readEA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      matches(query, U(M, U(UWsets(Trc), UTsets(Trc)))))
   |>
   delta))))

% There are not matching entries in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      ReadIfExists(procId, trcId,  timeout, query)
      .javaspace(M,  in(action(readEA, trcId,
         procId, tt(0), query), PA), Trc, Reg)
   <| and(and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      not(matches(query, U(M, U(UWsets(Trc), UTsets(Trc))))))
   |>
   delta))))

% NULL RETURN
   +
   sum(a:Action,
      readIfExistsReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      eq(type(a), readEA)),
      eq(trcId(a), NULL)),
      isExpired(a)),
      not(matches(query(a), M)))
   |>
   delta)

% NON-NULL RETURN
   +
   sum(a: Action, sum(o: Object,
      readIfExistsReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      get(M, o)),
      eq(type(a), readEA)),
      eq(trcId(a), NULL)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TAKE_IF_EXISTS PRIMITIVE WITHOUT TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% There are matching entries in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      TakeIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(takeEA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      matches(query, U(M, U(UWsets(Trc), UTsets(Trc)))))
   |>
   delta))))
```

```
% There are not matching entries in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query,
      TakeIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(takeEA, trcId,
         procId, tt(0), query), PA), Trc, Reg)
   <| and(and(and(
      eq(trcId, NULL),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      not(matches(query, U(M, U(UWsets(Trc), UTsets(Trc))))))
   |>
   delta))))

% NULL RETURN
   +
   sum(a:Action,
      takeIfExistsReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(
      get(PA, a),
      eq(type(a), takeEA)),
      eq(trcId(a), NULL)),
      isExpired(a)),
      not(matches(query(a), M)))
   |>
   delta)

% NON-NULL RETURN
   +
   sum(a: Action, sum(o: Object,
      takeIfExistsReturn(procId(a), entry(o))
      .javaspace(rem(o, M), rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(
      get(PA, a),
      get(M, o)),
      eq(type(a), takeEA)),
      eq(trcId(a), NULL)),
      match(query(a), entry(o))),
      not(readLocked(o, Trc)))
         % Not read by any transactions
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TAKE PRIMITIVE WITH TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      Take(procId, trcId,  timeout, query)
      .javaspace(M, in(action(takeA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions))
   |>
   delta)))))

% NULL RETURN
   +
   sum(a:Action, sum(trc: Transaction,
      takeReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(Trc, trc)),
      eq(trcId(a), id(trc))),
      get(PA, a)),
      eq(type(a), takeA)),
      isExpired(a)),
      not(matches(query(a), M))),
      not(matches(query(a), Wset(trc))))
   |>
   delta))

% NON-NULL RETURN (entry is in M)
   +
   sum(a:Action, sum(o: Object,
      takeReturn(procId(a), entry(o))
      .javaspace(rem(o, M), rem(a, PA),
         in(inT(o, getById(Trc, trcId(a))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(M, o)),
      eq(type(a), takeA)),
      or(eq(timeout(lease(o)), FOREVER),
         lt(timeout(lease(getById(Trc, trcId(a)))),
            timeout(lease(o))))),
      match(query(a), entry(o))),
      not(readLocked(o, rem(getById(Trc, trcId(a)), Trc))))
   |>
   delta))
```

```
% NON-NULL RETURN (entry is in Trc)
   +
   sum(a:Action, sum(o: Object,
      takeReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), in(remR(o, remW(o,
         getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(Wset(getById(Trc, trcId(a))), o)),
      eq(type(a), takeA)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   READ PRIMITIVE WITH TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      Read(procId, trcId,  timeout, query)
      .javaspace(M, in(action(readA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions))
   |>
   delta)))))

% NULL RETURN
   +
   sum(a:Action, sum(trc: Transaction,
      readReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(Trc, trc)),
      eq(trcId(a), id(trc))),
      get(PA, a)),
      eq(type(a), readA)),
      isExpired(a)),
      not(matches(query(a), M))),
      not(matches(query(a), Wset(trc))))
   |>
   delta))
```

```
% NON-NULL RETURN (entry is in M)
   +
   sum(a:Action, sum(o: Object,
      readReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA),  in(inR(o,
         remR(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(M, o)),
      eq(type(a), readA)),
      or(eq(timeout(lease(o)), FOREVER),
         lt(timeout(lease(getById(Trc, trcId(a)))),
            timeout(lease(o))))),
      match(query(a), entry(o)))
   |>
   delta))

% NON-NULL RETURN (entry is in Trc)
   +
   sum(a:Action, sum(o: Object,
      readReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), in(inR(o,
         remR(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(Wset(getById(Trc, trcId(a))), o)),
      eq(type(a), readA)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   TAKE_IF_EXISTS PRIMITIVE WITH TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% There are matches in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      TakeIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(takeEA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      matches(query, U(M, U(UWsets(Trc), UTsets(Trc)))))
   |>
   delta)))))

% There are not matches in M or in Trc
  +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      TakeIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(takeEA, trcId,
         procId, tt(0), query), PA), Trc, Reg)
   <| and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      not(matches(query, U(M, U(UWsets(Trc), UTsets(Trc))))))
   |>
   delta)))))

% NULL RETURN
   +
   sum(a:Action, sum(trc: Transaction,
      takeIfExistsReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(Trc, trc)),
      eq(trcId(a), id(trc))),
      get(PA, a)),
      eq(type(a), takeEA)),
      isExpired(a)),
      not(matches(query(a), M))),
      not(matches(query(a), Wset(trc))))
   |>
   delta))

% NON-NULL RETURN (entry is in M)
   +
   sum(a:Action, sum(o: Object,
      takeIfExistsReturn(procId(a), entry(o))
      .javaspace(rem(o, M), rem(a, PA), in(inT(o,
         remT(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(M, o)),
      eq(type(a), takeEA)),
      or(eq(timeout(lease(o)), FOREVER),
         lt(timeout(lease(getById(Trc, trcId(a)))),
            timeout(lease(o))))),
      match(query(a), entry(o)))   ,
      not(readLocked(o, rem(getById(Trc, trcId(a)), Trc))))
   |>
   delta))

% NON-NULL RETURN (entry is in Trc)
   +
   sum(a:Action, sum(o: Object,
      takeIfExistsReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), in(remR(o,
         remW(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(Wset(getById(Trc, trcId(a))), o)),
      eq(type(a), takeEA)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   READ_IF_EXISTS PRIMITIVE WITH TRANSACTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% There are matches in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      ReadIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(readEA, trcId,
         procId, timeout, query), PA), Trc, Reg)
   <| and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      matches(query, U(M, U(UWsets(Trc), UTsets(Trc)))))
   |>
   delta)))))

% There are not matches in M or in Trc
   +
   sum(procId:Nat, sum(trcId:Nat, sum(timeout:Time,
      sum(query:Query, sum(trc:Transaction,
      ReadIfExists(procId, trcId,  timeout, query)
      .javaspace(M, in(action(readEA, trcId,
         procId, tt(0), query), PA), Trc, Reg)
   <| and(and(and(and(and(
      not(eq(trcId, NULL)),
      get(Trc, trc)),
      eq(trcId, id(trc))),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      lt(len(PA), maxActions)),
      not(matches(query, U(M, U(UWsets(Trc), UTsets(Trc))))))
   |>
   delta)))))

% NULL RETURN
   +
   sum(a:Action, sum(trc: Transaction,
      readIfExistsReturn(procId(a), entryNull)
      .javaspace(M, rem(a, PA), Trc, Reg)
   <| and(and(and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(Trc, trc)),
      eq(trcId(a), id(trc))),
      get(PA, a)),
      eq(type(a), readEA)),
      isExpired(a)),
      not(matches(query(a), M))),
      not(matches(query(a), Wset(trc))))
   |>
   delta))

% NON-NULL RETURN (entry is in M)
   +
   sum(a:Action, sum(o: Object,
      readIfExistsReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), in(inR(o,
         remR(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(M, o)),
      eq(type(a), readEA)),
      or(eq(timeout(lease(o)), FOREVER),
         lt(timeout(lease(getById(Trc, trcId(a)))),
            timeout(lease(o))))),
      match(query(a), entry(o)))
   |>
   delta))
```

```
% NON-NULL RETURN (entry is in Trc)
   +
   sum(a:Action, sum(o: Object,
      readIfExistsReturn(procId(a), entry(o))
      .javaspace(M, rem(a, PA), in(inR(o,
         remR(o, getById(Trc, trcId(a)))),
            rem(getById(Trc, trcId(a)), Trc)), Reg)
   <| and(and(and(and(
      not(eq(trcId(a), NULL)),
      get(PA, a)),
      get(Wset(getById(Trc, trcId(a))), o)),
      eq(type(a), readEA)),
      match(query(a), entry(o)))
   |>
   delta))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  TRANSACTIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% CREATE TRANSACTION
   +
   sum(timeout:Time, sum(trcId:Nat, sum(leaseId:Nat,
      Create(trcCreated(trcId, leaseId), timeout)
     .javaspace(M, PA, in(transaction(trcId,
        lease(leaseId, timeout), emO, emO, emO), Trc), Reg)
   <| and(and(and(and(
      eq(trcId, freshId(Trc)),
      lt(trcId, maxTrc)),
      or(lt(timeout, maxTime), eq(timeout,FOREVER))),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |> delta)))

% COMMIT
   +
   sum(trcId: Nat, sum(trc: Transaction,
      Commit(trcId)
      .javaspace(U(M, Wset(trc)), PA,
         rem(trc, Trc), remRegs(id(trc), Reg))
   <| and(
      get(Trc, trc),
      eq(trcId, id(trc)))
   |> delta))

% ABORT
   +
   sum(trcId: Nat, sum(trc: Transaction,
   Abort(trcId)
   .javaspace(U(M, Tset(trc)), PA,
      rem(trc, Trc), remRegs(id(trc), Reg))
   <| and(
      get(Trc, trc),
      eq(trcId, id(trc)))
   |> delta))

% TIMEOUT
   +
   sum(trc: Transaction,
   exception(id(trc))
   .javaspace(U(M, Tset(trc)), remPAs(id(trc), PA),
      rem(trc, Trc), remRegs(id(trc), Reg))
   <| and(
      get(Trc, trc),
      isExpired(trc))
   |> delta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  NOTIFICATION MECHANISM
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% REGISTRATION
   +
   sum(trcId: Nat, sum(timeout:Time, sum(query:Query,
   sum(regId: Nat, sum(leaseId:Nat,
      Notify(trcId, timeout, query, evReg(regId, leaseId))
     .javaspace(M, PA, Trc, in(registration(regId, trcId,
        lease(leaseId, timeout), query, 0, F), Reg))
   <| and(and(and(and(
      or(lt(timeout, maxTime), eq(timeout,FOREVER)),
      lt(len(Reg), maxRegistrations)),
      eq(regId, freshId(Reg))),
      eq(leaseId,
         freshId(U(leases(M), U(leases(Trc), leases(Reg)))))),
      lt(leaseId, maxLeases))
   |> delta)))))

% TIMEOUT
   +
   sum(reg: Registration,
      gc(reg)
      .javaspace(M, PA, Trc, rem(reg, Reg))
   <| and(
      get(Reg, reg),
      isExpired(reg))
   |> delta)

% SEND EVENT
```

```
   +
   sum(reg:Registration,
       _notify(event(id(reg), seq(reg)))
      .javaspace(M, PA, Trc, in(notify(reg), rem(reg, Reg)))
   <| and(and(
      get(Reg, reg),
      not(notified(reg))),
      lt(0, seq(reg)))
   |> delta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  LEASE RENEWAL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% LEASE OF OBJECT
   +
   sum(leaseId:Nat, sum(timeout: Time, sum(o: Object,
      Renew(leaseId, timeout)
      .javaspace(in(object(id(o), entry(o),
         lease(leaseId, timeout)), rem(o,M)), PA, Trc, Reg)
   <| and(
      get(M,o),
      eq(id(lease(o)), leaseId))
   |> delta)))

% LEASE OF TRANSACTION
   +
   sum(leaseId:Nat, sum(timeout: Time, sum(trc: Transaction,
      Renew(leaseId, timeout)
      .javaspace(M, PA, in(transaction(id(trc),
         lease(leaseId, timeout), Wset(trc), Tset(trc),
            Rset(trc)), rem(trc, Trc)), Reg)
   <| and(
      get(Trc,trc),
      eq(id(lease(trc)), leaseId))
   |> delta)))

% LEASE OF REGISTRATION
   +
   sum(leaseId:Nat, sum(timeout: Time, sum(reg: Registration,
      Renew(leaseId, timeout)
      .javaspace(M, PA, Trc,
         in(registration(id(reg), trcId(reg),
            lease(leaseId, timeout), query(reg),
               seq(reg), notified(reg)), rem(reg, Reg)))
   <| and(
      get(Reg,reg),
      eq(id(lease(reg)), leaseId))
   |> delta)))
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  GARBAGE COLLECTOR
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   +
   sum(o:Object,
      gc(entry(o))
      .javaspace(rem(o,M), PA, Trc, Reg)
   <| and(and(
      get(M,o),
      isExpired(o)),
      not(match(o, PA)))
   |> delta)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      CLOCK
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% TICK
   +
   tick.javaspace(decT(M), decT(PA), decT(Trc), decT(Reg))
   <| and(
      or(or(or(
      areTimeouts(M),
      areTimeouts(PA)),
      areTimeouts(Trc)),
      areTimeouts(Reg)),
      and(and(and(
      not(areExpired(M)),
      not(areExpired(PA))),
      not(areExpired(Trc))),
      not(areExpired(Reg)))
      )
    |> delta
```

A.7 Auxiliary Items

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    AUXILIARY CONSTANTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map  maxObjects:->Nat
rew  maxObjects = 6
map  maxActions:->Nat
rew  maxActions =  6
map  maxTime:->Time
rew  maxTime = tt(3)
map  maxEvents:->Nat
rew  maxEvents = 3
map  maxRegistrations:->Nat
```

```
rew  maxRegistrations = 3
map  maxLeases:->Nat
rew  maxLeases = plus(maxObjects,
   plus(maxTrc, maxRegistrations))
map  NULL:->Nat
rew  NULL = 0
map  maxTrc:->Nat
rew  maxTrc = 3

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EVENT COMMUNICATION CHANEL (bounded)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc Network(E:EventList) =

% RECEIVE AN EVENT
   sum(e:Event,
   __Notify(e).Network(in(e, E))
   <| lt(len(E), maxEvents) |> delta)

% DELIVER AN UNORDERED  EVENT
   +
   sum(e:Event,
   __notify(registrationId(e), seq(e))
   .Network(rem(e, E))
   <| get(E,e) |> delta)

% DUPLICATE
   +
  duplicate.Network(in(e, E))
   <| and(and(
      lt(len(E), maxEvents),
      lt(0, len(E))),
      get(E,e))
   |> delta

% LOSE HEAD
   +
   lose.Network(tail(E))
   <| lt(0, len(E)) |> delta
```

A.8 Summation System

```
sort  Entry
func  entryNull:->Entry
      number:Nat->Entry
      lock:->Entry
map   eq:Entry#Entry->Bool
      value:Entry->Nat
```

```
var   e:Entry
      n, n’:Nat
rew   eq(entryNull,entryNull) =  T
      eq(entryNull,number(n)) =  F
      eq(number(n), entryNull) =  F
      eq(number(n),number(n’)) =  eq(n, n’)
      eq(lock,lock) =  T
      eq(entryNull,lock) =  F
      eq(lock, entryNull) =  F
      eq(lock,number(n)) =  F
      eq(number(n), lock) =  F
      value(number(n)) = n

sort  Query
func  any:->Query
      anyNumber:->Query
      anyLock:->Query
      noEntry:->Query
map   match:Query#Entry->Bool
      eq:Query#Query->Bool
var   e:Entry
      n:Nat
rew   match(any, e) = T
      match(anyNumber, number(n)) = T
      match(anyNumber, lock) = F
      match(anyLock, lock) = T
      match(anyLock, number(n)) = F
      match(noEntry, e) = F
      eq(any,any) = T
      eq(any,anyNumber) = F
      eq(any,anyLock) = F
      eq(any,noEntry) = F
      eq(anyNumber,anyNumber) = T
      eq(anyNumber,any) = F
      eq(anyNumber,anyLock) = F
      eq(anyNumber,noEntry) = F
      eq(anyLock,anyLock) = T
      eq(anyLock,noEntry) = F
      eq(anyLock,any) = F
      eq(anyLock,anyNumber) = F
      eq(noEntry,noEntry) = T
      eq(noEntry,anyLock) = F
      eq(noEntry,anyNumber) = F
      eq(noEntry,any) = F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

act   waiting safe
      wrote, M_wrote, W_wrote, SM_wrote:Nat
      publish:Nat
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

map   t_ma:->Time
rew   t_ma =  tt(2)
map   t_opM:->Time
rew   t_opM =  tt(1)
map   t_op:->Time
rew   t_op =  tt(0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc Master(id:Nat) =
sum(trc:Nat, sum(leaseId:Nat,
   create(trcCreated(trc, leaseId), t_ma)
   .(write(lock, trc, FOREVER) + ExM(id, trc))
   .(take(id, trc, t_opM, noEntry) + ExM(id, trc))
   .(TakeReturn(id, entryNull) + ExM(id, trc))
        .(take(id, trc, tt(0), anyNumber)+ ExM(id, trc))
        .(sum(e1:Entry, TakeReturn(id, e1)
      .(((take(id, trc, tt(0), anyNumber) + ExM(id, trc))
      .(sum(e2:Entry, TakeReturn(id, e2)
         .((commit(trc) + ExM(id, trc))
         .publish(value(e1)).delta
         <|eq(e2, entryNull)|>
         (write(number(plus(value(e1),
               value(e2))), trc, FOREVER)
            + ExM(id, trc))
         .(take(id, trc, tt(0), anyLock) + ExM(id, trc))
         .(TakeReturn(id, lock) + ExM(id, trc))
         .(commit(trc)  + ExM(id, trc))
         .M_wrote(plus(value(e1), value(e2)))))
      + ExM(id, trc)))
      <| not(eq(e1, entryNull)) |>
      abort(trc).delta))
        + ExM(id, trc))))
.Master(id)

proc ExM(id:Nat, trc:Nat) =
  Exception(trc).(Master(id) + safe.SafeMaster(id))

proc SafeMaster(id:Nat) =
sum(trc:Nat, sum(leaseId:Nat,
   create(trcCreated(trc, leaseId), FOREVER)
   .write(lock, trc, FOREVER)
   .take(id, trc, t_opM, noEntry)
   .TakeReturn(id, entryNull)
        .take(id, trc, tt(0), anyNumber)
        .sum(e1:Entry, TakeReturn(id, e1)
      .(take(id, trc, tt(0), anyNumber)
      .sum(e2:Entry, TakeReturn(id, e2)
         .(commit(trc)
         .publish(value(e1)).delta
         <|eq(e2, entryNull)|>
         write(number(plus(value(e1),
            value(e2))), trc, FOREVER)
         .take(id, trc, tt(0), anyLock)
         .TakeReturn(id, lock)
         .commit(trc)
         .SM_wrote(plus(value(e1), value(e2)))))
      <| not(eq(e1, entryNull)) |>
      abort(trc).delta))))
.SafeMaster(id)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proc Worker(id:Nat) =
sum(trc:Nat, sum(leaseId:Nat,
   create(trcCreated(trc, leaseId), t_op)
   .(readIfExists(id, trc, FOREVER, anyLock) + ExW(id, trc))
   .(sum(l:Entry, ReadIfExistsReturn(id, l)
      .((take(id, trc, tt(0), anyNumber) + ExW(id, trc))
      .(sum(e1:Entry, TakeReturn(id, e1)
         .((take(id, trc, tt(0), anyNumber) + ExW(id, trc))
         .(sum(e2:Entry, TakeReturn(id, e2)
            .((write(number(plus(value(e1),
                  value(e2))),trc,FOREVER)
               + ExW(id, trc))
            .(W_wrote(plus(value(e1),
                  value(e2))) + ExW(id, trc))
            .commit(trc)
            <| not(eq(e2, entryNull)) |>
            abort(trc).Worker(id)))
            + ExW(id, trc))
         <| not(eq(e1, entryNull)) |>
         abort(trc).Worker(id)))
      + ExW(id, trc))
      <| eq(l, entryNull) |>
      abort(trc).Worker(id)))
   + ExW(id, trc))))
.Worker(id)

proc ExW(id:Nat, trc:Nat) = Exception(trc).Worker(id)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Sys = hide({
         tick, W, T, TReturn, R, RReturn,
         C, Cm, A, RIE, E, RIEReturn, TIE, TIEReturn,
         _N, __N, N, Rnew},
      encap({
         write, Write,
         take, Take, takeReturn, TakeReturn,
         read, Read, readReturn, ReadReturn,
         readIfExists, ReadIfExists,
         takeIfExists, TakeIfExists,
         readIfExistsReturn, ReadIfExistsReturn,
         takeIfExistsReturn, TakeIfExistsReturn,
         notify, Notify, _notify, __Notify, __notify, _Notify,
         create, Create, commit, Commit, abort, Abort,
         exception, Exception,
         renew, Renew},
      javaspace(M2, emA, emT, emR) || Master(0) || Worker(1)))

init Sys

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

func  M1, M2, M3, M4, M5:->ObjectSet
rew   M1 = in(object(0, number(1), lease(0, FOREVER)), emO)
      M2 = in(object(1, number(1), lease(1, FOREVER)), M1)
      M3 = in(object(2, number(1), lease(2, FOREVER)), M2)
      M4 = in(object(3, number(1), lease(3, FOREVER)), M3)
      M5 = in(object(4, number(1), lease(4, FOREVER)), M4)
```

Bibliography

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *CONCUR*, volume 3170 of *LNCS*, pages 35–48. Springer, 2004.

[2] R. Alur, R.. Brayton, T. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Formal Methods in System Design*, 18(2):97–116, 2001.

[3] K.R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 22(6):307–309, 1986.

[4] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[5] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[6] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[7] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In *WADT*, volume 1827 of *LNCS*. Springer, 2000.

[8] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *TCS*, pages 77–121, 1985.

[9] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra.* North-Holland, 2001.

[10] G. Berry and G. Boudol. The chemical abstract machine. *TCS*, 96(1):217–248, 1992.

[11] M. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. In *The Computer Journal*, volume 37(4), pages 289–307, 1994.

[12] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[13] S. Blom, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. New developments around the μCRL tool set. *ENTCS*, 80, 2003.

[14] S. Blom and S.M. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, page To appear, 2005.

[15] S. Blom and J. van de Pol. State space reduction by proving confluence. In *CAV*, volume 2404 of *LNCS*, pages 596–609. Springer, 2002.

[16] M. Boasson. Control systems software. In *Transactions on Automatic Control*, pages 1094–1107. IEEE, 1993.

[17] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *SAC*, pages 146–155. ACM, 1999.

[18] D. Bosnacki, N. Ioustinova, and N. Sidorova. Using fairness to make abstractions work. In *SPIN*, volume 2989 of *LNCS*, pages 198–215. Springer, 2004.

[19] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*. Springer, 2000.

[20] D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *ACM*, 31(3):560–599, 1984.

[21] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *CAV*, volume 1877 of *LNCS*, pages 274–287. Springer, 1999.

[22] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In *AMAST*, volume 1816 of *LNCS*, pages 198–212. Springer, 2000.

[23] N. Busi and G. Zavattaro. On the serializability of transactions in JavaSpaces. *ENTCS*, 54, 2001.

[24] N. Busi and G. Zavattaro. Publish/subscribe v.s. shared dataspace coordination infrastructures. In *WETICE*, pages 328–333. IEEE, 2001.

[25] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[26] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM*, pages 343–354, 1992.

[27] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM*, 28(4):626–643, 1996.

[28] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 138 – 143. Springer, 2001.

[29] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. *ACM*, pages 238–252, 1977.

[30] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *ACM*, pages 269–282, 1979.

[31] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[32] D. Dams. Abstraction in software model checking: Principles and practice (tutorial overview and bibliography). In *SPIN*, volume 2318 of *LNCS*, pages 14–21. Springer, 2002.

[33] D. Dams and R. Gerth. The bounded retransmission protocol revisited. *ENTCS*, 9, 2000.

[34] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM*, pages 253–291, 1997.

[35] D. Dams, W. Hesse, and G. Holzmann. Abstracting C with abC. In *CAV*, volume 2404 of *LNCS*, pages 515–520. Springer, 2002.

[36] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.

[37] P. Dechering and I. van Langevelde. The verification of coordination. In *COORDINATION*, volume 1906 of *LNCS*, pages 335–340. Springer, 2000.

[38] P.H.J. van (eds) Eijk, C.A. Vissers, and M. Diaz. The formal description technique LOTOS, 1989.

[39] E.A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000.

[40] E.A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *TACAS*, volume 2619 of *LNCS*, pages 144–159. Springer, 2003.

[41] Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *VMCAI*, volume 2937 of *LNCS*, pages 223–238. Springer, 2004.

[42] A. Fantechi, S. Gnesi, and D. Latella. Towards automatic temporal logic verification of value passing process algebra using abstract interpretation. In *CONCUR*, volume 1119 of *LNCS*, pages 562–578. Springer, 1996.

[43] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *CAV*, volume 1102 of *LNCS*, pages 437–440. Springer, 1996.

[44] D. Fisman and A. Pnueli. Beyond regular model checking. In *FSTTCS*, volume 2245 of *LNCS*, pages 156–170. Springer, 2001.

[45] W. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. Springer, 2000.

[46] W. Fokkink, J.F. Groote, J. Pang, B. Badban, and J. van de Pol. Verifying a sliding window protocol in μCRL. In *AMAST*, volume 3116 of *LNCS*, pages 148–163. Springer, 2004.

[47] W. Fokkink, N. Ioustinova, E. Kesseler, J. van de Pol, Y. Usenko, and Y. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *CONCUR*, volume 2421 of *LNCS*, pages 1–23. Springer, 2002.

[48] W. Fokkink and J. Pang. Cones and foci for protocol verification revisited. In *FOSSACS*, volume 2620 of *LNCS*, pages 267–281. Springer, 2003.

[49] W. Fokkink and J. Pang. Simplifying Itai-Rodeh leader election for anonymous rings. *ENTCS*, 128:53–68, 2005.

[50] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999.

[51] M. M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. αSPIN: A tool for abstract model checking. *STTT*, 5(2-3):165–184, 2004.

[52] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, 2002.

[53] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, volume 38 of *IFIP*, pages 3–18, 1995.

[54] J. Giesl, R., P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

[55] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *TACAS*, volume 2154 of *LNCS*, pages 426–440. Springer, 2001.

[56] J. F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. In *AMAST*, volume 1101 of *LNCS*, pages 536–550. Springer, 1996.

[57] J. F. Groote and A. Ponse. The syntax and semantics of μCRL. In *ACP*, Workshops in Computing Series, pages 26–62, 1995.

[58] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in μCRL. In *ACP*, Workshops in Computing Series, pages 63–86, 1995.

[59] J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summation algorithm. *Formal Aspects of Computing*, 2004.

[60] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *JLAP*, 56(1-2):21–56, 2003.

[61] J.F. Groote, A. Ponse, and Y. Usenko. Linearization in parallel pCRL. *JLAP*, 48(1-2):39–70, 2001.

[62] J.F. Groote and M.A. Reniers. Algebraic process verification. In *Handbook of Process Algebra*, pages 1151–1208. North-Holland, 2001.

[63] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *JLAP*, 49(1-2):31–60, 2001.

[64] J.F. Groote and J.J. van Wamel. Algebraic data types and induction in μCRL. Technical report, Universiteit van Amsterdam, 1994.

[65] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. *TCS*, 266(1-2):65–75, 2001.

[66] J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56:251–273, 2005.

[67] S. L. Halter. *JavaSpaces example by example.* Prentice Hall PTR, 2002.

[68] D. Harel, A. Pnueli, and J. Stavi. Propositional dynamic logic of context-free programs. *FOCS*, pages 310–321, 1981.

[69] J. Hatcliff, M. Dwyer, C. Pasareanu, and Robby. Foundations of the Bandera abstraction tools. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 172 – 203. Springer, 2002.

[70] K. Havelund and J. Skakkebaek. Applying Model Checking in Java Verification. In *SPIN*, volume 1680 of *LNCS*, pages 216–232. Springer, 1999.

[71] M. Hennessey and R. Milner. On observing nondeterminism and concurrency. In *ICALP*, volume 85 of *LNCS*, pages 295–309. Springer, 1980.

[72] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[73] G.J. Holzmann and M.H. Smith. A practical method for verifying event-driven software. In *ICSE*. ACM, 1999.

[74] J.M.M. Hooman and J. van de Pol. Formal verification of replication on a distributed data space architecture. In *SAC*, pages 351–358. ACM, 2002.

[75] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: a foundation for three-valued program analysis. In *VMCAI*, volume 2294 of *LNCS*, pages 155–169. Springer, 2001.

[76] C.N. Ip and D.L. Dill. Verifying systems with replicated components in Murϕ. In *CAV*, volume 1102 of *LNCS*, pages 147–158. Springer, 1996.

[77] N. D. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–636. Oxford Science Publications, 1995.

[78] P. Kelb. Model checking and abstraction: a framework preserving both truth and failure information. Technical report, University of Oldenburg, 1994.

[79] Y. Kesten and A. Pnueli. Verifying liveness by augmented abstraction. In *CSL*, volume 1683 of *LNCS*, pages 141–145. Springer, 1999.

[80] Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *STTT*, 2(4):328–342, 2000.

[81] D. Kozen. Results on the propositional μ-calculus. In *ICALP*, volume 140 of *LNCS*, pages 348–359. Springer, 1982.

[82] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In *ICFEM*, volume 3380 of *LNCS*, pages 224–238. Springer, 2004.

[83] K. G. Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer, 1989.

[84] K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE, 1988.

[85] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, pages 11–44, 1995.

[86] S. Luttik. Description and formal specification of the Link Layer of P1394. In *WAFMSD*, 1997.

[87] Z. Manna, M. Colon, B. Finkbeiner, H. Sipma, and T. E. Uribe. Abstraction and modular verification of infinite-state reactive systems. *Requirements Targeting Software and Systems Engineering*, pages 273–292, 1997.

[88] R. Mateescu. *Verification des proprietes temporelles des programmes paralleles*. PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[89] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free μ-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[90] K. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1993.

[91] SUN Microsystems. *JavaSpaces*$^{\text{tm}}$ *Service Specification*, 2000. See `http://java.sun.com/products/javaspaces/`.

[92] SUN Microsystems. *Jini*$^{\text{tm}}$ *Technology Core Platform Specification*, 2000. See `http://www.sun.com/jini/specs/`.

[93] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[94] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[95] V. K. Murthy and E. V. Krishnamurthy. Gamma programming paradigm and heterogeneous computing. In *HICSS*, page 273. IEEE, 1996.

[96] O. Ore. Galois connexions. *Trans. American Mathematics Society*, 55:493–513, 1944.

[97] S. Orzan. *On distributed verification and verified distribution*. PhD thesis, Free University Amsterdam, 2004.

[98] S. Orzan, J. van de Pol, and M. Valero Espada. A state space distribution policy based on abstract interpretation. *ENTCS*, 128:35–45, 2005.

[99] J. Pang, W. Fokkink, R. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *IPDPS*, page 238. IEEE, 2003.

[100] J. Pang, J. van de Pol, and M. Valero Espada. Abstraction of parallel uniform processes with data. In *SEFM*, pages 14–23. IEEE, 2004.

[101] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[102] A. Pnueli. Abstraction for liveness. In *VMCAI*, volume 3385 of *LNCS*, pages 146–164. Springer, 2005.

[103] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV*, volume 1855 of *LNCS*, pages 328–343. Springer, 2000.

[104] J. van de Pol. A prover for the μCRL toolset with applications. Technical Report SEN-R0106, CWI, 2001.

[105] J. van de Pol and M. Valero Espada. An abstract interpretation toolkit for μCRL. *ENTCS*, 133:295–313, 2005.

[106] J. van de Pol and M. Valero Espada. Formal specification of JavaSpaces$^{\text{TM}}$ architecture using μCRL. In *COORDINATION*, volume 2315 of *LNCS*, pages 274–290. Springer, 2002.

[107] J. van de Pol and M. Valero Espada. μCRL specification of event notification in JavaSpaces, 2002.

[108] J. van de Pol and M. Valero Espada. Verification of JavaSpaces parallel programs. In *ACSD*, pages 196–205. IEEE, 2003.

[109] J. van de Pol and M. Valero Espada. Modal abstraction in μCRL. In *AMAST*, volume 3116 of *LNCS*, pages 409–425. Springer, 2004.

[110] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. In *TPDS*, pages 773–787. IEEE, 1995.

[111] J.M. Rushby. Formal methods and their role in the certification of critical systems. Technical report, CSL, 1995.

[112] D. Schmidt. Binary relations for abstraction and refinement, 1999.

[113] M. Shaw and D. Garlan. *Software Architecture: Perspectives of an Emerging Discipline.* Prentice-Hall, 1996.

[114] C. Stirling. *Modal and Temporal Properties of Processes.* Texts in Computer Science. Springer, 2001.

[115] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[116] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *TACAS*, volume 2031 of *LNCS*, pages 113–127. Springer, 2001.

[117] Y. Usenko. *Linearization in μCRL.* PhD thesis, Eindhoven University of Technology, 2002.

[118] T. Willemse. *Semantics and Verification in Process Algebras with data and timing.* PhD thesis, Eindhoven University of Technology, 2003.

[119] H. Zantema. TORPA: Termination of rewriting proved automatically. In *RTA*, volume 3091 of *LNCS*, pages 95–104. Springer, 2004.

Summary

This thesis is composed by two parts:

Part I, On Abstraction... The first five chapters are dedicated to the study of abstract interpretation (or simply, abstraction). The automatic verification of realistic applications, such as air control management, traffic control systems, ... is limited by the complexity of the systems. Abstraction constitutes a framework for program analysis. The idea is to extract program approximations by removing uninteresting information. In order to obtain approximations, computations over concrete universes of data are performed over smaller abstract domains. A typical example of this technique is the so-called "rule of signs" used to determine the sign of arithmetic expressions by performing the computation only over the signs of the operators, e.g., the expression $-5 * 10$ is abstracted to $neg * pos$ which preserves the sign of the result. The integration of this technique in an automatic verification framework allows to significantly reduce the complexity of the analyzed systems. The idea is to prove or disprove properties in the (small) abstract system and then to infer the satisfaction or refutation to the (large) concrete one.

Part II, On Coordination... The second part of the thesis is dedicated to the study of a shared dataspace architecture, called JavaSpaces. Real-time critical systems have to deal with severe requirements of reliability, fault-tolerance, extensibility, timeliness, efficiency, availability, ... These requirements make the task of building such systems extremely hard. A way of managing these severe requirements is to use software architectures. A software architecture is basically a description of how the different components of a system are to be composed. Shared dataspace architectures implement repositories for data elements in such a way that external components communicate by sharing information through the repositories instead of interacting directly between each other. This feature contributes to the conception of modular applications. The architecture is in charge of handling the concurrent access of the processes to the shared resources. External processes have a unified view of the shared space, although the repository may be distributed or centralised. We have modelled a formal framework to verify applications built on top of JavaSpaces and we have used it to analyse a non-trivial application. Furthermore, we have provided some guidelines to apply the results of the first part of the thesis to the second.

Samenvatting

Dit proefschrift bestaat uit twee delen:

Deel I, Over Abstractie... De eerste vijf hoofdstukken zijn gewijd aan de studie van abstracte interpretatie (of, eenvoudiger uitgedrukt, abstractie). De geautomatiseerde verificatie van realistische toepassingen, zoals luchtverkeersleidingssystemen, wordt bemoeilijkt door de complexiteit van dergelijke systemen. Abstractie ligt aan de basis van een raamwerk voor programma-analyse. Het onderliggende idee is om benaderingen te extraheren uit een programma, door middel van het verwijderen van oninteressante informatie. Om zulke benaderingen te verkrijgen, worden berekeningen voor het concrete universum van een datadomein uitgevoerd over een abstractie van dit universum. Een typisch voorbeeld van deze techniek is om het plus- of minteken van een aritmetische expressie te bepalen door de berekening alleen uit te voeren met betrekking tot de tekens van operatoren; bijvoorbeeld, de expressie $-5 * 10$ wordt geabstraheerd tot $neg * pos$, dat als uitkomst neg heeft. De integratie van abstractie in een geautomatiseerd raamwerk voor verificatie kan een aanzienlijke reductie opleveren van de complexiteit van de geanalyseerde systemen. Daardoor kunnen algoritmische technieken voor verificatie worden toegepast op grote systemen. Het idee is om eigenschappen te bewijzen voor (kleine) geabstraheerde systemen, en vervolgens aan te tonen dat een dergelijke eigenschap dan ook geldt voor het (grote) concrete systeem.

Deel II, Over Coordinatie... Het tweede gedeelte van dit proefschrift is gewijd aan de studie van een architectuur met een enkele, gedeelde dataruimte, JavaSpaces genaamd. Systemen in bijvoorbeeld lucht- en ruimtevaart moeten voldoen aan strenge eisen wat betreft betrouwbaarheid, fout-tolerantie, uitbreidbaarheid, tijdigheid, efficiëntie, beschikbaarheid, ... Deze eisen maken het buitengewoon moeilijk om dergelijke systemen te construeren. Een manier om met deze eisen om te gaan is het gebruik van software-architecturen. Een software-architectuur bestaat in feite uit een beschrijving hoe de verschillende componenten van een systeem dienen te worden samengesteld. JavaSpaces implementeert opslagplaatsen voor data-elementen, en externe componenten communiceren indirect met elkaar door middel van het delen van informatie via deze bewaarplaatsen. Dit kenmerk draagt bij aan de begripsvorming wat betreft modulair ontwerp. De architectuur zorgt voor de afhandeling van gelijktijdige toegang van processen tot gedeelde middelen. Externe processen hebben een uniforme kijk op de gedeelde dataruimte, ongeacht of de ruimte gedistribueerd of gecentraliseerd is. Wij hebben een formeel raamwerk gemodelleerd om toepassingen de verifiëren die bovenop JavaSpaces zijn gebouwd, en we hebben dit gebruikt om een niet-triviale toepassing te analyseren. Verder hebben we enkele richtlijnen opgesteld hoe de resultaten uit het eerste gedeelte van dit proefschrift kunnen worden toegepast in het tweede gedeelte.

Titles in the IPA Dissertation Series

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14