IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Specification and Analysis of Systems with Dynamic Structure

PhD Program in Computer Science and Engineering

XXV Cycle

By

Andrea Vandin

2012

The dissertation of Andrea Vandin is approved.

Program Coordinator: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Dr. Alberto Lluch Lafuente, IMT Institute for Advanced Studies, Lucca

Supervisor: Prof. Fabio Gadducci, University of Pisa

Tutor: Dr. Alberto Lluch Lafuente, IMT Institute for Advanced Studies, Lucca

The dissertation of Andrea Vandin has been reviewed by:

Prof. Arend Rensink, University of Twente

Prof. Stefan Leue, University of Konstanz

IMT Institute for Advanced Studies, Lucca

2012

The preparation of this thesis is the last step of a fantastic experience full of challenges and satisfaction which began on March 2010. It has been a great privilege to spend three years at IMT Institute for Advanced Studies, Lucca.

First of all I must thank Alberto Lluch Lafuente, my co-supervisor, but above all a great friend whose constant support has been critical to my growth. The same shall be said about Fabio Gadducci, who has supervised me since I started working on my Master thesis.

Many other people shall be thanked, comprising Ugo Montanari and Rocco De Nicola, respectively, the former and current coordinator of the PhD Program in Computer Science and Engineering at IMT, together with the academic and administrative members of IMT, the co-authors with whom I had the pleasure to work, and the members of the European project ASCENS.

Last but not least, a special thanks goes to my family, to Lavinia and to her family, that were always by my side.

To conclude, I quote a phrase already used in my Master thesis: "Graduation is not a point of arrival but a point of departure".

Contents

List of Figures **x**

Acknowledgements **xiii**

Vita, Publications and Presentations **xiv**

Abstract **xix**

1 Introduction **1**
1.1 Contribution 8
1.2 Structure of the thesis 14

I Technical contribution **16**

2 Counterpart semantics for quantified modal logics **18**
2.1 Running example 19
2.2 Introducing counterpart models 20
2.2.1 Many-sorted algebras 20
2.2.2 Labelled transition systems with state structure . . 22
2.3 Syntax of a quantified μ-calculus 24
2.4 Counterpart semantics 27
2.4.1 Sets of assignments 27
2.4.2 The semantic model 30
2.5 Semantics at work 34
2.6 Monotony and decidability results 39

2.6.1 Monotony 40
2.6.2 Decidability of model checking for finite models . . 42

3 Approximating the behaviour of structured systems 45
3.1 Running example 46
3.2 Counterpart model approximations 46
3.3 Reductions for counterpart models 49

4 Sound approximated model checking of infinite-state systems 53
4.1 Preservation and reflection of formulae 54
4.2 Approximated semantics and model checking 58
4.3 Dealing with untyped formulae 63
4.4 Soundness proofs 65

II Tool support 71

5 A gentle introduction to rewriting logic and Maude 74
5.1 Informal discussion 74
5.2 Detailed discussion 76

6 An explicit-state counterpart model checker for finite models 82
6.1 System specification 83
6.2 Counterpart model generation 91
6.3 Formulae evaluation 92

7 Tool support for c-reductions 94

8 Model checker at work 106
8.1 Leader election system 106
8.2 Dining philosophers with disposable forks 120

III Closing part 137

9 Discussion 138
9.1 Quantified modal logics 139

9.2 Techniques and tools for the verification of visual specification formalisms 143

10 Conclusions **153**

References **156**

List of Figures

1.1 A model with a component i in a state and none in the other 5
1.2 Schematic unfolded transition t from state s to state s' . . . 7
1.3 Two models: an infinite-state one (left) and a finite one (right) 11

2.1 An execution of an *erroneous* leader survivor algorithm . . 19
2.2 Three graphs: $\mathbf{G}_0$ (left), $\mathbf{G}_1$ (middle) and $\mathbf{G}_2$ (right) 21
2.3 A counterpart model with three sequential worlds 24

3.1 An infinite-state counterpart model 46
3.2 Approximations 48
3.3 A model (left) and its c-reduction (right) 52

4.1 A formula (ψ) preserved by a simulation (R). 54
4.2 Over-approximated semantics. 59
4.3 Approximated semantics 60

8.1 State-space sizes (left) and time necessary for their generation (right) at the varying of system size 110
8.2 Reduced state-space sizes (left) and time necessary (right) at the varying of system size 114
8.3 State-space sizes (left) and time necessary (right) at the varying of system size, reducing with `compactNames` . . 115
8.4 An illustration of the dining philosophers problem 121
8.5 State-space sizes (left) and time necessary for their generation (right) at the varying of system size 132

Listings

6.1 The Maude code to define the graph signature 84
6.2 The Maude code to define the initial state of the leader election system 86
6.3 The Maude code to define the dynamics of the leader election system 87
6.4 The global rule which applies the system-specific local rules 90
7.1 An enriched global rule to support `c`-reductions 96
7.2 The Maude code to exploit `compactNames` (1) 98
7.3 The Maude code to exploit `compactNames` (2) 99
7.4 The Maude code to exploit `freeTransps` (1) 102
7.5 The Maude code to exploit `freeTransps` (2) 103
7.6 The implementation of the canonizer `freeTransps` 104
8.1 The code to exploit the composition of `freeTransps` with `compactNames` 113
8.2 The counterpart model for four processes reduced with canonizer `freeTranspsocompactNames` 116
8.3 The Maude code to connect the counterpart model generator with the model checker 117
8.4 Evaluating formulae against the model of Listing 8.2 118
8.5 The Maude code to define the signature of the dining philosophers 125
8.6 The Maude code to define the initial state of the dining philosophers system 127

8.7 The Maude code to specify the dynamics of the dining philosophers system 129
8.8 The Maude code to exploit `freeTransps rotations` and `compactNames`(1) 134
8.9 The Maude code to exploit `freeTransps rotations` and `compactNames`(2) 135

Acknowledgements

As discussed in the Introduction, part of the material presented in this thesis has been previously published in several co-authored papers.

In particular, Chapter 2 is based on (GLV10) and (GLV12a), a joint work with Fabio Gadducci, University of Pisa, and Alberto Lluch Lafuente, IMT Institute for Advanced Studies, Lucca. Chapters 3 and 4 are instead based on (GLV12b), co-authored by Fabio Gadducci and Alberto Lluch Lafuente, and on (LMV12), co-authored by Alberto Lluch Lafuente, and José Meseguer, University of Illinois, Urbana Champaign. Finally, Part II is in part based on (LV11), co-authored by Alberto Lluch Lafuente.

Vita

December 07, 1984	Born, La Spezia, Italy.
September 2003 - December 2006	Bachelor Degree in Computer Science, University of Pisa, Italy, Final mark 110/110 cum laude.
September 2005 - January 2006	Visiting Student, Queen Mary College of London, UK, Granted by the Erasmus program.
January 2007 - October 2009	Master Degree in Computer Science, University of Pisa, Italy, Final mark 110/110 cum laude.
December 2009 - March 2010	Designer of software systems, ION Trading.
March 2010 - Now	PhD Student, IMT Lucca, Italy.
March 2010 - Now	Active member of the European FP7-ICT Integrated Project 257414 ASCENS.
January - June 2012	Visiting Student and Teaching Assistant, University of Leicester, UK.
June 2012	Mentor, AWASS2012 Awareness Summer School, Edinburgh Napier University, UK.

Publications

Journal papers:

1. Counterpart semantics for a second-order mu-calculus, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, Fundamenta Informaticae, volume 118, pages 177-205, ISSN 0169-2968, 2012;

Conference papers:

2. State Space c-Reductions of Concurrent Systems in Rewriting Logic, Alberto Lluch Lafuente, José Meseguer, Andrea Vandin, 14th International Conference on Formal Engineering Methods (ICFEM'12), Springer LNCS, volume 7635, 2012;
3. Exploiting over- and under-approximations for infinite-state counterpart models, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, 6th International Conference on Graph Transformation (ICGT'12), Springer LNCS, volume 7562, 2012;
4. A Conceptual Framework for Adaptation, Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12), Springer LNCS, volume 7212, 2012;
5. Counterpart semantics for a second-order mu-calculus, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, 5th International Conference on Graph Transformation (ICGT'10), Springer LNCS, volume 6372, 2012;

Workshop papers:

6. Adaptable Transition Systems, Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, to appear in the post-proceedings of the 21st International Workshop on Algebraic Development Techniques (WADT'12), Springer LNCS;
7. Modeling and analyzing adaptive self-assembling strategies with Maude, Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, 9th International Workshop on Rewriting Logic and its Applications (WRLA'12), Springer LNCS, volume 7571, 2012;
8. Towards a Maude Tool for Model Checking Temporal Graph Properties, Alberto Lluch Lafuente, Andrea Vandin, 10th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'11), EASST ECEASST, volume 41, 2011;

Extended abstracts:

9. Towards the Specification and Verification of Modal Properties for structured systems, Andrea Vandin, Doctoral Symposium of ICGT'12, Springer LNCS, volume 7562, 2012;
10. A Lewisian Approach to the Verification of Adaptive Systems, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, Another world is possible (Conference on David Lewis), Rivista Italiana di Filosofia Analitica Junior, volume 2, number 2, ISSN 2037-4445, 2011;
11. On a Counterpart Semantics for predicate modal mu-Calculus, Fabio Gadducci, Alberto Lluch Lafuente, Andrea Vandin, 12th Italian Conference on Theoretical Computer Science (ICTCS'10).

Presentations

Invited talks:

1. Specification and Analysis of Systems with Dynamic Structure, ETH Zurich, CH, September 2012;
2. Self-Assembling Strategies, case study mentor at AWASS2012 Awareness Summer School, Edinburgh Napier University, UK, June 2012;
3. Towards the Specification and Analysis of Systems with Dynamic Structure, PhD Seminar Series, University of Leicester, UK, December 2011.

Conference talks:

4. State Space C-Reductions of Concurrent Systems in Rewriting Logic, 14th International Conference on Formal Engineering Methods Kyoto Research Park, Japan, November 2012;
5. Specification and Analysis of Systems with Dynamic Structure, Doctoral Symposium of the 6th International Conference on Graph Transformation, University of Bremen, Germany, September 2012;
6. Exploiting Over- and Under-Approximations for Infinite-State Counterpart Models, 6th International Conference on Graph Transformation, University of Bremen, Germany, September 2012;
7. Modelling and Analyzing Adaptive Self-Assembling Strategies in Maude, 9th International Workshop on Rewriting Logic and its Applications, RWTH AACHEN University, Tallin, Estonia, March 2012;
8. A Lewisian Approach to the Verification of Adaptive Systems, Another World is Possible (Conference on David Lewis), University of Urbino Carlo Bo, Italy, June 2011;
9. Towards a Maude Tool for Model Checking Temporal Graph Properties, 10th International Workshop on Graph Transformation and Visual Modeling Techniques, Saarland University, Saarbrucken, Germany, April 2011;
10. Counterpart Semantics for a Second-Order mu-calculus, 5th International Conference on Graph Transformation, University of Twente, Netherlands, October 2010;
11. Counterpart Semantics for a Second-Order mu-calculus, 12th Italian Conference on Theoretical Computer Science, University of Camerino, Italy, September 2010.

Other talks:

12. Modelling and Analyzing Adaptive Self-Assembling Strategies in Maude, 6th meeting of the EU project ASCENS, University of Florence, Italy, March 2012;
13. Status of the integration of Maude in the SDE, 6th meeting of the EU project ASCENS, University of Florence, Italy, March 2012;
14. On Graph-Based Verification of Evolving Structures, 5th meeting of the EU project ASCENS, Verimag - CTL, Grenoble, France, July 2011.

Abstract

In many areas of Computer Science we face systems with dynamic structure, i.e. where components and resources may dynamically join and leave, or even get combined. When analyzing these systems one is not only interested in properties about global behaviours (e.g. correctness or safety), but also about the evolution of single components or of their interrelations. In order to achieve this, logic-based specification languages and techniques equipped with a neat handling of components and their dynamicity are needed.

Our solution belongs to the family of quantified μ-calculi, i.e. languages combining quantifiers with the fix-point and modal operators of temporal logics, for which we propose a novel approach to their semantics. We use counterpart models as semantic domain, i.e. labeled transition systems where states are algebras, and transitions are enriched with counterpart relations (partial morphisms) between states, explicitly encoding the evolution of components. Formulae are interpreted as sets of assignments, associating formula variables to state components. Our proposal allows to deal with the peculiarities of the considered systems, and avoids limitations of existing approaches, often enforcing restrictions of the transition relation.

Unfortunately, dynamicity may easily lead to infinite-state systems, paradigmatic examples being those with unbounded creation of components. Verification can become intractable, calling for approximation techniques. In this direction, we propose a general notion of model approximation, and exploit it by resorting to a type system which denotes formulae as reflected or preserved, together with an approximated model checking technique based on sets of approximations.

Chapter 1

Introduction

The increasing complexity of software systems is making more and more necessary the development of automatic techniques for their analysis. Commonly, in order to reason about a software system we first define an abstract mathematical model of it, where only relevant informations and properties are highlighted. Then, different techniques can be applied to *analyze* such models, one of the most successful of which is model checking (CGP99). When analyzing a system with model checking, we actually ask if a given *property* holds in its model. In order to answer to this question, we exhaustively explore all the possible executions of a system, that is the graph of the states reachable in its model starting from a given initial state.

Depending on the kind of systems to be analyzed, and on the properties to be checked, different kinds of models and property specification languages exist, providing different levels of abstraction. In the models used with traditional (i.e. propositional) model checking, the internal structure of system states is abstracted away, replaced by sets of boolean propositions which are used to provide informations about states. The property specification mechanisms commonly used in propositional model checking are propositional temporal logics, a family of rigorous, compact, expressive and natural languages. On the one hand these languages allow to reason about system states by expressing boolean formulae, com-

posed using state propositions as building blocks. On the other hand, those languages allow to reason about system evolutions, thanks to temporal operators which can be used to express formulae regarding the value of propositions in some future states.

Thanks to these abstractions, propositional model checking is very successful in managing closed systems with a fixed number of components, for example hardware circuits, communication protocols with a fixed number of partners, and programs without dynamic resource allocation (e.g. generation of new objects). However, in this thesis we focus on systems with dynamic structure, that is systems whose components and their interrelations may vary over time. This kind of systems can be found in many areas of Computer Science, like service-oriented and cloud computing, or in multi-agent environments, where system components and resources may dynamically join and leave the system, or even get combined during its evolution. Other examples are dynamic architectures and workflows, as well as dynamic data structures like the heap for object oriented languages where objects can be dynamically created.

In such dynamic scenarios, where components are created or deleted, and relations between them are established or broken, one is not only interested in proving properties of global behaviours, like e.g. system correctness (*the system will never reach a given bad state*) or safety (*the system will always reach a given good state*). In fact, in those scenarios one may be interested in reasoning about finer properties on the evolution of single components, or of groups of them. For example, one could be interested in verifying if *in a certain state there exists a particular component (e.g. a process) that will lead the system to a particular bad or good behaviour* (a sort of *individual liveness* property). Similarly, we could search for *sets of unrelated components which will establish some kind of relation (e.g. a collaboration) in the future*. Moreover, we can be interested on checking that stored messages are consumed with given policies, e.g. FIFO for queues or LIFO for stacks.

Propositional model checking is not enough to perform this kind of analysis, because it is not possible to keep track of the evolution of single components, as the internal structure of system states is abstracted away. Richer models and non propositional logics are necessary. For

what regards the models, a solution is to enrich states with an algebraic structure, explicitly representing state components and the interrelations between them. An example is graph transformation (Roz97; EEKR99; EEPT06), where graphs are used to represent the internal structure of each state. Moreover, since we are interested in reasoning about the evolution of single components and of the inner topology of each state, we also need a mechanism to identify components in different states.

For what regards the logics, (i) we need to be able to reason about the components of a state, and we can do this thanks to variables and quantifiers, (ii) we need to be able to reason about the evolution of a system, and for this we can exploit temporal operators, and finally, (iii) we need to be able to reason about the evolution of components across the evolution of the system. This last requirement can be achieved by allowing to freely mix the use of quantifiers and temporal operators, so that we can select components in a state (with quantifiers), and then study their evolution (thanks to temporal operators). Hence we need quantified temporal logics, which are equipped with a finer handling for individual components, together with specific techniques taking into account dynamicity. Such logics have been proposed by several authors in the context of software analysis, for example to reason about the life of components (e.g. (DRK02)), allowing to answer to questions like *has a component been allocated or deallocated?*, or *has it been created in the current state?*

However, before analyzing a system it is necessary to define an abstract mathematical representation of it. Several system specification formalisms have been proposed in the literature. Visual specification formalisms, in particular, are nowadays widely used in almost the whole spectrum of software and hardware development activities. In the particular case of analysis and verification activities, visual specifications are complemented with appropriate property specification languages and tools for checking and verifying properties. Actually, it can be considered as a common sense remark that any assessment on the usability of a (visual) formalism for the specification of software systems should rely on the existence of languages for expressing properties, as well as on the availability of tools

for their verification.

A prominent example of visual specification formalism supported by suitable property specification languages and tools are graph transformation systems (GTSs), temporal graph logics and the corresponding verification tools, which are used to reason about the possible transformations in a graph topology. Considering the area of graph transformation, a state of a software system is represented by a graph of some flavour (e.g. directed, edge-labeled and/or node-labelled), and each of its elements, namely nodes and edges, usually bears some relevance for the system. System dynamics are instead specified as sets of (graph) transformation rules. The application of a rule to a graph representing a state s generates a new graph representing a state s' successor of s, in which some elements of s may have been deleted, merged, renamed or new ones may have been created.

For the specific area of graphs, the issue of defining a suitable property specification language has been strongly advocated in the seminal research of Courcelle on Monadic Second-Order logic for graphs (MSO) (CE12; Cou90; Cou89). After these works, suitable variants of graph logics have been investigated and their connection with topological properties of graphs thoroughly investigated (e.g. (Cou97; DGG07)).

Coming to the area of graph transformation, the need to reason about the possible changes in a graph topology has successively led to the idea of combining temporal and graph logics. Before that, many authors studied decidability and complexity of first-order temporal logics, developed for reasoning about the evolution of individual components within a software system. Unfortunately, such logics are in general not decidable (see e.g. (FT03; HWZ01) and the references therein). As a consequence, many efforts have been devoted to the definition of logics (or the identification of fragments) that sacrifice expressiveness in favour of computability and efficiency, thus providing verification tools where logics become effective specification mechanisms.

The first approaches to combine temporal and graph logics consisted in propositional temporal logics whose state observations were limited to pure graph formulae. The impossibility to interleave the graphical and

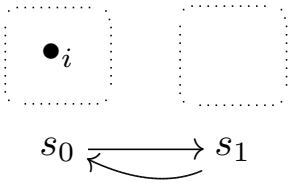

Figure 1.1: A model with a component i in a state and none in the other

temporal dimensions was thus forbidding to reason about the evolution of individual components within a graph. More recent approaches (BCKL07) propose variants of quantified μ-calculi, a combination of the fix-point and modal operators of temporal logics with the Monadic Second-Order one. Albeit less expressive than full second-order proposals, because the class of admissible predicates is restricted to first-order equality and set membership, these logics fit at the right level of abstraction for graph transformation: if state systems are graphs, and state components are thus graph elements, one is not only interested in the topological structure of each reachable graph alone, but on the evolution and fate of its individual elements as well.

Unfortunately, the semantical models for such logics are not clearly cut. Consider for instance the simple model shown in Figure 1.1 with two states s_0 and s_1, a transition from s_0 to s_1, a transition from s_1 to s_0, and an element i that appears in s_0 only. Is the *same* element i being destroyed and (re)created again and again at every iteration of the loop? Or is i just an identifier that is being reused to represent newly created elements?

From a philosophical point of view, the issue is denoted in the literature as the *trans-world identity* problem (see (Haz04) as well as (Bel06) for a survey of the related philosophical issues). From a practitioner point of view, a typical solution follows the so-called "Kripke semantics" approach. Roughly, a set of universal (graph) elements is chosen, and its elements are used to form each state. It is then obvious how it is possible to refer to the same element across states, and in fact elements are syntactically identified across different states. This solution is the most widely adopted, and it underlines most of the proposals we are aware of (as they

are surveyed in Chapter 9). For example, it is implicitly used also in the approach discussed in (BCKL07).

However, Kripke-like solutions are not perfectly suited to reason on the evolution of system components, for systems where they can be dynamically allocated, deallocated, reallocated, renamed and merged. Consider again the example of Figure 1.1. The problem is that element i belongs to the universal domain, and hence it is exactly the same i after every deallocation. But, intuitively, every *instance* of i should instead be considered to be distinct, even if syntactically equivalent. Similar considerations can be done for the renaming or merging of elements. In fact, Kripke-like structures are not suitable to deal with the possibility that elements might be merged: if each element is universal, how does one account for the coalescing of two of them?

Those problems are often solved by restricting the class of admissible evolution relations, so to avoid name reusing after deallocation, renaming and merging of elements, but in this case we would not be able to model those classes of systems. For example, in (DRK02; Ren06a; MP05a) it is forbidden to merge components. Alternatively, as done e.g. in (BCKL07), those problems are dealt by considering infinite universal domains and by unfolding transition systems so to ensure uniqueness of element identifiers. Intuitively, cycles in the transition systems are replaced with transitions towards new states where newly created elements have fresh names, thus giving rise to infinite transition systems.

Unfortunately, such solutions tend to hamper either the usability of the transformation technique in the first case, or the intuitive meaning of the logic in the second.

Consider again the case of (BCKL07), and two states s and s' of an unfolded transition system connected by a transition t as depicted in Figure 1.2. Then the relation between the elements of s and s' consists in an injective partial inclusion, meaning that elements may get deallocated by t (depicted in Figure 1.2 as the area inside the square with label "d", which appears only in s), while others (with fresh names) can be created (depicted in Figure 1.2 as the area inside the hexagon with label "n", which appears only in s'). The preserved elements have instead to remain

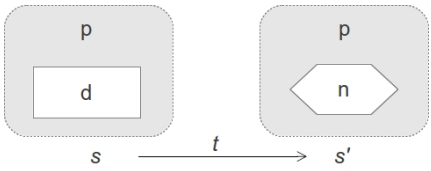

Figure 1.2: Schematic unfolded transition t from state s to state s'

unchanged (depicted in Figure 1.2 as the grey area with label "p" which appears both in s and s'). Moreover, names of deallocated elements cannot be reused in any of the successor states of s'.

The above mentioned limitations of Kripke-like structures in presence of merging or (de/re)allocation are worsened when it comes to defining the semantics of logics with recursive operators, such as quantified μ-calculi. While it is obvious that a closed formula should be evaluated as the set of states where it holds, consider instead the open formula $\mu Z.(p(x) \vee \Box Z)$. Once the value of x is chosen in a state, how is such value passed to the states denoted by the fix-point variable Z? In general, similar problems arise during the evaluation of formulae with temporal operators and free (i.e. unquantified) variables, where those variables will refer to elements of the different states met during the evaluation of the temporal operators.

Alternative solutions to Kripke models are the ones based on Lewis's *Counterpart Theory* (Lew68). Those proposals exploit *counterpart relations*, namely (partial) functions among states, explicitly relating elements of different states, which are in principle all different. Here element names have meanings local to single states, in the sense that two elements sharing the same name and residing in different states do not necessarily represent the same element.

Thanks to this intuition we are able to avoid the unique domain and the trans-world identity. The above mentioned limitations of Kripke-like models are thus mitigated.

As a matter of fact, similar approaches have been considered by various authors (e.g. (DRK02; YRSW06)) as a semantic domain for temporal

logics to reason about the evolution of individual components of a software system. Our contribution belongs to this tradition.

1.1 Contribution

In this section we summarize the contribution presented in the thesis.

Counterpart-like semantics. In (GLV10) we introduced a novel semantics for quantified μ-calculi, based on the counterpart paradigm, and we instantiated our proposal by considering a simple second-order syntax reminiscent of (BCKL07).

In our models, named *counterpart models*, system states are worlds labelled with algebras of a signature that is fixed for the whole model, and the evolution relation is given by a family of partial morphisms between accessible worlds. Intuitively, those morphisms represent our notion of counterpart relations. Clearly, since there exist several variants of many-sorted signatures for graphs, we obtain worlds labelled with graphs as particular case. However, our framework is more general, in the sense that we can choose any many-sorted signature.

More importantly, and surely one of the most original components of our work, open formulae are interpreted over sets of pairs (w, σ), for w a world of the considered model, and σ a substitution associating the variables occurring in the formula to individual elements of the (algebra labelling) the world w. In other words, a pair (w, σ) belongs to the evaluation of a formula if the formula is satisfied in the world w under the variable assignment σ mapping the free variables of the formula to elements of the algebra labelling w.

The choice of a counterpart-based semantics allows for the creation (allocation), deallocation, renaming and merging of elements by the counterpart relations labelling the transitions. Considering two worlds w, w', and a transition t from w to w', then t is labelled with a partial morphism cr representing the counterpart relation from the elements of w to the ones of w' following transition t. Intuitively, the elements of w for which cr is undefined are deallocated by t, while the elements of w' not in the image

of cr are newly created. The counterpart relation cr does not necessarily need to be injective, so that elements of w can be merged by it.

Finally, it may be worth to mention that our models faithfully take into account also the presence of cycles in the evolution relation, even in presence of name reusing, thus dispensing with the reformulation of the evolution relation (e.g. unfolding of the transition system).

In (GLV12a) we extended our original proposal in two main directions. The first research thread concerns expressiveness. Our original proposal aimed at providing a logic with a streamlined and intuitive semantics. However, it adopted a purely counterpart solution, discarding in the evaluation of the modal operator those worlds where elements previously assigned to a variable got deallocated. In other terms, it essentially represented a logic to reason about the evolution of living entities, even if their deallocation might be somehow modelled via the introduction of *ad-hoc* constants in each world.

The solution proposed in (GLV12a), and here discussed in Chapter 2, dispenses with the use of ad-hoc constants. Instead, substitutions belonging to the evaluation of an open formula might now be partial with respect to the set of (first-order) variables occurring free in the formula. Partiality is then interpreted as the deallocation of a resource explicitly addressed by the formula. This solution slightly departs from the canonical counterpart paradigm, yet it falls in line with the standard practice of graph logics.

In the rest of the thesis we will consider the semantics of (GLV12a). The interested reader can easily understand the differences between the two proposed semantics by comparing the revised set of properties discussed in Section 2.5 with the original presentation in (GLV10, Section 5).

The second line of inquiry considered in (GLV12a) concerns the decidability of our logic. Summing up, Proposition 2.2 here reported in Section 2.6, and whose proof is in Section 2.6.2, states that the validity of a formula may be established against a counterpart model as long as its set of worlds is finite, and each world is in turn labelled with a finite algebra.

Notice that this restriction is less severe than it may seem. For instance, visual specification languages are usually concerned with finite structures, equipped with an upper bound to the number of elements that is linked

with the complexity of the initial state. More in general, a wide class of software systems are *resource-bounded*, meaning that the number of elements in each of their states is bounded by some constant. This enables the use of automated verification via name-reuse techniques. These state-space reduction techniques are based on a sort of garbage collection mechanism of identifiers that guarantees finiteness of the state-space of resource-bounded systems.

As we discuss in the next paragraph, and in Chapters 3 and 4, this is an important aspect of our proposal, since in (GLV12b) and (LMV12) we enriched it with state-space reduction techniques which always lead to finite models for the particular case of resource-bounded systems, together with a framework based on counterpart model approximations that can be applied also to other kinds of systems. Then, the necessary requirements (finite worlds with algebras of finite dimension) for decidability seem quite convenient.

State-space reductions and approximations. As mentioned in the previous paragraph, in (LMV12) and (GLV12b) we enriched our framework with state-space reduction techniques which allow us to obtain finite *behaviourally equivalent* models for the particular case of resource-bounded systems, together with a general framework to exploit counterpart model approximations.

Those contributions, discussed in detail in Chapters 3 and 4, are fundamental components of our framework, since software systems with dynamically evolving components often have huge or infinite state-spaces even for very simple cases. For such systems, verification can become intractable, thus calling for the development of approximation techniques that may ease the verification at the cost of losing in preciseness and completeness.

Consider the trivial but infinite-state system modelled in the left of Figure 1.3. The initial state s_0 is composed by only one component u_0, which is deallocated in the transition towards s_1 to be replaced by a new component u_1. The system then computes an infinite sequence of steps in which the component residing in the current state is deallocated to be

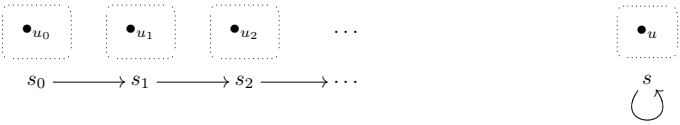

Figure 1.3: Two models: an infinite-state one (left) and a finite one (right)

replaced by a new component which will be deallocated in the following transition. Looking at the right part of Figure 1.3, we instead see a system with only one state s containing a component u, and whose transition relation is composed by only a self-loop on s.

If we consider the model in Figure 1.3 (right) as a Kripke model, then we would have an (idle) system which evolves without affecting its own state, that is preserving its only component u. Considering instead the model in Figure 1.3 (right) as a counterpart model whose transition is labelled with an empty counterpart relation, then we would have that at each step u is deallocated, and it is replaced by a new distinct one (even if sharing the same name). Hence, in this second case, it is intuitive and easy to see that the two models of Figure 1.3 express the *same behaviours*.

Notice that adopting Kripke models, due to the trans-world identity problem, it is not possible to finitely represent the above mentioned model of Figure 1.3 (right). A sort of state-space unfolding is necessary. Moreover the components have to be uniquely identified by their names in all the states of the model, hence each state has to explicitly contain different names for different components, forbidding name reusing. Intuitively, the model in Figure 1.3 (left) would be obtained.

More in detail, in (LMV12) we proposed a state-space reduction technique for Kripke models capturing symmetry reduction (WD10), name reusing, and name abstraction. The technique is based on the idea of canonical reductions, abbreviated in c-reductions, as a generic means to reduce a Kripke model K by exploiting some equivalence relation $\sim$ on the states of K which is also a bisimulation on K (i.e. between K and itself). Then, the concept of state canonizers is introduced, that is functions mapping states in the (possibly not unique) representative of their equivalence class, modulo an equivalence which is also a bisimula-

tion (e.g. a canonical permutation of names of processes with identical behavior).

As a consequence, a model reduced exploiting a canonizer is bisimilar to the original one.

Moreover, c-reductions easily allow to obtain an on-the-fly reduction technique by applying canonizers to each newly generated state, directly during the generation of the state-space. Intuitively, we do not require to build the original state-space to reduce it, but we instead directly generate the reduced one. This allows to apply our reduction also to infinite-state models. In Chapter 3 we lift this proposal to counterpart models.

In (GLV12b) we proposed a general formalization of counterpart model approximations based on standard behavioural preorders for transitions systems (i.e. similarities) which we extended to counterpart models.

We consider *under-approximations*, namely models that express *less behaviours* than the original one, and *over-approximations*, namely models that express *more behaviours* than the original one. Intuitively, considering three models $\underline{M}$, M and $\overline{M}$, we say that $\underline{M}$ is an under-approximation of M if $\underline{M}$ is similar to M, because M *simulates* $\underline{M}$. Conversely, we say that $\overline{M}$ is an over-approximation of M if M is similar to $\overline{M}$, because $\overline{M}$ simulates M. In the particular case in which a model is bisimilar to the original one, then it is both an under- and an over-approximation, and it represents exactly the same behaviours of the original one, as it is intuitively for the two models of Figure 1.3.

Building on the concepts of under- and over-approximation, we then proposed a sound approximated model checking procedure based on a partial type system that types formulae as preserved or reflected by an approximation. The procedure approximates the evaluation of formulae in a model M, exploiting sets of its under- and over-approximations.

It is worth to note that our type system, which generalizes the one presented in (BCKL07), becomes complete for bisimilar models, in which case it types every formula as strongly preserved, that is preserved and reflected. In particular, it can be proved that our logic characterizes our notion of bisimilarity, meaning that we can freely analyze models reduced up-to bisimilarity.

This contribution is discussed in detail in Section 3.2 and in Chapter 4.

Tool support. In (LV11) we instantiated our approach implementing a prototypal explicit-state model checker for our logic. We implemented it in Maude (CDE+07), a high-performance formal language and execution environment based on equational and rewriting logic.

The tool has been implemented to test our semantics and its feasibility, and has to be intended as our first step towards the development of a tool framework supporting our proposal. Part II details and exemplifies our efforts in this direction.

Currently, the tool supports both the original semantics proposed in (GLV10) and the one proposed in (GLV12a). And, in particular, it allows to specify systems, to automatically generate a counterpart model starting from a system specification, and to check properties of our second-order μ-calculus against finite counterpart models.

The tool is parametric with respect to the signature Σ of the (algebras assigned to the) worlds of the models, representing the internal structure of the states of the systems. In order to fix the signature, the user has only to list its sorts and operation names. Once the signature Σ is fixed, systems are specified by an initial state (i.e. a Σ-algebra), and by a set of behavioural rules specifying the dynamics.

More recently, as detailed in Chapter 7 and exemplified in Chapter 8, we integrated our c-reductions approach (Section 3.3) in our prototypal model checker. As discussed, c-reductions allow to reduce counterpart models in behaviourally equivalent (i.e. bisimilar) ones. Interestingly, many infinite counterpart models, and in particular the class of resource-bounded ones, can be reduced to finite counterpart models. This, together with the fact that every formula of our logic is preserved and reflected in bisimilar models (Chapters 3 and 4), allows to exploit our model checker to analyze possibly infinite-state systems by checking formulae against their c-reductions, provided suitable state canonizers.

Summing up, this thesis proposes a novel approach to the semantics of quantified μ-calculi which solves problems of some existing proposals,

and that is well suited to reason about the evolution of system components.

Then we complement the logic with a general formalization of model approximations, with a sound approximated model checking technique for infinite-state systems, and with a state-space reduction technique to reduce systems in bisimilar ones with smaller state-spaces.

By exploiting our framework we can approximate the evaluation of formulae in infinite-state systems by resorting to under- and over-approximations, that is similar systems representing, respectively, less and more behaviours. As particular case we have that the logic preserves our notion of bisimilarity, meaning that we can always reason with models minimized up-to it. Interestingly, we can always reduce infinite-state resource bounded systems to bisimilar finite-state state-spaces, allowing for their (not approximated) analysis.

In this thesis we defined and studied a general framework for the analysis of systems with dynamically evolving structure, based on a very expressive logic. This opens other possibly future interesting lines of research, like the definition of proper model checking algorithms, or of less expressive logics encodable in the presented one, so that we will be able to reuse all the proposals and results presented in this thesis.

1.2 Structure of the thesis

The thesis is structured in an introduction (this section) followed by three parts.

Part I. This part discusses in detail our technical contribution. Chapter 2 introduces our novel approach to the semantics of quantified μ-calculi. Chapter 3 first presents a general notion of counterpart model approximation based on behavioural preorders, and then proposes a state-space reduction technique based on state canonizers which captures symmetry reduction and name reusing. Finally, in Chapter 4 we study how the semantics of our logic is related to counterpart model approximations, and

basing on this we propose a sound approximated model checking procedure which exploits sets of approximations to estimate the evaluation of a formula in a model.

Part II. This part discusses our efforts in developing a tool framework to support our proposal. Chapter 5 introduces rewriting logic (Mes12), and its instantiation in Maude (CDE+07), on which our tool framework is based. Clearly, this chapter does not represent a contribution of our thesis, however it is necessary to make more accessible the following chapters. Chapter 6 presents a prototypal model checker which evaluates formulae of our logic against finite-state counterpart models, while Chapter 7 discusses how we extended it to implement our c-reductions approach. Finally, Chapter 8 validates our tool against some interesting systems.

Part III. The thesis is concluded by a closing part. Chapter 9 technically discusses how our contribution is related to existing proposals, while Chapter 10 contains the conclusive remarks and future works.

Part I

Technical contribution

This part of the thesis discusses the technical contributions.

Chapter 2 introduces our novel approach to the semantics of quantified μ-calculi, instantiating it to a simple second-order syntax.

Chapter 3 proposes a general formalization of counterpart model approximation, together with a state-space reduction technique to obtain behaviourally equivalent (i.e. bisimilar) models.

Chapter 4 complements Chapter 3 by providing a sound approximated model checking procedure exploiting sets of over- and under-approximations of a counterpart model.

Chapter 2

Counterpart semantics for quantified modal logics

Quantified μ-calculi combine the fix-point and modal operators of temporal logics (such as μ-calculus, CTL, LTL) with (existential and universal) quantifiers, and they allow to reason about the possible behaviour of individual components within a software system.

In this chapter we introduce our novel approach to the semantics of such calculi: we consider as semantic models a sort of labelled transition systems called *counterpart models*, inspired by Graph Transition Systems (BCKL07) and Lewis Counterpart Theory (Lew68). In counterpart models, states are algebras and transitions are defined by counterpart relations (a family of partial morphisms) between states.

Formulae are interpreted over sets of state assignments, namely families of partial substitutions, associating formula variables to state components.

Our proposal allows us to model and reason about the creation and deletion of components, as well as about their merging. Moreover, it avoids the limitations of existing approaches, usually enforcing restrictions of the transition relation: the resulting semantics is a streamlined and intuitively appealing one, yet it is general enough to cover most of the alternative proposals we are aware of. The chapter concludes with

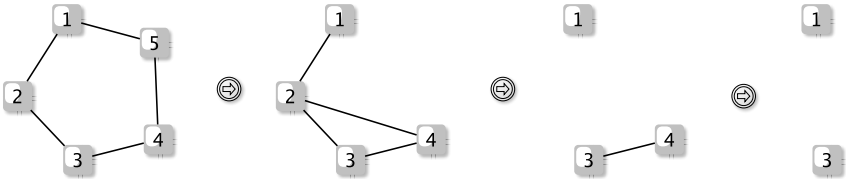

Figure 2.1: An execution of an *erroneous* leader survivor algorithm

some considerations about expressiveness and decidability aspects.

2.1 Running example

For a better illustration of our concerns, we will use a simple running example throughout this chapter.

The example is inspired by the classical circle elimination games such as the *chair dance*, *bang bang* and the *electric chair*. It consists of a sort of distributed leader *survivor* game, in which a set of processes is connected through communication ports forming a ring topology. The game evolves performing a series of elimination rounds. After each round the loser is eliminated from the game: it abandons the ring and its neighbours should be connected so to close the ring again. The game ends when there is only one process left: the winner (leader, survivor) of the game.

We actually abstract away from the algorithms used in each round, and focus instead on the evolution of the communication topology. Figure 2.1 illustrates a possible execution of some *wrong* algorithm for the leader survivor game. The initial state (left) consists of five participants. The first round eliminates participant 5 in an ill-manner: instead of closing the circle again, a shortcut between 2 and 4 is introduced. The system evolves until the end state (right) where two processes (1 and 3) remain and claim to be the winners. This is a typical situation one might want to characterize and avoid.

As a matter of fact, depending on the algorithm used in each round, the evolution of the topology might exhibit different properties that one

might want to check. We enumerate a set of properties that will be used throughout this chapter:

p1: *"Will a leader be elected in all possible executions?"*;

p2: *"Can two distinct leaders be elected in the same execution?"*;

p3: *"Is there a process that necessarily becomes the leader?"*;

p4: *"In which states do we have a leader?"*;

p5: *"For which processes does an execution leading to its election exist?"*;

p6: *"Which communication ports will eventually merge?"*;

p7: *"Are processes connections correctly updated after each round?"*.

In Section 2.5 we will see how to express formulae stating these and other properties, and how they are evaluated against a model of one possible game instance. In Chapter 8 we will instead see how these properties are evaluated using a prototypal model checker that we developed for our logic.

2.2 Introducing counterpart models

In this section we define the class of models over which our logic is interpreted.

2.2.1 Many-sorted algebras

We begin by recalling the definition of many-sorted algebras and their (possibly partial) morphisms, which lies at the basis of the structure of our worlds.

Definition 2.1 (Many-sorted algebra) *A* many-sorted signature Σ *is a pair* (S_Σ, F_Σ) *composed by a set of sorts* $S_\Sigma = \{\tau_1, \cdots, \tau_m\}$ *and by a set of function symbols* $F_\Sigma = \{f_\Sigma : \tau_1 \times \ldots \times \tau_n \to \tau \mid \tau_i, \tau \in S_\Sigma\}$ *typed over* S^*_Σ. *A* many-sorted algebra $\boldsymbol{A}$ *with signature* Σ *(a* Σ*-algebra) is a pair* (A, F^A_Σ) *such that*

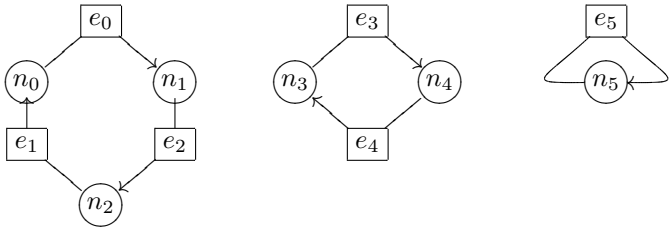

Figure 2.2: Three graphs: $\mathbf{G}_0$ (left), $\mathbf{G}_1$ (middle) and $\mathbf{G}_2$ (right)

- *the* carrier A *is a set of elements typed over* S_Σ*;*
- $F_\Sigma^{\boldsymbol{A}} = \{f_\Sigma^{\boldsymbol{A}} : A_{\tau_1} \times \ldots \times A_{\tau_n} \to A_\tau \mid f_\Sigma : \tau_1 \times \ldots \times \tau_n \to \tau \in F_\Sigma\}$ *is a family of functions on* A *typed over* S_Σ^**.*

Where $A_\tau = \{a \in A \mid a : \tau\}$ *is the subset of* τ*-typed elements of* A*, and to each typed function symbol* $f_\Sigma \in F_\Sigma$*, called fundamental operations of* $\boldsymbol{A}$*, corresponds a function* $f_\Sigma^{\boldsymbol{A}}$ *in* $F_\Sigma^{\boldsymbol{A}}$*.*

Given two Σ-algebras $\mathbf{A}$ and $\mathbf{B}$, a *(partial) morphism* ϱ is a family of (possibly partial) functions $\{\varrho_\tau : A_\tau \rightharpoonup B_\tau \mid \tau \in S_\Sigma\}$ typed over S_Σ (mapping elements of sort $\tau \in A$ in elements of the same sort in B), such that for each typed function symbol $f_\Sigma : \tau_1 \times \ldots \times \tau_n \to \tau \in F_\Sigma$ and list of elements $a_1, \ldots, a_n$, if each function ϱ_{τ_i} is defined for the element a_i of type τ_i, then ϱ_τ is defined for the element $f_\Sigma^{\mathbf{A}}(a_1, \ldots, a_n)$ of type τ and the elements $\varrho_\tau(f_\Sigma^{\mathbf{A}}(a_1, \ldots, a_n))$ and $f_\Sigma^{\mathbf{B}}(\varrho_{\tau_1}(a_1), \ldots, \varrho_{\tau_n}(a_n))$ coincide.

Note that our morphisms can be partial, possibly decreasing the domain of definition of a function and thus modeling the removal of world elements.

Example 2.1 (Graph algebra) *As an instance of many-sorted algebras we adopt a very simple unary algebra, the one for ordinary directed graphs. More precisely, the signature for directed graphs is* (S_{Gr}, F_{Gr})*. The set* S_{Gr} *consists of the sort of nodes* τ_N *and the sort of edges* τ_E*, while the set* F_{Gr} *is composed by the function symbols* $s : \tau_E \to \tau_N$ *and* $t : \tau_E \to \tau_N$ *which determine, respectively, the source and the target node of an edge.*

In Figure 2.2 we find the visual representations for three graphs: $\boldsymbol{G}_0$*,* $\boldsymbol{G}_1$*, and* $\boldsymbol{G}_2$*. The first of these graph algebras is given by* $\boldsymbol{G}_0 = (N_0 \uplus E_0, \{s^{\boldsymbol{G}_0}, t^{\boldsymbol{G}_0}\})$*,*

where $N_0 = \{n_0, n_1, n_2\}$, $E_0 = \{e_0, e_1, e_2\}$, $s^{\boldsymbol{G}_0} = \{e_0 \mapsto n_0, e_1 \mapsto n_2, e_2 \mapsto n_1\}$ *and* $t^{\boldsymbol{G}_0} = \{e_0 \mapsto n_1, e_1 \mapsto n_0, e_2 \mapsto n_2\}$. *We omit the definitions of* $\boldsymbol{G}_1$ *and* $\boldsymbol{G}_2$, *since these are already clear from their visual representations.*

As we will see in Example 2.2, each graph can be understood as the communication topology of a state of our running example (Section 2.1). Thus, edges represent processes participating in the leader survivor game, source and target functions denote the communication ports, respectively, of the processes, and nodes are the communication channels (port attachments).

We will make use of terms with variables. For this purpose we take into account signatures Σ_X obtained by extending a many-sorted signature Σ with a denumerable set X of variables typed over S_Σ. We let X_τ denote the τ-typed subset of variables and with x_τ or $x : \tau$ a variable with sort τ. Similarly, we let ϵ_τ or $\epsilon : \tau$ indicate a τ-sorted term.

Definition 2.2 (Terms) *Let* Σ *be a signature, let* X *be a (denumerable) set of individual variables typed over* S_Σ, *and let* Σ_X *denote the signature obtained extending* Σ *with* X. *The (many-sorted) set* $T(\Sigma_X)$ *of* terms *obtained from* Σ_X *is the smallest set such that*

$$\frac{}{X \subseteq T(\Sigma_X)} \qquad \frac{f : \tau_1 \times \ldots \times \tau_n \to \tau \in F_\Sigma,\ \forall i \in [1, n].\ \epsilon_i : \tau_i \in T(\Sigma_X)}{f(\epsilon_1, \ldots, \epsilon_n) : \tau\ \in\ T(\Sigma_X)}$$

We omit the sort when it is clear from the context or when it is not necessary.

2.2.2 Labelled transition systems with state structure

We now introduce the notion of *counterpart model*. The origin of the name is due to the "Counterpart Theory" (Lew68) of David Lewis, where entities reside in just one world and have explicit counterparts in other worlds, as opposed to the trans-world identity of Kripkean models.

Definition 2.3 (Counterpart model) *Let* Σ *be a signature, and* $\mathcal{A}$ *the set of algebras over* Σ. *A* counterpart model M *is a triple* $(W, \rightsquigarrow, d)$ *such that*

- W *is a set of worlds;*
- $d : W \to \mathcal{A}$ *is a function assigning to each world* $w \in W$ *a* Σ*-algebra;*

- $\rightsquigarrow \subseteq (W \times (\mathcal{A} \rightharpoonup \mathcal{A}) \times W)$ *is the* accessibility relation *over* W*, enriched with (partial) morphisms (*counterpart relations*) between the algebras of the connected worlds.*

The elements of $\rightsquigarrow$ are defined such that, for every $(w_1, cr, w_2) \in \rightsquigarrow$, we have that $cr : d(w_1) \to d(w_2)$ is a (partial) morphism. In particular, each component "cr" of $\rightsquigarrow$ explicitly defines the counterparts in (the algebras assigned to) the target world of (the algebras assigned to) the source world. In the following we may use $w_1 \overset{cr}{\rightsquigarrow} w_2$ for $(w_1, cr, w_2) \in \rightsquigarrow$.

In standard terminology, we are considering a transition system labeled with morphisms between algebras, as an immediate generalization of *graph transition systems* (BCKL07).

The counterpart relations allow to avoid the *trans-world identity*, i.e. the implicit identification of elements of (the algebras of) different worlds sharing the same identifier, meaning that two elements with the same identifier but residing in different worlds does not necessarily represent the same entity. As a consequence, the identifiers of the elements have a meaning that is local to the world where they reside. For this reason, as we will see, the counterpart relations allow for the creation, deletion, renaming and merging of elements in a type-respecting way. Duplication is forbidden, because no cr can associate an element of $d(w_1)$ to more than one element of $d(w_2)$.

Should Σ be a signature for graphs, a counterpart model is visually a two-level hierarchical graph: at the higher level the nodes are the worlds $w \in W$, and the edges are the evolution steps labeled with the associated counterpart relation; at the lower level, each world w contains an algebra over Σ, hence a graph, which represents its internal structure.

In the rest of the thesis we will consider counterpart models without deadlock worlds, i.e. worlds without outgoing transitions. This is not a limitation since a counterpart model not satisfying this condition can be transformed into one satisfying it by introducing self-loops (w, id_w, w) to $\rightsquigarrow$ for each w without outgoing transitions. In particular id_w represents the identity-preserving counterpart relation. This is a standard technique in Model Checking to define the semantics of temporal logics.

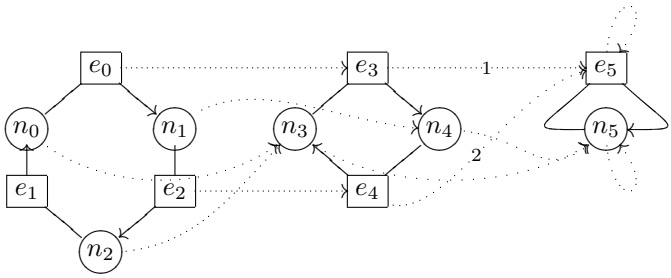

Figure 2.3: A counterpart model with three sequential worlds (w_0, w_1, w_2)

Example 2.2 *The counterpart model in Figure 2.3 illustrates two possible executions of our running example (Section 2.1) instantiated with three processes (edges). The model is made of three worlds, namely w_0, w_1, and w_2, that are mapped into the graph algebras $\boldsymbol{G}_0$, $\boldsymbol{G}_1$, and $\boldsymbol{G}_2$ of Figure 2.2, respectively. The counterpart relations (drawn with dotted lines, possibly numbered in order to remove any ambiguity) reflect the fact that at each transition one process (edge) is discarded and its source and target channels (nodes) are merged: e_1 from w_0 to w_1, and either e_3 or e_4 from w_1 to w_2. In particular the model contains three transitions: one from w_0 to w_1 and two from w_1 to w_2. The transition (w_0, cr_0, w_1) deletes edge e_1 and merges nodes n_0 and n_2 into n_3, while all the other graph items are simply renamed. Similarly for (w_1, cr_1, w_2) and (w_1, cr_2, w_2), their difference (as witnessed by the dotted arrows numbered with either 1 or 2) being the edge they decide to remove. Finally, (w_2, cr_3, w_2) is a simple cycle preserving both e_5 and n_5, denoting that the system is idle, yet alive.*

2.3 Syntax of a quantified μ-calculus

Before presenting the syntax of our logic, we introduce the notions of *second-order variables* ($Y \in \mathcal{X}$) and *fix-point variables* ($Z \in \mathcal{Z}$). Formally, a variable of the second-order Y_τ with sort $\tau \in S_\Sigma$ ranges over sets of elements of sort τ. Given a counterpart model M (Definition 2.7), we will see that an assignment σ_w, defined for a world w in M, associates variables of first- and of second-order to elements and to sets of elements, respectively, of the algebra $d(w)$ underlying w. Fix-point variables will range over the set of pairs (w, σ_w), for w a world and σ_w an assignment for w. Whenever clear from the context, the subscript world w may be

omitted from the assignment.

Definition 2.4 (Quantified modal formulae) *Let Σ be a signature, $\mathcal{Z}$ a set of fix-point variables, and X, $\mathcal{X}$ (denumerable) sets of first- and second-order variables typed over S_Σ, respectively. The set $\mathcal{F}_\Sigma$ of formulae of our logic is generated by the rules*

$$\psi ::= \ tt \mid \epsilon \in_\tau Y \mid \neg\psi \mid \psi \vee \psi \mid \exists_\tau x.\psi \mid \exists_\tau Y.\psi \mid \Diamond\psi \mid Z \mid \mu Z.\psi$$

where $x \in X$, $Y \in \mathcal{X}$, ϵ is a term over Σ_X, $\in_\tau$ is a family of membership predicates typed over S_Σ indicating that (the evaluation of) a term with sort τ belongs to (the evaluation of) a second-order variable with the same sort τ, $\exists_\tau$ allows to quantify over elements (sets of elements for the second-order case) with sort τ, $\Diamond$ is the "possibility" one-step modality, $Z \in \mathcal{Z}$, and μ denotes the least fixed point operator.

Whenever clear from the context, subscripts and types may be omitted. In this sense, we implicitly require for terms and second-order variables appearing in predicates $\in_\tau$ to have sort τ ($\epsilon : \tau \in_\tau Y_\tau$). As it is standard, we restrict to *monotonic* formulae, i.e. such that each fix-point variable Z occurs under the scope of an even number of negations. This is a necessary condition for fixed points to be well-defined.

An equivalence operator. Note that the logic is simple, yet reasonably expressive. Other than the standard boolean connectives $\wedge$, $\rightarrow$, $\leftrightarrow$, and the universal quantifiers $\forall_\tau$, we can derive other useful operators. For instance "$=_\tau$", the family of binary equivalence operators typed over S_Σ, stating the equality of terms with sort τ, can be derived as $\epsilon_1 =_\tau \epsilon_2 \ \equiv \forall_\tau Y.\ (\epsilon_1 \in_\tau Y \leftrightarrow \epsilon_2 \in_\tau Y)$.

Some temporal operators. We can derive as usual the greatest fix-point operator "ν" as $\nu Z.\psi \equiv \neg\mu Z.\neg\psi$, and the "necessarily" one-step modality $\Box$ as $\Box\psi \equiv \neg\Diamond\neg\psi$ *(ψ holds in all the next one-steps)*.

Moreover, in the next sections we use the standard universal temporal operators AG, AF, and AU: they are derived as $AG\psi \equiv \nu Z.(\psi \wedge \Box Z)$ *(for all departing paths, ψ always holds)*, $AF\psi \equiv \mu Z.(\psi \vee \Box Z)$ *(for all departing*

paths, ψ eventually holds), and $A(\psi_1 U \psi_2) \equiv \mu Z.[\psi_2 \vee (\psi_1 \wedge \Box Z)]$ *(for all departing paths, ψ_1 holds at least until a state is eventually met where ψ_2 holds)*.

Finally, we also use the standard existential temporal operators EG, EF, and EU: they are derived as $EG\psi \equiv \nu Z.(\psi \wedge \Diamond Z)$ *(there exists a departing path such that ψ holds always)*, $EF\psi \equiv \mu Z.(\psi \vee \Diamond Z)$ *(there exists a departing path such that ψ eventually holds)*, and $E(\psi_1 U \psi_2) \equiv \mu Z.[\psi_2 \vee (\psi_1 \wedge \Diamond Z)]$ *(there exists a departing path such that ψ_1 holds at least until a state is met where ψ_2 holds)*. The assumption in Definition 2.3 grants that all the paths are infinite.

Example 2.3 (Some properties) *Consider again the running example, the graph signature, the counterpart model of Figure 2.3, and the typed predicates $\boldsymbol{present}_\tau(x) \equiv \exists_\tau y.x =_\tau y$ expressing the presence of an entity with sort τ in a world (the typing is usually omitted). Moreover, consider the predicate $\boldsymbol{leader}(x) \equiv s(x) = t(x) \wedge \boldsymbol{present}(x)$ characterizing leader processes.*

The following formulae, omitting the typing informations, express different properties of our running example: $\psi_1 \equiv \mu Z.[\exists x.(\boldsymbol{leader}(x)) \vee \Box Z]$, also expressed as $AF[\exists x.(\boldsymbol{leader}(x))]$, states that for all departing paths from a world, eventually there will be a leader *(property **p1** of Section 2.1); while formula $\psi_2 \equiv \exists x.[\mu Z.(\boldsymbol{leader}(x) \vee \Box Z)]$, expressed also as $\exists x.[AF\boldsymbol{leader}(x)]$, states that (in a world)* there is a process that, for all departing paths, will eventually become the leader *(property **p3** of Section 2.1).*

Intuitively, ψ_1 is satisfied by those worlds w such that, for each path departing from w, a world is reached where a leader (self-closed edge) is present. Instead, ψ_2 is satisfied by those worlds w which contain a process (edge) that is a leader (i.e. its source *and* target *ports coincide) or that will become a leader in a world reachable after some finite number of steps following any path departing from w.*

Formula ψ_2 has thus quite a different meaning than ψ_1: in ψ_2 the subformula $AF(\boldsymbol{leader}(x))$ is inside the scope of the existential quantifier which fixes the element associated to x (the potential leader) in the source world to keep track of its evolution.

*Finally, $\psi_3 \equiv \mu Z'.(\psi_2 \vee \Box Z')$, also expressed as $AF(\psi_2)$, represents another particular flavour of property **p3** (and a variant of property **p1** of Section 2.1), stating that* for any execution, eventually there will be a state containing a process that will become the leader. *This property is satisfied by those worlds w that (for any possible execution) reach a world where there is a process that is the leader, or that will later on evolve in the leader following any departing path.*

In Example 2.5 we will see how these three properties are evaluated against the counterpart model of Figure 2.3. In particular, in that counterpart model,

ψ_1 and ψ_3 actually coincide, but this does not happen in general (e.g. the two formulae may differ in scenarios with participants joining the game dynamically).

We now introduce the notion of *context*, used for decorating formulae with relevant variable-related informations. For the sake of simplicity, in the rest of the paper we fix a signature Σ and denumerable sets $X, \mathcal{X}, \mathcal{Z}$ of first-order, second-order, and fix-point variables, respectively.

Definition 2.5 (First- and second-order context) *A* first-order context Γ *is a finite subset of X, while a* second-order context Δ *of $\mathcal{X}$.*

For a first-order context Γ and a variable x not contained in Γ, we write Γ, x to indicate $\Gamma \cup \{x\}$ and $\Gamma \setminus x$ to indicate $\Gamma \setminus \{x\}$. Similarly for second-order contexts and variables.

Our semantics does not evaluate naked formulae, but formulae-in-context, that is formulae decorated with informations about their free variables. This requirement is going to be pivotal in the definition of the semantics, as it is shown in Section 2.4.

Definition 2.6 (Formula-in-context) *A* formula-in-context *is defined as $\psi[\Gamma; \Delta]$, where ψ is a formula in $\mathcal{F}_\Sigma$, Γ is a first-order context and Δ is a second-order context, containing, respectively, at least the free first- and second-order variables of ψ.*

2.4 Counterpart semantics

In this section we introduce the semantic domain of our logic, together with the rules for evaluating a formula-in-context as a set of assignments on a counterpart model. Once more, for the sake of simplicity, in the rest of the thesis we fix a counterpart model M.

2.4.1 Sets of assignments

Our first step is to define the semantic domain of our formulae, which are sets of assignments.

Definition 2.7 (Assignments) *A* (variable) assignment $\sigma_w = (\sigma^1_w, \sigma^2_w)$ *for a world* $w \in M$ *is a pair of partial functions typed over* S_Σ *such that* $\sigma^1_w : X \rightharpoonup d(w)$ *and* $\sigma^2_w : \mathcal{X} \rightharpoonup 2^{d(w)}$.

Now, let Ω_M *denote the set of pairs* (w, σ_w)*, for* σ_w *an assignment over the world* w*. A* (fix-point variable) assignment *is a partial function* $\rho : \mathcal{Z} \rightharpoonup 2^{\Omega_M}$.

Given a term ϵ and an assignment $\sigma = (\sigma^1, \sigma^2)$, we freely write $\sigma(\epsilon)$ or $\sigma^1(\epsilon)$ to denote the lifting of σ^1 to the set of terms over Σ_X. Intuitively, it indicates the evaluation of the term ϵ under the assignment σ for its variables. If σ is undefined for any of the variables in ϵ, then also $\sigma(\epsilon)$ is undefined.

In the following, we write Ω whenever the referred counterpart model is clear from the context. We also denote by $\Omega^{[\Gamma;\Delta]}$ the set of pairs $(w, (\sigma^1_w, \sigma^2_w))$ such that the domain of definition of σ^1_w is contained in Γ (σ^1_w is defined for a subset of Γ), and such that the domain of definition of σ^2_w is exactly Δ. Moreover, $\Omega_w \subseteq \Omega$ denotes the subset of pairs over a world w (i.e. those pairs whose first component is the world w), and similarly for the subset $\Omega^{[\Gamma;\Delta]}_w \subseteq \Omega^{[\Gamma;\Delta]}$. As we will see, the evaluation function of our formulae-in-context is strongly based on the definitions of $\Omega^{[\Gamma;\Delta]}_w$ and $\Omega^{[\Gamma;\Delta]}$.

Note the asymmetry in the definition: an assignment σ may be undefined over the elements of Γ, yet not over those of Δ: intuitively, $\sigma(x)$ may be undefined if the element it was denoting has been deallocated, while we can always assign the empty set to $\sigma(Y)$, should all its elements be removed. We hence use partial first-order assignments to treat item deallocations.

The definition below addresses the need of either extending or restricting the domain of definition of (a set of) assignments.

Definition 2.8 (Restrictions and extensions) *Let* $[\Gamma; \Delta]$ *be a context, and* x *a first-order variable not in* Γ*. Given an assignment* $\sigma = (\sigma^1, \sigma^2)$*, such that* $(w, \sigma) \in \Omega^{[\Gamma,x;\Delta]}$*, its* restriction $\sigma \downarrow_x$ *is the assignment* $(\sigma^1 \downarrow_x, \sigma^2)$*, such that* $(w, \sigma \downarrow_x) \in \Omega^{[\Gamma;\Delta]}$*, obtained by removing* x *from the domain of definition of* σ^1.

Vice versa, let $a \in d(w)$ *be an element of the world* w*. Given an assignment* $\sigma = (\sigma^1, \sigma^2)$*, such that* $(w, \sigma) \in \Omega^{[\Gamma;\Delta]}_w$*, its* extension $\sigma[^a/_x]$ *is the assignment* $(\sigma^1[^a/_x], \sigma^2)$*, such that* $(w, \sigma[^a/_x]) \in \Omega^{[\Gamma,x;\Delta]}_w$*, obtained by extending the do-*

main of definition of σ^1 with the first-order variable x by assigning the element a to it.

The notation above (as well as the one introduced below) is analogously and implicitly given also for second-order variables.

Definition 2.9 (Powerset lifting) *Let $[\Gamma;\Delta]$ be a context and $x \notin \Gamma$ a first-order variable. The function $2^{\downarrow_x} : 2^{\Omega^{[\Gamma,x;\Delta]}} \to 2^{\Omega^{[\Gamma;\Delta]}}$ lifts $\downarrow_x$ to sets of pairs.*

Vice versa, let $\uparrow_x \colon \Omega^{[\Gamma;\Delta]} \to 2^{\Omega^{[\Gamma,x;\Delta]}}$ be the function mapping each $(w,\sigma) \in \Omega_w^{[\Gamma;\Delta]}$ to the set $\{(w, \sigma[^a/_x]) \mid a \in d(w)\} \subseteq \Omega_w^{[\Gamma,x;\Delta]}$. The function $2^{\uparrow_x} : 2^{\Omega^{[\Gamma;\Delta]}} \to 2^{\Omega^{[\Gamma,x;\Delta]}}$ lifts $\uparrow_x$ to sets.

Intuitively, by extending the set $\Omega^{[\Gamma;\Delta]}$ with respect to a variable $x_\tau \notin \Gamma$, we extend the assignment of every pair $(w, \sigma_w) \in \Omega^{[\Gamma;\Delta]}$ denoting the variable x_τ with all elements of sort τ in w. Note that the extensions may shrink the set of assignments, should the algebra associated to the world have no element of the correct type. In general terms, given a variable $x_\tau \notin \Gamma$ and a pair (w,σ), the cardinality of $2^{\uparrow_{x_\tau}}(\{(w,\sigma)\})$ is the cardinality of $d(w)_\tau$, i.e. the cardinality of the set of elements of type τ in $d(w)$.

The corresponding functions $\downarrow_Y$, $2^{\downarrow_Y}$, $\uparrow_Y$, and $2^{\uparrow_Y}$, with respect to a second-order variable $Y \notin \Delta$, are defined in the same way.

Example 2.4 (Assignments) *Consider the counterpart model of Figure 2.3, and let us denote by $\lambda = (\lambda_1, \lambda_2)$ the empty assignment, regardless of the world. Then, each set $\Omega_{w_i}^{[\emptyset;\emptyset]}$ simply corresponds to $\{(w_i, \lambda)\}$, and consequently $\Omega^{[\emptyset;\emptyset]}$ corresponds to $\{(w_0,\lambda),(w_1,\lambda),(w_2,\lambda)\}$. If we extend $\Omega_{w_1}^{[\emptyset;\emptyset]}$ including the first-order variable x with sort τ_E, we obtain*

$$2^{\uparrow_x}(\Omega_{w_1}^{[\emptyset;\emptyset]}) = \big\{\big(w_1, (\lambda_1[^{e_3}/_x], \lambda_2)\big), \big(w_1, (\lambda_1[^{e_4}/_x], \lambda_2)\big)\big\}$$

which is in turn equal to $\big\{\big(w_1, (\{x \mapsto e_3\}, \lambda_2)\big), \big\{\big(w_1, (\{x \mapsto e_4\}, \lambda_2)\big)\big\}$.

As a final step, we define when two assignments are compatible with a counterpart relation.

Definition 2.10 (Counterpart assignment) *Let $M = (W, \rightsquigarrow, d)$, $(w, cr, w') \in \rightsquigarrow$, and σ_w an assignment for w. Then the counterpart assignment of σ_w relatively to cr is the assignment $\sigma_{w'}$ (for w') obtained applying cr to the components of σ_w, denoted as $cr \circ \sigma_w$, that is $\sigma^1_{w'} = cr \circ \sigma^1_w$, and $\sigma^2_{w'} = 2^{cr} \circ \sigma^2_w$, for 2^{cr} the lifting of cr to sets.*

In the following, we may use $\sigma_w \overset{cr}{\rightsquigarrow} \sigma_{w'}$ to indicate that $\sigma_{w'}$ is the counterpart assignment of σ_w relatively to cr. Moreover, given any (possibly partial) morphism $\phi : d(w) \to d'(w')$, we use $\phi \circ \sigma_w$ to denote the assignment (for w') obtained applying ϕ to the components of σ_w.

It is interesting to understand which cases may arise for the first-order. Consider a first-order variable x in Γ. Intuitively, should $\sigma_w(x)$ be undefined, we want $\sigma_{w'}(x)$ to be undefined as well: the meaning is that if $\sigma_w(x)$ refers to an element deallocated in w, then we want $\sigma_{w'}(x)$ to represent an element deallocated in w'. Should $\sigma_w(x)$ be defined, but $cr(\sigma_w(x))$ undefined, then $\sigma_w(x)$ has been deallocated in the evolution from w to w' through the considered transition, hence we want $\sigma_{w'}(x)$ to be undefined. Whenever both $\sigma_w(x)$ and $cr(\sigma_w(x))$ are defined, we want $\sigma_w(x)$ to evolve in $\sigma_{w'}(x)$ accordingly to cr.

Considering the second-order case, for each variable $Y \in \Delta$, the elements in $\sigma_w(Y)$ that are preserved by cr are mapped in $\sigma_{w'}(Y)$. Hence if $\sigma_w(Y)$ is defined, then $\sigma_{w'}(Y)$ is also defined, with a cardinality equal or smaller to the one of $\sigma_w(Y)$, due to the fusion or deletion of elements in $\sigma_w(Y)$ induced by cr.

2.4.2 The semantic model

We are now ready to introduce the semantic evaluation of our logic in a model M. The evaluation of formulae is a mapping of formulae-in-context $\psi[\Gamma;\Delta]$ into sets of pairs contained in $\Omega^{[\Gamma;\Delta]}$. Hence, the domain of the assignments in these pairs must be, respectively, contained in Γ for the first-order component, and exactly equal to Δ for the second-order component. Intuitively, a pair (w, σ) belongs to the semantics of a formula-in-context $\psi[\Gamma;\Delta]$ if the formula-in-context holds in w under the assignment σ for its free variables. For the sake of presentation, we assume that all the bound variables are different among themselves, and from the free ones.

Definition 2.11 (Semantics) *Let $\psi[\Gamma;\Delta]$ be a formula-in-context. Let x and Y be a first- and second-order variable, and let Z be a fix-point variable. The evaluation of a formula $\psi[\Gamma;\Delta]$ in M under assignment $\rho : \mathcal{Z} \to 2^{\Omega^{[\Gamma;\Delta]}}$ is*

given by the function $[\![\cdot]\!]^M_\rho : \mathcal{F}^{[\Gamma;\Delta]} \to \Omega^{[\Gamma;\Delta]}$

$$
\begin{array}{rcl}
[\![tt[\Gamma;\Delta]]\!]^M_\rho & = & \Omega^{[\Gamma;\Delta]}_M \\
[\![(\epsilon \in_\tau Y)[\Gamma;\Delta]]\!]^M_\rho & = & \{(w,\sigma_w) \in \Omega^{[\Gamma;\Delta]}_M \mid \sigma_w(\epsilon) \textit{ is defined and } \sigma_w(\epsilon) \in \sigma_w(Y)\} \\
[\![\neg\psi[\Gamma;\Delta]]\!]^M_\rho & = & \Omega^{[\Gamma;\Delta]}_M \setminus [\![\psi[\Gamma;\Delta]]\!]^M_\rho \\
[\![\psi_1 \vee \psi_2[\Gamma;\Delta]]\!]^M_\rho & = & [\![\psi_1[\Gamma;\Delta]]\!]^M_\rho \cup [\![\psi_2[\Gamma;\Delta]]\!]^M_\rho \\
[\![\exists_\tau x.\, \psi[\Gamma;\Delta]]\!]^M_\rho & = & 2^{\downarrow_x}(\{(w,\sigma_w) \in [\![\psi[\Gamma,x;\Delta]]\!]^M_{(2^{\uparrow_x} \circ \rho)} \mid \sigma_w(x) \textit{ is defined}\}) \\
[\![\exists_\tau Y.\, \psi[\Gamma;\Delta]]\!]^M_\rho & = & 2^{\downarrow_Y}([\![\psi[\Gamma;\Delta,Y]]\!]^M_{(2^{\uparrow_Y} \circ \rho)}) \\
[\![\Diamond\psi[\Gamma;\Delta]]\!]^M_\rho & = & \{(w,\sigma_w) \in \Omega^{[\Gamma;\Delta]}_M \mid \exists w \overset{cr}{\rightsquigarrow} w'.\, (w', cr \circ \sigma_w) \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho\} \\
[\![Z[\Gamma;\Delta]]\!]^M_\rho & = & \rho(Z) \\
[\![\mu Z.\psi[\Gamma;\Delta]]\!]^M_\rho & = & \mathit{lfp}(\lambda Y.[\![\psi[\Gamma;\Delta]]\!]^M_{\rho[^Y/_Z]})
\end{array}
$$

We now comment in detail each single equation defining the evaluation function of our formulae. We omit M.

Trivial cases. The formula-in-context $tt[\Gamma;\Delta]$ holds in any pair definable for M given the context $[\Gamma;\Delta]$. The predicate $\in_\tau$ regards the membership of (the evaluation of) a term $\epsilon : \tau$ to (the evaluation of) a second-order variable with the same sort. Hence $\epsilon \in_\tau Y[\Gamma;\Delta]$ is satisfied by those pairs (w,σ) such that $\sigma(\epsilon)$ is defined, and belongs to $\sigma(\chi)$. A formula with $\neg\psi$ is satisfied by those pairs not satisfying ψ, while the disjunction $\psi_1 \vee \psi_2$ of two formulae is evaluated as the union of the pairs satisfying the two disjuncts ψ_1 and ψ_2 (keeping the same context). The evaluation of $Z[\Gamma;\Delta]$ and $\mu Z.\psi[\Gamma;\Delta]$ follows the standard definitions.

Quantifiers. More interesting is the case of formulae with a quantifier as main operator. Evaluating such formulae one is usually interested in finding an item (a set of items for the second-order case) that, if assigned to the quantified variable, would satisfy the subformula where the variable is replaced by the value determined by the assignment. We follow the same idea: considering $\exists_\tau x.\psi[\Gamma;\Delta]$, we first evaluate ψ extending the context with x (which occurs free in ψ). The resulting set of pairs is hence included in $\Omega^{[\Gamma,x;\Delta]}$. Dropping those pairs whose assignment is not defined for x, we obtain exactly the set of pairs whose worlds contain an item that assigned to x satisfies ψ. The last step is the restriction of x in the assignments of the filtered set of pairs. The restricted set thus contains pairs in $\Omega^{[\Gamma;\Delta]}$.

The second-order case is similar, except for the fact that we do not have to check for the assignments to be defined for the second-order variables, since (differently from the first-order case) the second-order assignments are defined for all the variables in the second-order context.

Note that during the evaluation of quantifiers it is pivotal to require that also the assignment ρ is modified, in order to account for the extensions to the newly introduced variables: it allows for a proper sorting of $\rho(Z)$, since it must belong to the subsets of $\Omega^{[\Gamma,x;\Delta]}$ ($\Omega^{[\Gamma;\Delta,Y]}$ in the second-order case).

In sum, in order to give a meaningful existential semantics to the first-order case, we consider only those pairs in $[\![\psi[\Gamma, x; \Delta]]\!]_{(2^{\uparrow_x} \circ \rho)}$ whose assignments actually relate x with a concrete entity (the required *existing* one), discarding thus the pairs whose assignments are not defined for x. By doing so we grant that a pair (w, σ) belongs to the semantics of a formula $\exists_\tau x.\psi$ only if $d(w)$ actually contains at least an entity e with sort τ, such that $(w, \sigma[^e/_x])$ belongs to the semantics of ψ.

Modal operator. Another interesting case is that of formulae whose top operator is the modal one. More precisely, in the evaluation of $\Diamond\psi[\Gamma; \Delta]$ we search for pairs (w, σ_w) such that there exists an outgoing transition $w \overset{cr}{\rightsquigarrow} w'$ and an assignment $\sigma_{w'}$ for w', such that the pair $(w', \sigma_{w'})$ belongs to the evaluation of $\psi[\Gamma; \Delta]$. Moreover, in order to relate the relevant items in w with the ones in w', we require that $\sigma_{w'}$ is the counterpart assignment of σ_w relatively to cr (Definition 2.10). In other words, we require that $\sigma_{w'}$ respects the relation induced by cr between the (involved) items of the source and destination worlds. We impose this condition because items are not implicitly identified across worlds of the model, but they are instead correlated by the counterpart relations.

Finally note that the evaluation of a closed formula, i.e. of a formula $\psi[\emptyset; \emptyset]$ with an empty context, is just a set of pairs (w, λ), for λ the empty variable assignment. Hence, such an evaluation characterises a set of worlds: this ensures that our proposal properly extends the standard semantics for propositional modal logics.

In proposition 2.1, ensuring the existence of suitable solutions to fixpoints equations, we state the well-definedness of the semantics.

Example 2.5 (Evaluation of formulae-in-context) *We now exemplify the evaluation of the formulae proposed in Example 2.3 against the model of Figure 2.3, namely* $\psi_1[\emptyset;\emptyset] \equiv \mu Z.[\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z][\emptyset;\emptyset]$*, and* $\psi_2[\emptyset;\emptyset] \equiv \exists x.[\mu Z.(\textbf{\textit{leader}}(x) \vee \Box Z)][\emptyset;\emptyset]$*. We omit the typing information as done in Example 2.3. For easiness of presentation we use* $(w_{0,1,2}, \lambda)$ *to denote the set* $\{(w_0,\lambda),(w_1,\lambda),(w_2,\lambda)\}$*.*

Recall that formula ψ_1*, also written* $AF[\exists x.(\textbf{\textit{leader}}(x))]$*, states that for all departing paths from a world, eventually there will be a leader, identified by a self-closed edge (property* ***p1*** *of Section 2.1). Then, we intuitively expect that the formula holds for all worlds (paired with the empty assignment), i.e.* $(w_{0,1,2}, \lambda)$*, because* w_2 *contains the process (edge)* e_5 *which is a leader,* w_1 *evolves into* w_2*, and* w_0 *into* w_1*.*

According to the semantics, fixed an assignment ρ*, we have to evaluate the least fixed point of* $\lambda Y.[\![\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z[\emptyset;\emptyset]]\!]_{\rho[^Y/_Z]}$*. Consider* ρ *being the empty assignment.*

According to Kleene's theorem, we first evaluate the function with $Y = \emptyset$ *(i.e. with* $\rho[^Y/_Z]$ *the function* $(Z \mapsto \emptyset)$ *assigning the empty set to* Z*), i.e.* $[\![\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z\ [\emptyset;\emptyset]]\!]_{(Z\mapsto\emptyset)}$*. This clearly evaluates in* $\{(w_2,\lambda)\}$*, in fact only* w_2 *contains a self-closed edge (*e_5*), and hence only the pair* (w_2,λ) *satisfies* $\psi_x \equiv \exists x.(\textbf{\textit{leader}}(x))$*, and* $[\![\Box Z]\!]_{(Z\mapsto\emptyset)}$ *does not add any other pair.*

We now consider the case of $Y = \{(w_2,\lambda)\}$*, that is* $[\![\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z[\emptyset;\emptyset]]\!]_{(Z\mapsto\{(w_2,\lambda)\})}$*, evaluated as* $\{(w_1,\lambda),(w_2,\lambda)\}$*, because* (w_1,λ) *and* (w_2,λ) *belong to* $[\![\Box Z]\!]_{(Z\mapsto\{(w_2,\lambda)\})}$*.*

As third step we consider the function with $Y = \{(w_1,\lambda),(w_2,\lambda)\}$*, that is* $[\![\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z[\emptyset;\emptyset]]\!]_{(Z\mapsto\{(w_1,\lambda),(w_2,\lambda)\})}$*, which is this time evaluated as the set* $(w_{0,1,2},\lambda)$*, since now also* (w_0,λ) *belongs to* $[\![\Box Z]\!]_{(Z\mapsto\{(\lambda,w_1),(\lambda,w_2)\})}$*.*

Finally, we fix Y *as the set of pairs* $(w_{0,1,2},\lambda)$*. Then, it is easy to see that* $[\![\exists x.(\textbf{\textit{leader}}(x)) \vee \Box Z[\emptyset;\emptyset]]\!]_{(Z\mapsto(w_{0,1,2},\lambda))}$ *is evaluated as* $(w_{0,1,2},\lambda)$*. As expected, this means that the set of pairs* $(w_{0,1,2},\lambda)$ *is the least fixed point of the function.*

Different is the case for $\psi_2[\emptyset;\emptyset] \equiv \exists x.[\mu Z.(\textbf{\textit{leader}}(x) \vee \Box Z))][\emptyset;\emptyset]$*. Recall that the formula* ψ_2*, also written* $\exists x.[AF(\textbf{\textit{leader}}(x))]$*, states that (in a world) there exists a process (edge) that, for all departing paths, will eventually become the leader, identified by a self-closed edge (property* ***p3*** *of Section 2.1). Intuitively we expect that the formula holds only for the world* w_2 *(and empty assignment), i.e.* $\{(w_2,\lambda)\}$*, because* w_2 *contains the process (edge)* e_5 *which is a leader. The*

world w_1 contains the processes e_3, e_4, each of which will become leader (e_5) only following one of the two outgoing paths. The same reasoning can be applied to e_0 and e_2 of the world w_0.

According to the semantics, fixed an assignment ρ, we first have to evaluate the set $[\![\mu Z.(\textbf{\textit{leader}}(x) \vee \Box Z)[\{x\};\emptyset]]\!]_{(2^{\uparrow_x} \circ \rho)}$, and then to apply the function $2^{\downarrow_x}$ to those pairs belonging to the resulting set whose assignments are defined for x. Considering ρ being the empty assignment, $2^{\uparrow_x} \circ \rho$ remains the empty assignment.

Applying the semantics of the μ operator, we have to evaluate the least fixed point of the function $\lambda Y.[\![\textbf{\textit{leader}}(x) \vee \Box Z[\{x\};\emptyset]]\!]_{\emptyset[^Y/_Z]}$. Following Kleene's theorem, we first evaluate the function with $Y = \emptyset$, i.e. $[\![\textbf{\textit{leader}}(x) \vee \Box Z[\{x\};\emptyset]]\!]_{(Z \mapsto \emptyset)}$, which clearly is (considering only the pairs with assignment defined for x) $\{(w_2, (x \mapsto e_5))\}$, in fact e_5 is the only self-closed edge of the model, and $[\![\Box Z]\!]_{(Z \mapsto \emptyset)}$ does not add any other pair. Then we consider the function with $Y = \{(w_2, (x \mapsto e_5))\}$, that is $[\![\textbf{\textit{leader}}(x) \vee \Box Z[\{x\};\emptyset]]\!]_{(Z \mapsto \{(w_2,(x \mapsto e_5))\})}$, which again is evaluated as $\{(w_2, (x \mapsto e_5))\}$. This tells us that $\{(w_2, (x \mapsto e_5))\}$ is indeed the least fixed point of the function. Finally, $2^{\downarrow_x}(w_2, \{(x \mapsto e_5)\}) = \{(w_2, \lambda)\}$, as we intuitively expected.

2.5 Semantics at work

This section illustrates the use of our logic to express properties of the evolution of systems and of their components. Most examples are drawn from the literature reviewed in Chapter 9. In order to make the syntax lighter, we do not indicate the contexts of the formulae and the types of the variables and of the quantifiers.

Deallocation. The creation and destruction of entities has attracted the interest of several authors (e.g. (DRK02; YRSW06)) as a means for reasoning about the allocation, deallocation and preservation of resources or processes. Our logic does not offer an explicit mechanism for this purpose. Nevertheless, as we have shown in Example 2.3, we can easily derive the predicate **present**(x), expressing the presence of an entity in a certain world. Using this predicate in conjunction with the next-time modalities, we can reason about the preservation of entities after a transition of the sys-

tem as **possiblyPreserved**$(x) \equiv$ **present**$(x) \wedge \Diamond$**present**(x), for entities preserved by at least an outgoing transition, and **necessarilyPreserved**$(x) \equiv$ **present**$(x) \wedge \Box$**present**(x), for entities preserved by every outgoing transition. We similarly reason about the deallocation of entities after a transition as **possiblyDeallocated**$(x) \equiv$ **present**$(x) \wedge \Diamond \neg$**present**(x), for entities deallocated by at least a transition, and **necessarilyDeallocated**$(x) \equiv$ **present**$(x) \wedge \Box \neg$**present**(x), for for entities deallocated by every outgoing transition.

Moreover, if we are interested in reasoning about the preservation or deallocation of entities during a generic execution of a system, we can require that an entity is always preserved by every execution with **AgloballyPreserved**$(x) \equiv$ **present**$(x) \wedge AG$**present**(x), or by at least an execution with **EgloballyPreserved**$(x) \equiv$ **present**$(x) \wedge EG$**present**(x). Finally, we require for an entity to be eventually deallocated in at least a departing execution with **EeventuallyDeallocated**$(x) \equiv$ **present**$(x) \wedge EF \neg$**present**(x), or in all the possible departing executions with the formula **AeventuallyDeallocated**$(x) \equiv$ **present**$(x) \wedge AF \neg$**present**(x), to express properties like "*the message msg_i will be eventually delivered*", assuming messages to be deallocated only after their delivery.

In our original proposal (GLV10) we adopted a purely counterpart solution, where the evaluation of the modal operator did not consider those worlds where graph items previously assigned to a variable were deleted. In this setting, the semantics of the predicates regarding the presence and absence of entities might have not been meaningful. For instance, under the scope of the next-time modality, predicates over x should have been intended as "as long as x is present", so that formulae like $\Box$**present**(x) could have accepted assignments for x in worlds that could have evolved by deleting x. The key point of our original proposal was its use to reason about living entities and to faithfully follow Lewis Theory. However, their deallocation might have been modeled via the introduction of *ad-hoc* constants in each world. The current solution dispenses with the use of ad-hoc constants, allowing for an easier definition of these kind of predicates.

Birth, growth and preservation. When reasoning about entity creation, it is interesting to distinguish new from old entities. Our logic has no built-in mechanism (like e.g. in (DRK02)) for this purpose, yet one can assume that this information is provided by the model (by using *new* and *old* values and a function from entities into those values). Still, in the rest of this paragraph we see first how it is possible to state bounds on the number of components in a system, and then how it is possible to define modalities to capture the preservation and the creation of entities.

Our logic is well suited to express properties about the growth of a system. For instance, a growth bound of 2 is stated with **at-most-2** $\equiv \forall x.\forall y.\forall z.x = y \vee y = z \vee z = x$ as in (DRK02) and it is required as an invariant with AG **at-most-2**.

More interestingly, we can express properties along entity preserving behaviours. For instance, a modality restricted to (i.e. considering only) transitions preserving at least an item of the source state (i.e. a **one-preserved** modality) is defined as $\langle$**one-preserved**$\rangle(\psi) \equiv \exists Y.\exists x.[x \in Y \wedge \Diamond(x \in Y \wedge \psi)]$, with neither x nor Y appearing in ψ. Intuitively, the idea here is that the subformula $\Diamond(x \in Y \wedge \psi)$ can be satisfied only if at least an item of the source state is preserved. In the same line we can define a modality restricted to transitions preserving a given number n of components of the source state. For example, a modality considering transitions preserving (at least) two of the items of the source state can be defined as $\langle$**pairs-preserved**$\rangle(\psi) \equiv \exists Y.\exists x.\exists y.[x \in Y \wedge y \in Y \wedge x \neq y \wedge \Diamond(x \in Y \wedge y \in Y \wedge \psi)]$. If we are instead interested in behaviours creating at least an element, we can define the modality $\langle$**one-new**$\rangle(\psi) \equiv \forall Y.\Diamond(\exists x.x \notin Y \wedge \psi)$. Finally, an abbreviation for transitions deallocating at least an element is $\langle$**one-deallocated**$\rangle(\psi) \equiv \exists Y.\exists x.[x \in Y \wedge \Diamond(x \notin Y \wedge \psi)]$.

Intuitively, fixing the formula $\psi \equiv tt$ and the empty context, we can use these derived modalities to express behaviours regarding preservation and creation of elements. Consider $\langle$**one-preserved**$\rangle(tt)[\emptyset;\emptyset]$. Evaluating it we obtain the states from which it is possible to execute a transition preserving at least an item.

Moreover, using these derived $\Diamond$-modalities, e.g. $\langle$**one-preserved**$\rangle$ and $\langle$**one-new**$\rangle$, we can derive other standard temporal operators like

$EG_{\textbf{one-preserved}}(\psi) \equiv \nu Z.[\psi \wedge \langle \textbf{one-preserved} \rangle (Z)]$, and $EF_{\textbf{one-new}}(\psi) \equiv \mu Z.[\psi \vee \langle \textbf{one-new} \rangle (Z)]$. Intuitively, these operators are similar to the standard EG and EF ones, but consider only transitions which, respectively, preserve at least an item, and create at least a new one. Other temporal operators can be similarly defined. Fixing again the formula tt, with $EG_{\textbf{one-preserved}}(tt)$ we can now ask for those states from which (at least) a path departs where at each transition at least an element is preserved. While, more generally, with $EF_{\textbf{one-new}}(\psi)$ we ask for those states from which a path departs creating a new element after every transition, such that eventually ψ will hold.

It is also possible to define the $\Box$-versions of the above introduced $\Diamond$-modalities. For example we define $[\textbf{one-preserved}](\psi) \equiv \exists Y.\exists x.[x \in Y \wedge \Box(x \in Y \wedge \psi)]$. In this case we consider only states having an element preserved following any of the outgoing transitions.

Entity evolution. Apart from the growth in the number of entities, our logic can asses their evolution in general. A typical example is the mobility of objects (as in the message propagation example of (BCKL07)). Assuming an algebra of objects and locations with a function symbol loc denoting the location of an object, we can express location change for an object x with the predicate $\textbf{moves}(x, \psi) \equiv \exists y.[y = loc(x) \wedge \Box(loc(x) \neq y \wedge \psi)]$. Then we can express that x never remains in the same location with $\nu Z.\textbf{moves}(x, Z)$.

Along the same lines, we can define other typical individual safety and liveness properties. For instance, individual mutual exclusion (used e.g. in (YRSW06)) can be stated with formula $\nu Z.[(loc(x) \neq loc(y)) \wedge \Box Z]$, also written as $AG(loc(x) \neq loc(y))$, which requires that x and y will never be in the same location. Another example is represented by individual responsiveness properties, like requiring two entities to eventually meet whenever they are in separate locations: $\nu Z.[loc(x) \neq loc(y) \rightarrow \mu Z'.(loc(x) = loc(y) \vee \Diamond Z') \wedge \Box Z]$, or $AG[loc(x) \neq loc(y) \rightarrow EF(loc(x) = loc(y))]$.

Example 2.6 (Semantics at work on the running example) *Making use of the operators derived in this section, we can formalize the properties of our run-*

ning example enumerated in Section 2.1. We will in particular refer to their evaluation over the counterpart model of Figure 2.3. In the following examples we will consider two first-order edge and node variables, respectively x_E, y_E, and x_N, y_N, but we will omit the typing informations when clear from the context.

*We start with property **p1** of Section 2.1, i.e. the necessity of the existence of (at least) a leader. As shown in Example 2.3, considering the predicate* $\textbf{\textit{leader}}(x)$*, we express the property, intended as* for all departing paths from a world, eventually there will be a leader, *as* $AF[\exists x.(\textbf{\textit{leader}}(x))]$*. Evaluating the formula (with empty context) against the model of Figure 2.3, we obtain the set of pairs* $\{(w_0, \lambda), (w_1, \lambda), (w_2, \lambda)\}$*, with* λ *the empty context, meaning that the property is satisfied in every world of the model.*

*Next we might be interested in proving the uniqueness of the leader (**p2** of Section 2.1). In order to do so we first define a predicate denoting states with zero or one leaders (a "correct" number of leaders) as:* $\textbf{\textit{atMostOneLeader}}(x, y) \equiv [(\textbf{\textit{leader}}(x) \wedge \textbf{\textit{leader}}(y)) \rightarrow x = y]$*. In order to check that all the states of the model satisfy property **p2**, we evaluate the negation of this predicate. Evaluating the formula with context composed by the two first-order (edge) variables* x *and* y*, we obtain the expected empty set of pairs. Notice that this formula does not just tell us which are the erroneous states (i.e. states with more than a leader), but it also tells us which are those multiple leaders of the state. In fact the formula has two free variables* x *and* y*. Hence, in case of erroneous states, we would obtain variable assignments from the two variables, to the different leaders found in the state. If we are instead interested only in knowing which are the erroneous states, then we could alternatively express the property by resorting to the existential quantification:* $\exists x.\exists y.[\neg\textbf{\textit{atMostOneLeader}}(x, y)]$*. Of course, the formula (this time with empty context) is again evaluated as the empty set of pairs.*

*Property **p3** of Section 2.1, which states that (in a world)* there is a process that, for all departing paths, will eventually become the leader *is formalized with* $\exists x.[\mu Z.(\textbf{\textit{leader}}(x) \vee \Box Z)]$*, or* $\exists x.[AF(\textbf{\textit{leader}}(x))]$*. In Example 2.5 we already evaluated this formula (*ψ_2*).*

*Considering property **p4** of Section 2.1, with* $\exists x.\textbf{\textit{leader}}(x)$ *we check "*in which states do we have a leader?*". The formula (with empty context) is evaluated as a set containing only a pair composed by the empty assignment and the world* w_2*. With the formula* $\textbf{\textit{leader}}(x)$ *we instead check for "*which process of which state is a leader*" (a variant of property **p4** of Section 2.1). The formula (with context containing only* x*) is evaluated as a pair composed by the assignment mapping* x *to* e_5*, paired with the world* w_2*, which indeed is the only world with a leader.*

Another interesting formula is $EF(\textbf{\textit{leader}}(x))$*, i.e. "*for which process does there exist an execution leading to its election?*" (property **p5** of Sec-*

tion 2.1). The formula (with context composed by the variable x) is evaluated as the set $\{(w_0, x \mapsto e_0),\ (w_0, x \mapsto e_2),\ (w_1, x \mapsto e_3),\ (w_1, x \mapsto e_4),\ (w_2, x \mapsto e_5)\}$, as the only process never elected is e_1 of world w_0.

*We now move our attention to the evolution of entities with sort node, representing the communication ports between connected processes. We can for example be interested in knowing "*which ports will eventually (for any possible path) merge*" (property **p6** of Section 2.1), written $(x \neq y) \wedge AF[\textbf{\textit{present}}(x) \wedge \textbf{\textit{present}}(y) \wedge (x = y)]$. The formula (with context composed by the two node variables) is evaluated as the set $\{(w_0, (x \mapsto n_0, y \mapsto n_1)),\ (w_0, (x \mapsto n_0, y \mapsto n_2)),\ (w_0, (x \mapsto n_1, y \mapsto n_2)),\ (w_1, (x \mapsto n_3, y \mapsto n_4))\}$, plus, of course, the symmetric pairs with inverted assignments for x and y. In fact, all the nodes get merged in the node n_5 of world w_2 for any execution. To be noticed is the fact that n_1 and n_2 (and n_0) of w_0 get merged in n_5 after two transitions through n_3 and n_4 of w_1.*

*Considering the property **p7** of Section 2.1 "*are the connections between processes correctly updated after each round?*", with $\textbf{\textit{present}}(x_E) \wedge (x_N = s(x_E)) \wedge (y_N = t(x_E)) \wedge \Diamond[(\neg \textbf{\textit{present}}(x_E)) \wedge (x_N \neq y_N)]$ we search for states that can evolve deallocating a process without merging its communication ports. We hence search for configurations possibly leading to the problem mentioned in Section 2.1, and represented in Figure 2.1. Evaluating this formula over the model of Figure 2.3, with context containing the three involved first-order variables, we obtain the empty set.*

*Finally, the last property we consider regards the necessary deallocation of at least a process per transition. In particular, we check that our running example actually deallocates (at least) a process per transition, except in final states. We can express such property with the derived predicate $\textbf{\textit{allPreserved}} \equiv \forall x. \Box \textbf{\textit{present}}(x)$. By instantiating the predicate with sort edge (e.g. **allEdges Preserved**), and evaluating it with empty context against the counterpart model of Figure 2.3, we obtain the world w_2 paired with the empty assignment, stating that the property "*there exists a state which can evolve without deallocating any process*" holds just in final states.*

2.6 Monotony and decidability results

In this section we state two important properties for our logic, namely we first state that its semantics is well-defined, and then we prove that its model checking problem is decidable for finite models.

2.6.1 Monotony

The next proposition ensures the existence of suitable solutions to fix-points equations, stating the well-definedness of the semantics. In particular, we state that function $\lambda Y.[\![\psi[\Gamma;\Delta]]\!]_{\rho[Y/Z]}$ is monotonic, for ψ a formula where all occurrences of fix-point variables are positive, and any suitable Γ, Δ and ρ.

Proposition 2.1 (Monotony) *Let ψ be a (naked) formula such that all occurrences of fix-point variables are positive. Then the function $\lambda Y.[\![\psi[\Gamma;\Delta]]\!]_{\rho[Y/Z]}$ is monotonic in the lattice $(\Omega^{[\Gamma;\Delta]},\subseteq)$ for any choice of context $[\Gamma;\Delta]$ (such that $\psi[\Gamma;\Delta]$ is well-formed), of fix-point variable Z, and of assignment ρ (such that ρ is undefined on Z).*

Proposition 2.1 tells us that the restriction to formulae where all occurrences of fix-point variables are positive guarantees that any function $\lambda Y.[\![\psi[\Gamma;\Delta]]\!]_{\rho[Y/Z]}$ is monotonic in the lattice $(\Omega^{[\Gamma;\Delta]},\subseteq)$. Therefore, by Knaster-Tarski theorem, fixed points (and thus our semantics) are well-defined.

Proof: Given a formula ψ, we prove that for any two sets $\Omega_1,\Omega_2 \subseteq \Omega^{[\Gamma;\Delta]}$, we have that $\Omega_1 \subseteq \Omega_2$ implies $[\![\psi[\Gamma;\Delta]]\!]_{\rho[\Omega_1/Z]} \subseteq [\![\psi[\Gamma;\Delta]]\!]_{\rho[\Omega_2/Z]}$ for any (consistent) choice of Γ, Δ, Z and ρ. The proof is done by structural induction on ψ.

The proposition trivially holds whenever Z is not a free variable of ψ, as in the formulae tt and $\epsilon \in_\tau Y$.

[$\psi \equiv \neg\psi'$] By definition, $[\![\neg\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_i/Z]} = \Omega^{[\Gamma;\Delta]} \setminus [\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_i/Z]}$ for $i \in \{1,2\}$. Now, Z appears under an odd number of negations in ψ' (i.e. all Z-occurrences are negative). We can prove a dual proposition stating the anti-monotonicity in those cases. Mutual reference can be ruled out by structural induction on the formulae. Thus, we can assume $[\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_1/Z]} \supseteq [\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_2/Z]}$. Therefore, by the properties of set complement we obtain $\Omega^{[\Gamma;\Delta]} \setminus [\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_1/Z]} \subseteq \Omega^{[\Gamma;\Delta]} \setminus [\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_2/Z]}$. Following the definition we close the proof for this case, i.e. $[\![\neg\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_1/Z]} \subseteq [\![\neg\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_2/Z]}$.

[$\psi \equiv \psi_1 \vee \psi_2$] Let us now consider the case when ψ is $\psi_1 \vee \psi_2$. By definition, $[\![\psi_1 \vee \psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}$ is $[\![\psi_1[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]} \cup [\![\psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}$. We can then apply the induction hypothesis on both ψ_1 and ψ_2. Hence we have that $[\![\psi_1[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\psi_1[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$ and that $[\![\psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$. We can finally conclude that $[\![\psi_1 \vee \psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\psi_1 \vee \psi_2[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$.

[$\psi \equiv \exists_\tau x.\ \psi'$] Following the semantics, we evaluate the expressions $[\![\exists_\tau x.\ \psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}$ as set $2^{\downarrow_x}(\{(w,\sigma) \in [\![\psi'[\Gamma,x;\Delta]]\!]_{2^{\uparrow_x}\circ\rho[^{\Omega_i}/_Z]} \mid \sigma(x) \text{ is defined}\})$ for $i \in \{1,2\}$. We can apply the induction hypothesis on ψ' (even if context and assignment are changed): noting that also $2^{\uparrow_x}$ is monotone, we have that $[\![\psi'[\Gamma,x;\Delta]]\!]_{2^{\uparrow_x}\circ\rho[^{\Omega_1}/_Z]} \subseteq [\![\psi'[\Gamma,x;\Delta]]\!]_{2^{\uparrow_x}\circ\rho[^{\Omega_2}/_Z]}$. Finally, note that $2^{\downarrow_x}$ is monotonic as well, hence by applying the definition we obtain $[\![\exists_\tau x.\ \psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\exists_\tau x.\ \psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$.

[$\psi \equiv \exists_\tau Y.\ \psi'$] The proof is similar to the first-order case.

[$\psi \equiv \Diamond\psi'$] The expressions $[\![\Diamond\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}$ are defined as $\{(w,\sigma_w) \in \Omega^{[\Gamma;\Delta]} \mid \exists w \overset{cr}{\rightsquigarrow} w'.\ (w', cr \circ \sigma_w) \in [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}\}$ for $i \in \{1,2\}$. By induction hypothesis on ψ' we assume that $[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$. Given that $\rho(Z)$ has no influence in $cr \circ \sigma_w$, we clearly have that $(w', cr \circ \sigma_w) \in [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]}$ implies $(w', cr \circ \sigma_w) \in [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$. Hence, we conclude $[\![\Diamond\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![\Diamond\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$.

[$\psi \equiv Z$] The proof is trivial, since by definition $[\![Z[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]}$ is equal to $\rho(Z) = \Omega_1$, and $[\![Z[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$ is equal to $\rho(Z) = \Omega_2$. We thus have $[\![Z[\Gamma;\Delta]]\!]_{\rho[^{\Omega_1}/_Z]} \subseteq [\![Z[\Gamma;\Delta]]\!]_{\rho[^{\Omega_2}/_Z]}$.

[$\psi \equiv \mu Z'.\psi'$] The last case of this set regards the monotony of fixed points, i.e. when we have $\psi \equiv \mu Z'.\psi'$ for some $Z' \neq Z$. By definition, $[\![\mu Z'.\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega_i}/_Z]}$ is $lfp(\lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{Y}/_{Z'},^{\Omega_i}/_Z]})$ for $i \in \{1,2\}$. By induction hypothesis we assume that the functions $F' = \lambda Y'.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{Y'}/_{Z'},^{\Omega}/_Z]}$ and $F = \lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega}/_{Z'},^{Y}/_Z]}$ are monotonic for any possible Ω. So, let us consider the values

$\Omega'_i = \mathit{lfp}(\lambda Y'.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{Y'}/_{Z'},^{\Omega_i}/_Z]})$. Then, by the monotonicity of F we have $[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega}/_{Z'},^{\Omega_1}/_Z]} \subseteq [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega}/_{Z'},^{\Omega_2}/_Z]}$ for any possible Ω, so by properties of the least fixed points we get $\Omega'_1 \subseteq \Omega'_2$. By the monotonicity of F and F' we now have $[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega'_1}/_{Z'},^{\Omega_1}/_Z]} \subseteq [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega'_2}/_{Z'},^{\Omega_1}/_Z]} \subseteq [\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{\Omega'_2}/_{Z'},^{\Omega_2}/_Z]}$. So, we conclude $\mathit{lfp}(\lambda Y'.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{Y'}/_{Z'},^{\Omega_1}/_Z]}) \subseteq \mathit{lfp}(\lambda Y'.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[^{Y'}/_{Z'},^{\Omega_2}/_Z]})$.

□

We now turn our attention towards the use of our logic as a specification language in an automated verification setting like model checking.

2.6.2 Decidability of model checking for finite models

As we have already mentioned in the Introduction, most model checking problems for the family of non-propositional modal logics (to which our logic belongs) are in general undecidable. Intuitively, the reasons lie behind the desire of keeping track of elements in highly dynamic systems where components can be allocated and deallocated at any time, giving rise to infinite state-spaces.

Fortunately, the are many such systems which are *resource bounded*. That is, even if new elements can be created infinitely often, the number of elements per state is bounded by some constant. In these cases, resource reallocation techniques (for instance, based on the reuse of resource identifiers) can be applied to reduce the infinite state-space of the system into a finite (semantically equivalent) one, enabling thus the use of automated verification via model checking.

In the next chapters we will define the concept of over- and under-approximations for counterpart models, and that of sound approximated model checking exploiting them. However, intuitively, thanks to the locality of the identifiers, a resource bounded system can be modeled with a *finite* counterpart model, i.e. counterpart models whose set of worlds is finite, and whose algebras have finite domains. In addition we shall consider *finite* formulae-in-context, i.e. finite formulae with finite

contexts, and *finite* assignments, i.e assignments with finite domains and codomains.

It is not trivial that evaluating a finite formula over a finite counterpart model is decidable because of the need to compute fixed points in the semantics of formulae with recursion. Computing them by iteration, for instance, may result in endless loops. However, as the following proposition shows, the problem is indeed decidable, intuitively due to the fact that the functions for which one has to compute the fixed points are monotonic (c.f. Proposition 2.1) and have finite domain.

Proposition 2.2 (Decidability) *Let M be a finite counterpart model and ψ a (naked) formula. Then establishing the validity of the predicate $\_ \in [\![\psi[\Gamma;\Delta]]\!]_\rho$, i.e. solving a local model checking, is decidable in M for any choice of context $[\Gamma;\Delta]$ (such that $\psi[\Gamma;\Delta]$ is well-formed) and of assignment ρ.*

Proof: Being M a finite model means that it has a finite set of worlds, and that the algebra underlying each world is finite: hence the underlying transition system is finitely branching and $\Omega^{[\Gamma;\Delta]}$ is a finite set for any pair Γ, Δ (since both are finite sets by definition). Hence, also $[\![\psi[\Gamma;\Delta]]\!]_\rho$ is finite as well as $\rho(Z)$ for any fix-point variable Z. Given a formula ψ, we need to prove that for any (w,σ) the statement $(w,\sigma) \in [\![\psi[\Gamma;\Delta]]\!]_\rho$ can be verified for any (consistent) choice of Γ, Δ, and ρ. The proof is done by structural induction on ψ.

The proposition trivially holds for the formulae Z and tt (their value being a finite set, as noted before), as well as, by resorting to simple induction, for the boolean connectives $\neg$ and $\vee$.

We provide the proofs for the rest of the cases.

[$\psi \equiv \epsilon \in_\tau Y$] Since any possible interpretation of Y is a finite set, then decidability boils down to verify if an element $\sigma(\epsilon)$ belongs to the finite set $\sigma(Y)$, which is clearly decidable.

[$\psi \equiv \exists_\tau x.\ \psi'$] To check if (w,σ) belongs to $[\![\exists_\tau x.\ \psi'[\Gamma;\Delta]]\!]_\rho$ it is first necessary to check if $(w, \sigma[^a/_x])$ belongs to $[\![\psi'[\Gamma;\Delta]]\!]_{2^{\uparrow_x} \circ \rho}$ for at least one element a of sort τ in the world w. Being the carrier of the underlying algebra finite, though, we need again only to

verify a finite number of instances of the predicate on ψ', which is a decidable procedure.

[$\psi \equiv \exists_\tau Y.\ \psi'$] The proof is similar to the first-order case.

[$\psi \equiv \Diamond\psi'$] To check if (w, σ) belongs to $[\![\Diamond\psi'[\Gamma;\Delta]]\!]_\rho$ it is necessary to check if (w', σ') belongs to $[\![\psi'[\Gamma;\Delta]]\!]_\rho$ for at least a pair (w', σ') such that $w \overset{cr}{\rightsquigarrow} w'$, and σ' is the counterpart assignment of σ via cr. Being M finite means that also the number of outgoing transitions from w is finite, hence we need again only to verify a finite number of instances of the predicate on ψ', which is a decidable procedure.

[$\psi \equiv \mu Z.\psi'$] Note that the fixed point of $\mathit{lfp}(\lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[Y/Z']})$ can be computed as the sequence of values $\Omega_0 = \lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[\emptyset/Z]}$, $\Omega_1 = \lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[\Omega_0/Z]}$, ... , with $\Omega_i \subseteq \Omega_{i+1}$. Since $\Omega^{[\Gamma;\Delta]}$ is a finite set, the sequence is going to reach a fixed point after a finite number of iterations. And since $(w, \sigma) \in \mathit{lfp}(\lambda Y.[\![\psi'[\Gamma;\Delta]]\!]_{\rho[Y/Z]})$ if and only if there exists an j such that $(w, \sigma) \in \Omega_j$, the predicate for ψ is decidable.

□

Chapter 3

Approximating the behaviour of structured systems

Software systems with dynamic resource allocation are often infinite-state. For such systems, verification can become intractable, thus calling for the development of approximation techniques that may ease the verification at the acceptable cost of losing in preciseness and completeness.

Both over- and under-approximations have been considered in the literature, respectively offering more and less behaviors than the original system. At the same time, properties of the system may be either preserved or reflected by a given approximation.

In this chapter we first propose a general notion of counterpart model approximation based on behavioural pre-orders. Then we present a state-space reduction technique based on state canonizers (LMV12), capturing symmetry reduction and name reusing. In the next chapter we will then see how the semantics of our logic is related to model approximations.

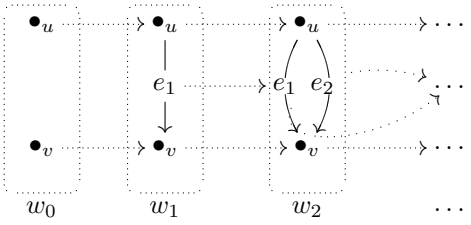

Figure 3.1: An infinite-state counterpart model

3.1 Running example

For a better illustration of our concerns, we will use a very simple model as running example of this chapter.

Figure 3.1 depicts an infinite-state counterpart model M. The model is made of an infinite sequence of worlds w_i, where world w_i is essentially associated to a graph $d(w_i)$ with i edges between nodes u and v. The counterpart relations (drawn with dotted lines) reflect the fact that each transition (w_i, cr_i, w_{i+1}) is such that cr_i is the identity for $d(w_i)$, meaning that all nodes and edges are preserved, while a new edge "e_{i+1}" is created.

In the rest of this chapter we will provide a simple example of over-approximation of this model, where we collapse edges of a world, and an example of under-approximation, where we stop the generation of the model after a few steps. These examples are simple enough to clarify the concepts introduced in this chapter and in the following one.

3.2 Counterpart model approximations

In this section we lift classical behavioural preorders and equivalences for transition systems to counterpart models. Once more, for the sake of presentation, we fix two models $M = (W, \rightsquigarrow, d)$ and $M' = (W', \rightsquigarrow', d')$.

We define relations R from M to M' as sets of triples (w, ϕ, w') formed by a world $w \in W$, a world $w' \in W'$ and a morphism $\phi : d(w) \to d(w')$ relating their respective structures.

Definition 3.1 (Simulation) *Let $R \subseteq W \times (\mathcal{A} \rightharpoonup \mathcal{A}) \times W'$ be a set of triples (w, ϕ, w'), with $\phi : d(w) \to d'(w')$ a morphism. Then R is a* simulation *from M to M' if for every $(w_1, \phi_1, w_1') \in R$ we have that $w_1 \overset{cr}{\rightsquigarrow} w_2$ implies $w_1' \overset{cr'}{\rightsquigarrow} w_2'$ for some $w_2' \in W'$, with $(w_2, \phi_2, w_2') \in R$ and $\phi_2 \circ cr = cr' \circ \phi_1$. If $R^{-1} = \{(w', \phi^{-1}, w) \mid (w, \phi, w') \in R\}$ is well defined, and it is also a simulation, then R (as well as R^{-1}) is called* bisimulation.

Notice how, due to the well-definedness of R^{-1}, the ϕ components of bisimulations are forcibly injections. We call "iso" a bisimulation whose ϕ components are total surjections, and hence (total) isomorphisms. We may abbreviate $(w, \phi, w') \in R$ in wRw' if ϕ is irrelevant.

As usual, we define (bi)similarity as the greatest (bi)simulation, and say that M *is similar to* M' or that M' *simulates* M, written $M \sqsubseteq_R M'$ (where we may omit R), if there exists a simulation R from M to M' such that, for every $w \in W$, there exists at least a $w' \in W'$ with wRw'. Similarly, we say that two models are doubly similar, written $M \backsimeq M'$, if $M \sqsubseteq M'$ and $M' \sqsubseteq M$, and, finally, we say that they are (iso-)bisimilar, written $M \sim M'$ ($M \sim_i M'$), if there exists an (iso-)bisimulation R such that $M \sqsubseteq_R M'$ and $M' \sqsubseteq_{R^{-1}} M$.

Given a set of pairs $\omega \subseteq \Omega_M$ and a simulation R from M to M' we use $R(\omega)$ to denote the set $\{(w', \phi \circ \sigma_w) \mid (w, \sigma_w) \in \omega \wedge (w, \phi, w') \in R\}$. Moreover, in the following, with an abuse of notation we use $R \circ \rho$ to indicate the composition of R with the fix-point assignment ρ, defined as $R \circ \rho = \{(Z \mapsto R(\omega)) \mid (Z \mapsto \omega) \in \rho\}$.

Note that R^{-1} is not always well-defined since the morphisms in the triples (w, ϕ, w') may not be injective. However, we often use the pre-image $R^{-1}[\cdot]$ of R, defined for a set of pairs $\omega' \subseteq \Omega_{M'}$ as $R^{-1}[\omega'] = \{(w, \sigma_w) \in \Omega_M \mid \exists (w, \phi, w') \in R.\ (w', \phi \circ \sigma_w) \in \omega'\}$.

Definition 3.2 (Under- and over-approximations) *Let M, $\underline{M}$ and $\overline{M}$, be counterpart models, and $\underline{R}$, $\overline{R}$ be, respectively, simulations from $\underline{M}$ to M and from M to $\overline{M}$ such that $\underline{M} \sqsubseteq_{\underline{R}} M \sqsubseteq_{\overline{R}} \overline{M}$. Then $\underline{M}$ and $\overline{M}$ are, respectively, an* under- *and* over-approximation *of M.*

Example 3.1 *Figure 3.2 depicts three counterpart models: M (center), $\overline{M}$ (top) and $\underline{M}$ (bottom).*

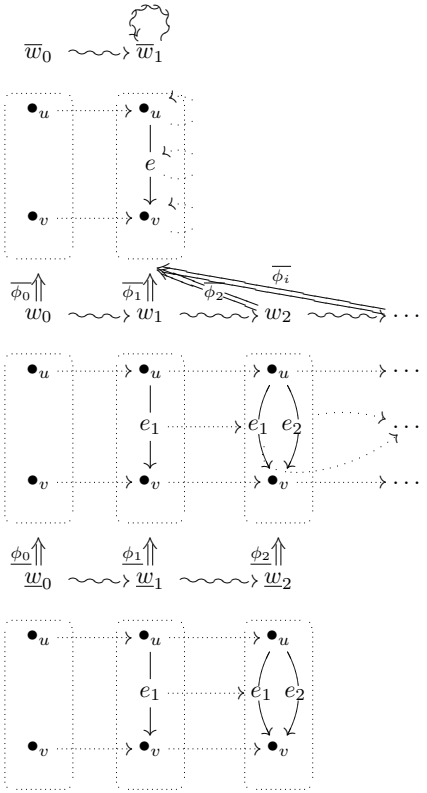

Figure 3.2: A model M (center), an over-approximation $\overline{M}$ (top) and an under-approximation $\underline{M}$ (bottom)

The model M, taken from Section 3.1 and Figure 3.1, is infinite-state. The finite-state models $\overline{M}$ and $\underline{M}$ can be understood, respectively, as over- and under-approximations of M. Indeed, we have relations $\overline{R}$ and $\underline{R}$ (denoted with double arrows) such that $\underline{M} \sqsubseteq_{\underline{R}} M \sqsubseteq_{\overline{R}} \overline{M}$.

Intuitively, $\underline{M}$ is a truncation of M considering only the first two transitions of M. Every tuple $(\underline{w}, \underline{\phi}, w)$ in $\underline{R}$ is such that $\underline{\phi} : \underline{d}(\underline{w}) \to d(w)$ is the identity.

On the other hand, the over-approximation $\overline{M}$ can be seen as "M modulo the fusion of edges". That is, every tuple $(w, \overline{\phi}, \overline{w})$ in $\overline{R}$ is such that $\overline{\phi} : d(w) \to \overline{d}(\overline{w})$ is a bijection for nodes (in particular, the identity restricted to the nodes of $d(w)$) and a surjection on edges mapping every edge e_i into edge e.

3.3 Reductions for counterpart models

In (LMV12) we presented "c-reductions" (`canonical`-reductions), a state-space reduction technique based on *canonizers*, namely functions mapping worlds to a (non necessarily unique) canonical representative of their equivalence class (given by a bisimulation). The approach is general enough to recast in it other reduction techniques like symmetry reduction (WD10), name reusing and name abstraction.

We proposed the technique for systems whose states are specified as multisets of concurrent, interrelated and distributed components, and whose dynamics is given as term rewrite rules. The obtained models belong to the class of *Kripke models*, where, as previously mentioned, in contrast to counterpart models no explicit structure is provided to relate components of different states (i.e. transitions are not labelled with counterpart relations). State components are hence implicitly syntactically identified across different worlds.

We implemented the approach in the setting of rewriting logic (Mes12) and the Maude framework (CDE+07), which provides automatization of the reduction infrastructure via meta-programming features, and reasoning support for checking correctness of the reduction (with respect to the original model) through its toolset, comprising the Maude LTL Model Checker (EMS03), the Invariant Analyzer (Inv), the Inductive Theorem Prover (CPR06; ITP) and the Church Rosser and Coherence Checker (DM10)).

We evaluated our approach considering an ample set of examples showing some interesting results: with respect to previous works we observed performance gains in some cases (including previous implementations of symmetry reductions in Maude (Rod09)), and more flexibility in the definition of reductions, allowing us to subsume a wide range of them, including permutation and rotation symmetries, name reuse and name abstraction, which have interesting applications, for example in the implementation of the operational semantics of languages with dynamic features such as resource allocation. A preliminary version of our tool is available for download (CRe).

In this section we see how c-reductions can be extended to reduce counterpart models.

Intuitively, we actually do not modify c-reductions themselves, but rather the way they are applied to a model. Namely, when applying a c-reduction to a world of a counterpart model, we also apply the induced components' renaming to the counterpart relations labelling the involved transitions.

We now introduce `canonical`-reductions (c-reductions) as a generic means to reduce a counterpart model M by exploiting an iso-bisimulation R on it (i.e. between M and itself). We start by defining canonizer functions, used to compute the representative of the equivalence class of (the structure associated to) a world, modulo R.

Definition 3.3 (canonizer functions) *Let $M = (W, \rightsquigarrow, d)$ be a counterpart model, and let R be an iso-bisimulation on M, that is among M and itself. A function $c : W \to W$ is an R-canonizer (resp. strong R-canonizer) if for every $w \in W$ we have exactly one $(w, \phi, c(w)) \in R$ (resp. $(w, \phi, c(w)) \in R$, and wRw' implies $c(w) = c(w')$).*

From the fact that R is an iso-bisimulation we know that its ϕ components are total isomorphisms. Intuitively, ϕ can be thought of as the renaming induced by the application of c to the components of $d(w)$ to obtain the ones of $d(c(w))$. In the following we will use $\alpha_{\mathtt{c}}(w)$ to indicate such renaming.

Canonizer functions are used to compute smaller but semantically equivalent (i.e. iso-bisimilar) counterpart models by applying canonizers after each transition. We distinguish among *strong* and *weak* canonizers.

Strong canonizers provide unique representatives for the equivalence classes of worlds, meaning that they collapse each equivalence class of worlds in a world, providing hence the maximal reduction given an equivalence relation. Well known examples of strong canonizers are based on *enumeration* strategies (DM06) which generate the complete set of worlds of an equivalence class, and then apply some function over it (e.g. based on a total ordering for worlds) to choose the canonical one (e.g. the minimal one for a given ordering for worlds).

Consider systems where some class of processes exhibit a *full symmetry*, of which a well-known example is the Peterson's mutual exclusion protocol (Lyn96). Here we can safely swap the names of any two processes in a state. Then an enumeration strategy canonizer just generates all structures resulting from permuting (symmetric) processes, and then selects one according to some total order (e.g. a lexicographical-like ordering for the structures). Of course, as detailed in (BDH01; BDH02), canonizers can be obtained in more efficient and smarter ways.

In the case of weak canonizers, we might have different representatives for equivalent worlds. Weak canonizers provide weaker space reductions but may enjoy advantages over strong ones, like easiness of definition and analisys, and less expensive computation.

Examples of weak canonizers can be found for instance in (BDH01; BDH02) where the rough idea is to consider an ordering of the states that depends on part of their structure only, resulting in a partial ordering.

As concrete examples, in Chapter 7 we present and discuss two canonizers which perform two particular flavours of symmetry reduction, namely the full one and the rotational one. When we apply the canonizers to an algebraic structure labelling a world, all the structures symmetric to the original one are generated, and the minimal one (provided an ordering) is returned. Intuitively, if the considered ordering is total, then those canonizers are strong, otherwise they are weak, as two symmetric structures may be mapped to distinct minimal structures. More details are provided in Chapter 7.

We obtain the *c-reduction* of a counterpart model applying the canonizer to worlds after each transition.

Definition 3.4 (c-reduction of a counterpart model) *Let $M = (W, \rightsquigarrow, d)$ be a counterpart model, and $c : W \to W$ an R-canonizer function for an iso-bisimulation R on M. We call $M_c = (W, (\rightsquigarrow; c), d)$ the* c-reduction *of M, whose (composed) transition relation is defined as $\rightsquigarrow; c = \{(w_1, \alpha_c(w_2) \circ cr, c(w_2)) \mid (w_1, cr, w_2) \in \rightsquigarrow\}$.*

Intuitively, by composing $\rightsquigarrow$ with the canonizer c, we map the worlds of a model in the ones obtained applying the canonizer to them.

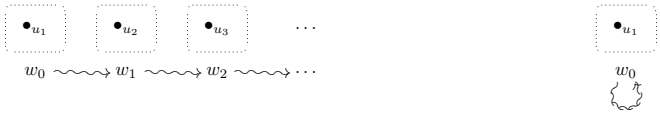

Figure 3.3: A model (left) and its c-reduction (right)

Practically, we can easily obtain an on-the-fly reduction technique during the generation of a counterpart model by simply applying a canonizer to every newly generated world, before storing it. This aspect will be better discussed in Chapter 7.

An important result shown in (LMV12) for Kripke models is that c-reductions preserve bisimulations. We now state that c-reductions preserve iso-bisimulations when applied to counterpart models.

Proposition 3.1 ($\sim$-preservation) *Let M be a counterpart model, R be an iso-bisimulation on M, and c be a R-canonizer. Then $M \sim_i M_{\mathsf{c}}$.*

Proof: The proof is straightforward, and is hence only sketched. Let $M = (W, \rightsquigarrow, d)$ and $M_{\mathsf{c}} = (W, (\rightsquigarrow; \mathsf{c}), d)$. From Definition 3.3 we know that there exists an iso-bisimulation R such that for any $w \in W$ we have $wR\mathsf{c}(w)$. In particular, we can define a relation $R_{\mathsf{c}} = \{(w, \alpha_{\mathsf{c}}(w), \mathsf{c}(w)) \mid w \in W\}$, which is clearly an iso-bisimulation from M to M_{c}. Finally, we notice that for each world in M there exists an R_{c}-bisimilar world in M_{c}, and that for each world in M_{c} there exists an R_{c}^{-1}-bisimilar world in M, which implies $M \sim_i M_{\mathsf{c}}$. □

Example 3.2 (c-reduction of a counterpart model) *Consider the simple, but still infinite-state, model M of Figure 3.3 (left), where each state has only a component, which is deallocated and replaced by a new one after each transition.*

Then one could easily define a canonizer c that maps the name of the component of each state to the same one.

Figure 3.3 (right) shows the c-reduction M_{c}, containing only a state, and a transition relation consisting of a self-loop with empty counterpart relation. This is a very simple example, more interesting ones will be proposed in Chapter 8.

Chapter 4

Sound approximated model checking of infinite-state systems

In the previous chapter we defined the concept of approximation of a counterpart model, and we exemplified a technique to obtain them.

In this chapter we study how we can exploit such approximations by analyzing how the semantics of our logic is related to them. In particular, in Section 4.1 we first define the concepts of *preservation* and *reflection* of a formula by an approximation, and present a partial type system to type formulae as preserved and/or reflected. Then, in Section 4.2 we define a sound approximated model checking procedure which exploit our type system to estimate the evaluation of a formula, based on sets of under- and over-approximations. In Section 4.3 we extend our approximated model checking procedure to deal with part of the untyped formulae. Finally, for the sake of presentation we provide all the soundness proofs in Section 4.4.

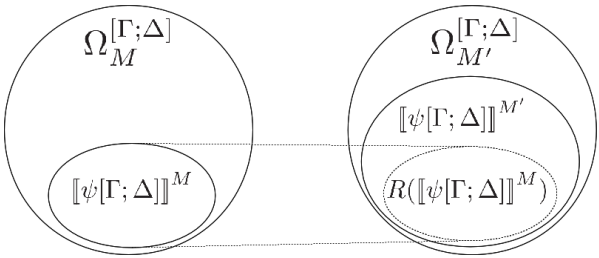

Figure 4.1: A formula (ψ) preserved by a simulation (R).

4.1 Preservation and reflection of formulae

Intuitively, if a model M' is an approximation of a model M, then the evaluation of formulae in M can be only approximated resorting to M'. We hence introduce the usual notions of *preserved* formulae, those whose "satisfaction" in M implies their "satisfaction" in M', and *reflected* formulae, those whose "satisfaction" in M' implies their "satisfaction" in M. Of course, since the semantic domain of our logic are assignment pairs, the notion of "satisfaction" corresponds to the existence of such pairs.

Definition 4.1 (Preservation and reflection) *Let R be a simulation from M to M' (i.e. $M \sqsubseteq_R M'$), $\psi[\Gamma;\Delta]$ a formula, and ρ an assignment. We say that ψ is* preserved *under R (written $\psi :_R \Rightarrow$) if $[\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho} \supseteq R([\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho})$;* reflected *under R (written $\psi :_R \Leftarrow$) if $R^{-1}[[\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}] \subseteq [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$; and* strongly preserved *under R (written $\psi :_R \Leftrightarrow$) if $\psi :_R \Rightarrow$ and $\psi :_R \Leftarrow$.*

Note that the choice of ρ, Γ, and Δ is irrelevant. In the definition of $\psi :_R \Leftarrow$ we use $R^{-1}[\cdot]$ rather than R^{-1} because the latter is not always defined. Moreover, by resorting to $R^{-1}[\cdot]$ we obtain a stronger (i.e. stricter) property rather than resorting to R^{-1}. In fact, if $\psi :_R \Leftarrow$, then we additionally have that $R([\![\neg\psi[\Gamma;\Delta]]\!]^{M}_{\rho}) \cap [\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho} = \emptyset$, i.e. that a pair in $\Omega_M^{[\Gamma;\Delta]} \setminus [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$ cannot be "similar" to any pair in $[\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$.

Figure 4.1 exemplifies a formula (ψ) preserved by a simulation (R). In particular, the two big circles on the left and on the right of the figure represent the set of pairs of the two models, i.e. respectively, $\Omega_M^{[\Gamma;\Delta]}$ and $\Omega_{M'}^{[\Gamma;\Delta]}$. Instead, the two ellipses therein contained represent the evaluation

of ψ in M ($[\![\psi[\Gamma;\Delta]]\!]^M$) and M' ($[\![\psi[\Gamma;\Delta]]\!]^{M'}$). Finally, we know that the formula is preserved because $R([\![\psi[\Gamma;\Delta]]\!]^M)$, depicted in the figure as a dotted ellipse, is contained in $[\![\psi[\Gamma;\Delta]]\!]^{M'}$.

Example 4.1 *Consider the model M of Figure 3.1, and its over- and under-approximations $\overline{M}$ and $\underline{M}$ of Example 3.1 shown in Figure 3.2, i.e $\underline{M} \sqsubseteq_{\underline{R}} M \sqsubseteq_{\overline{R}} \overline{M}$, and let r, z, x, y be first-order variables. As stated in the previous chapters, in our logic it is easy to define a predicate regarding the presence of an entity with sort τ in a world as $\boldsymbol{present}_\tau(z) \equiv \exists_\tau r.\ z = r$. The predicate evaluates in pairs $(w, (\{z \mapsto a\}, \lambda_2))$, with a being an element of the algebra assigned to the world w. Now, the predicate (omitting typings) $\boldsymbol{p}(x, y) \equiv \boldsymbol{present}(z) \wedge s(z) = x \wedge t(z) = y$ regards the existence of an edge (the one assigned to z) connecting two nodes x and y.*

Evaluating $\boldsymbol{p}(u, v)$ in M, having the concrete nodes u and v in its worlds, we obtain pairs whose assignment components map z to edges connecting u to v. In particular, considering the context $[\{z\}; \emptyset]$, we obtain $[\![\boldsymbol{p}(u, v)]\!]^M = \{(w_1, (\{z \mapsto e_1\}, \lambda_2)), (w_2, (\{z \mapsto e_1\}, \lambda_2)), (w_2, (\{z \mapsto e_2\}, \lambda_2)), \dots\}$, where λ_2 is the empty second-order variable assignment.

Now, it is easy to see that $\boldsymbol{p}(u, v)$ is strongly preserved under both $\underline{R}$ and $\overline{R}$. In fact, we have that $[\![\boldsymbol{p}(u, v)]\!]^{\underline{M}} = \{(\underline{w}_1, (\{z \mapsto e_1\}, \lambda_2)), (\underline{w}_2, (\{z \mapsto e_1\}, \lambda_2)), (\underline{w}_2, (\{z \mapsto e_2\}, \lambda_2))\}$, and hence $\underline{R}([\![\boldsymbol{p}(u, v)]\!]^{\underline{M}})$ is $\{(w_1, (\{z \mapsto e_1\}, \lambda_2)), (w_2, (\{z \mapsto e_1\}, \lambda_2)), (w_2, (\{z \mapsto e_2\}, \lambda_2))\}$, which is clearly contained in $[\![\boldsymbol{p}(u, v)]\!]^M$. Moreover we also have that $\underline{R}^{-1}[[\![\boldsymbol{p}(u, v)[\Gamma;\Delta]]\!]^M] \subseteq [\![\boldsymbol{p}(u, v)[\Gamma;\Delta]]\!]^{\underline{M}}$. We hence have that $\boldsymbol{p}(u, v) :_{\underline{R}} \Leftrightarrow$.

Similarly, we have that $[\![\boldsymbol{p}(u, v)]\!]^{\overline{M}} = \{(\overline{w}_1, (\{z \mapsto e\}, \lambda_2))\}$, and that $\overline{R}([\![\boldsymbol{p}(u, v)]\!]^{\overline{M}}) = \{(\overline{w}_1, (\{z \mapsto e\}, \lambda_2))\}$. Both conditions are again satisfied, and hence we have $\boldsymbol{p}(u, v) :_{\overline{R}} \Leftrightarrow$.

Of course, determining whether a formula is preserved (or reflected) cannot be done in practice by performing the above check, since that would require to calculate the evaluation of the formula in the (possibly infinite) original model M, which is precisely what we want to avoid. Moreover, note that determining wheter a formula is preserved (and the same occurs for being reflected) is an undecidable problem, since our logic subsumes that of (BKK03).

Fortunately, we can apply the same approach of (BKK03) and define a (partial) type system that approximates the preservation and reflection of

formulae. In particular, our type system generalizes the one of (BKK03) in several directions:

1. Our type system is agnostic with respect to the approximation technique adopted, and it is parametric with respect to the properties of the (morphisms of the) simulations R. While the original one is given for graph morphisms that are total and bijective for nodes and total and surjective for edges, we exploit the injectivity, surjectivity and totality of the morphisms of R for each sort τ;

2. We consider counterpart models, a generalization of graph transition systems;

3. We use the type system to reason on all formulae, while the original proposal restricts to closed ones;

4. We exploit over- and under-approximations of a model to obtain more precise approximated formulae evaluations;

5. As for the original proposal, not all formulae can be dealt by our type system. However, thanks to our enriched approximated semantics (Section 4.3), we are able to handle part of the untyped formulae.

It may be useful to comment on Point 1, where we state that the type system is parametric with respect to the properties of R. In particular, we actually consider the properties of the morphism components of R. Namely, for each sort τ, we distinguish τ-total (τ_t), τ-surjective (τ_s) or τ-bijective (τ_b). To ease the presentation, we say "$\tau_{prop}\ R$", with $prop \in \{t, s, b\}$, whenever all $(w, \phi, w') \in R$ are such that ϕ is τ-$prop$. Moreover, we shall consider the case in which R is an iso-bisimulation.

Definition 4.2 (Type system) *Let R be a simulation from M to M' (that is $M \sqsubseteq_R M'$), ψ a formula, and $\mathcal{T} = \{\leftarrow, \rightarrow, \leftrightarrow\}$ a set of types. We say that ψ*

has type $d \in \mathcal{T}$ if $\psi : d$ can be inferred using the following rules

$$\frac{}{tt :_R \leftrightarrow} \qquad \frac{d = \begin{cases} \rightarrow & \text{for } \tau_t\ R \\ \leftarrow & \text{for } \tau_b\ R \end{cases}}{\epsilon \in_\tau Y :_R d} \qquad \frac{\psi :_R \rightarrow \quad \psi :_R \leftarrow}{\psi :_R \leftrightarrow} \qquad \frac{\psi :_R \leftrightarrow}{\psi :_R d}$$

$$\frac{\psi_i :_R d}{\psi_1 \vee \psi_2 :_R d} \qquad \frac{\psi :_R d \text{ with } d = \begin{cases} \rightarrow & \text{for } \tau_t\ R \\ \leftarrow & \text{for } \tau_s\ R \end{cases}}{\exists_\tau x.\psi :_R d \text{ and } \exists_\tau Y.\psi :_R d} \qquad \frac{\psi :_R d}{\neg\psi :_R d^{-1}}$$

$$\frac{}{Z :_R \leftrightarrow} \qquad \frac{\psi :_R d \text{ with } d = \begin{cases} \rightarrow & \text{for any } R \\ \leftarrow & \text{for } R \text{ an iso-bisimulation} \end{cases}}{\Diamond\psi :_R d} \qquad \frac{\psi :_R d}{\mu Z.\psi :_R d}$$

where it is intended that $\rightarrow^{-1} = \leftarrow$, $\leftarrow^{-1} = \rightarrow$ and $\leftrightarrow^{-1} = \leftrightarrow$.

The type system is not complete, meaning that some formulae cannot be typed: if ψ cannot be typed, we then write $\psi :_R \bot$. However, the next proposition, proved in Section 4.4, states its soundness.

Proposition 4.1 (Type system soundness) *Let R be a simulation from M to M' (i.e. $M \sqsubseteq_R M'$) and ψ a formula. Then*

(i) $\psi :_R \rightarrow$ implies $\psi :_R \Rightarrow$;

(ii) $\psi :_R \leftarrow$ implies $\psi :_R \Leftarrow$;

(iii) $\psi :_R \leftrightarrow$ implies $\psi :_R \Leftrightarrow$.

As we already noted, our type system can be instantiated for graph signatures, in order to obtain the one of (BKK03) as a subsystem. In fact, the authors there consider only simulation relations R that are total on both sorts, as well as being $(\tau_N)_b$ (that is, bijective on nodes) and $(\tau_E)_s$ (surjective on edges).

Another instance is for iso-bisimulations. This is the case of the analysis of graph transition systems up to isomorphism (e.g. as implemented in (Ren06b)). In this case the type system is complete and correctly types every formula as $\psi :\leftrightarrow$.

Example 4.2 *Consider the models $\underline{M} \sqsubseteq_{\underline{R}} M \sqsubseteq_{\overline{R}} \overline{M}$ shown in Figure 3.2, and the formula $\boldsymbol{p}(u, v)$ of Example 4.1, where we saw that $\boldsymbol{p}(u, v) :_{\underline{R}} \Leftrightarrow$ and*

$\boldsymbol{p}(u,v) :_{\overline{R}} \leftrightarrows$. *Our type system provides the types* $\boldsymbol{p}(u,v) :_{\underline{R}} \leftrightarrow$ *and (since* $\overline{R}$ *is not injective on edges)* $\boldsymbol{p}(u,v) :_{\overline{R}} \rightarrow$. *Note that the type for* $\underline{R}$ *is exactly inferred, while for* $\overline{R}$ *it is only approximated as we get* preserved *while it is actually* strongly preserved.

4.2 Approximated semantics and model checking

Model approximations can be used to estimate the evaluation of formulae. Consider the case of three models $\underline{M}$, M and $\overline{M}$, with $\underline{M} \sqsubseteq_{\underline{R}} M \sqsubseteq_{\overline{R}} \overline{M}$, as in Figure 3.2, where $\underline{M}$ and $\overline{M}$ are under- and over-approximations of M, respectively. Intuitively, an approximated evaluation of a formula ψ in $\underline{M}$ or $\overline{M}$ may provide us a lower- and upper-bound, defined for either $\underline{M}$ or $\overline{M}$, of the actual evaluation of ψ in M. We call under- and over-approximated evaluations the ones obtained using, respectively, under- (e.g. $\underline{M}$), and over-approximations (e.g. $\overline{M}$).

Exploiting approximated evaluations, we may address the *local* model checking problem: *"does a given assignment pair belong to the evaluation of the formula* ψ *in* M*?"*. Formally, $(w,\sigma) \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

Given that our approximated semantics compute lower- and upper-bounds, we cannot define a complete procedure, i.e. one answering either $true$ or $false$. A third value is required for the cases of uncertainty. For this purpose we use a standard three valued logic (namely the Kleene's one) whose domain consists of the set of values $\mathbb{K} = \{T, F, ?\}$ (where $?$ reads "unknown"), and whose operators extend the standard Boolean ones with $T \vee ? = T$, $F \vee ? = ?$, $\neg ? = ?$ (i.e. where disjunction is the join in the complete lattice induced by the *truth* ordering relation $F < ? < T$). Moreover, we also consider a *knowledge addition* (binary, associative, commutative, partial) operation $\oplus : \mathbb{K} \times \mathbb{K} \rightharpoonup \mathbb{K}$ defined as $T \oplus T = T$, $F \oplus F = F$ and $x \oplus ? = x$ for any $x \in \mathbb{K}$. Notice how we intentionally let undefined the case of contradictory knowledge addition "$F \oplus T$", a case that, as we will prove, in our framework cannot arise.

We first consider the case of over-approximations, and then, similarly, the case of under-approximations.

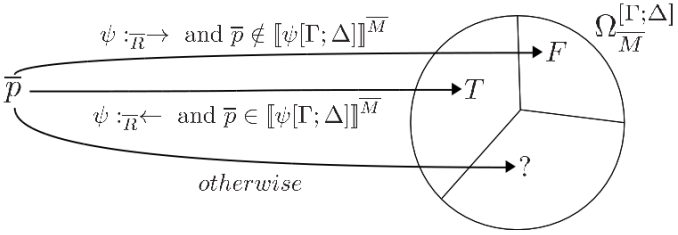

Figure 4.2: Over-approximated semantics.

Definition 4.3 (Over-approximated semantics) *Let $\overline{R}$ be a simulation from M to $\overline{M}$ (i.e. $M \sqsubseteq_{\overline{R}} \overline{M}$) and ρ an assignment. Then the* over-approximated semantics *of $[\![\cdot]\!]_\rho^M$ in $\overline{M}$ via $\overline{R}$ is given by the function $\{\!|\cdot|\!\}_\rho^{\overline{R}} : \mathcal{F}^{[\Gamma;\Delta]} \rightarrow (\Omega_{\overline{M}}^{[\Gamma;\Delta]} \rightarrow \mathbb{K})$, defined as $\{\!|\psi[\Gamma;\Delta]|\!\}_\rho^{\overline{R}} = \{(\overline{p}, \overline{k}(\overline{p}, \psi[\Gamma;\Delta], \overline{R})) \mid \overline{p} \in \Omega_{\overline{M}}^{[\Gamma;\Delta]}\}$, where*

$$\overline{k}(\overline{p}, \psi[\Gamma;\Delta], \overline{R}) = \begin{cases} T & \text{if } \psi :_{\overline{R}}\leftarrow \text{ and } \overline{p} \in [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho} \\ F & \text{if } \psi :_{\overline{R}}\rightarrow \text{ and } \overline{p} \notin [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho} \\ ? & \text{otherwise} \end{cases}$$

Intuitively, given a formula ψ, with our approximated semantics we group the pairs of an approximating model (in this case $\overline{M}$) in three distinct sets: the ones associated with T, the ones associated with F, and the ones associated with ?. In particular, as depicted in Figure 4.2 (where we omit the informations about the fix-point assignment), the boolean value associated to any of the pairs $\overline{p} \in \Omega_{\overline{M}}^{[\Gamma;\Delta]}$ depends on the type of ψ. If it is typed as reflected, then the pairs in $[\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$ are mapped to T, since their counterparts in M do certainly belong to the evaluation of ψ (top arrow of Figure 4.2). Nothing can be said about the rest of the pairs, which are hence mapped to ? (middle arrow of Figure 4.2).

Dually, if ψ is typed as preserved, then the pairs that do not belong to $[\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$ are mapped to F because we know that their counterparts in M do certainly not belong to the evaluation of ψ (bottom arrow of Figure 4.2). Again, nothing can be said about the rest of the pairs, which are hence mapped to ? (middle arrow of Figure 4.2).

Finally, if ψ cannot be typed, then all the pairs in $\Omega_{\overline{M}}^{[\Gamma;\Delta]}$ are mapped to ? (middle arrow of Figure 4.2).

	$\{\!\{\mathbf{p}(u,v)\}\!\}$	$\{\!\{\neg\mathbf{p}(u,u)\}\!\}$	$\{\!\{\mathbf{p}(u,v) \vee \neg\mathbf{p}(u,u)\}\!\}$	${}^{+}\{\!\{\mathbf{p}(u,v) \vee \neg\mathbf{p}(u,u)\}\!\}$
$(\overline{w}_0, \lambda)$, $(\overline{w}_1, \lambda)$	F	T	?	T
$(\overline{w}_1, (z \mapsto e, \lambda_2))$	?	T	?	T
$(\underline{w}_0, \lambda)$,$(\underline{w}_1, \lambda)$,$(\underline{w}_2, \lambda)$	F	T	T	T
$(\underline{w}_1, (z \mapsto e_1, \lambda_2)$	T	T	T	T
$(\underline{w}_2, (z \mapsto e_1, \lambda_2)$	T	T	T	T
$(\underline{w}_2, (z \mapsto e_2, \lambda_2))$	T	T	T	T
$(w_2, (z \mapsto e_2, \lambda_2)) \models^{\overline{R}} [\![\cdot]\!]$	?	T	?	T
$(w_2, (z \mapsto e_2, \lambda_2)) \models_{\underline{R}} [\![\cdot]\!]$	T	T	T	T
$(w_2, (z \mapsto e_2, \lambda_2)) \models^{\overline{R}}_{\underline{R}} [\![\cdot]\!]$	T	T	T	T

Figure 4.3: Approximated semantics and checks for some formulae and pairs

Notice how, in practice, we rarely have to explicitly compute $\overline{R} \circ \rho$. In fact, formulae of our logic are thought to be evaluated under an initial empty assignment for fix-point variables, which is manipulated during the evaluation. Clearly $\overline{R} \circ \emptyset = \emptyset$ for any $\overline{R}$, and it can be shown that the rules of the semantics manipulating the fix-point assignment are consistent with respect to the it.

We can use the over-approximated semantics to decide whether an assignment pair belongs to the evaluation of a formula in M or not, as formalized below.

Definition 4.4 (Over-check) *Let $\overline{R}$ be a simulation from M to $\overline{M}$ (i.e. $M \sqsubseteq_{\overline{R}} \overline{M}$) and ρ an assignment. The* over-approximated model check *(shortly,* over-check*) of $[\![\cdot]\!]^M_\rho$ in $\overline{M}$ via $\overline{R}$ is given by the function $\cdot \models^{\overline{R}} [\![\cdot]\!]^M_\rho : \Omega_M^{[\Gamma;\Delta]} \times \mathcal{F}^{[\Gamma;\Delta]} \to \mathbb{K}$, defined as*

$$p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = \bigvee_{\overline{p} \in \overline{R}(p)} \{\![\psi[\Gamma;\Delta]]\!\}^{\overline{R}}_\rho(\overline{p})$$

Example 4.3 *Consider again the predicate $\boldsymbol{p}(x,y)$ of Example 4.1 stating the existence of an edge (the one assigned to the variable z) connecting node x to node y, and the models M and $\overline{M}$ with $M \sqsubseteq_{\overline{R}} \overline{M}$ of Example 3.1 shown in Figure 3.2.*

In the first group of lines of Figure 4.3 we exemplify the over-approximated semantics in $\overline{M}$ via $\overline{R}$ of $[\![\boldsymbol{p}(u,v)]\!]^M$, $[\![\neg\boldsymbol{p}(u,u)]\!]^M$, and $[\![\boldsymbol{p}(u,v) \vee \neg\boldsymbol{p}(u,u)]\!]^M$, considering the pairs $\Omega_{\overline{M}}^{[z;\emptyset]} = \{(\overline{w}_0, \lambda), (\overline{w}_1, \lambda), (\overline{w}_1, (z \mapsto e, \lambda_2))\}$. We recall from Example 4.2 that $\boldsymbol{p}(u,v) :_{\overline{R}} \rightarrow$, and, hence, $\neg\boldsymbol{p}(u,u) :_{\overline{R}} \leftarrow$ and $\boldsymbol{p}(u,v) \vee$

$\neg \boldsymbol{p}(u,u) :_{\overline{R}} \bot$. *Moreover, from Example 4.1 we know* $[\![\boldsymbol{p}(u,v)]\!]^{\overline{M}} = \{(w_1, (z \mapsto e, \lambda_2))\}$, *while it is easy to see that* $[\![\neg \boldsymbol{p}(u,u)]\!]^{\overline{M}} = \Omega_{\overline{M}}^{[z;\emptyset]}$, *because no edge exists in* $\overline{M}$ *with same source and target* u *(this is actually true for any node). Following Definition 4.3, we hence have that* $(\overline{w}_0, \lambda)$ *and* $(\overline{w}_1, \lambda)$ *are mapped to* F *for* $\boldsymbol{p}(u,v)$, *and to* T *for* $\neg \boldsymbol{p}(u,u)$, *while* $(\overline{w}_1, (z \mapsto e, \lambda_2))$ *is mapped to* $?$ *and to* T. *Different is the case of* $\boldsymbol{p}(u,v) \vee \neg \boldsymbol{p}(u,u)$*: the formula cannot be typed and its approximated semantics hence maps the three pairs to* $?$.

In the third group of lines of Figure 4.3 we find the over-check "$\cdot \models^{\overline{R}} [\![\cdot]\!]$*" of* $(w_2, (z \mapsto e_2, \lambda_2))$ *in* $\overline{M}$ *via* $\overline{R}$ *for the three formulae. Note that* $\overline{R}((w_2, (z \mapsto e_2, \lambda_2))) = (\overline{w}_1, (z \mapsto e, \lambda_2))$, *hence the over-checks of* $\boldsymbol{p}(u,v)$ *and of* $\boldsymbol{p}(u,v) \vee \neg \boldsymbol{p}(u,u)$ *give* $?$, *because no pair in* $\overline{R}((w_2, (z \mapsto e_2, \lambda_2)))$ *is mapped to either* T *or* F. *Instead, given that* $\{[\neg \boldsymbol{p}(u,u)]\}((\overline{w}_1, (z \mapsto e, \lambda_2))) = T$, *then we have* $(w_2, (z \mapsto e_2, \lambda_2)) \models^{\overline{R}} [\![\neg \boldsymbol{p}(u,u)]\!] = T$.

With the next proposition, proved in Section 4.4, we state that the above described check is sound.

Proposition 4.2 (Soundness of over-check) *Let* $\overline{R}$ *be a simulation from* M *to* $\overline{M}$ *(i.e.* $M \sqsubseteq_{\overline{R}} \overline{M}$*),* $\psi[\Gamma;\Delta]$ *a formula, and* ρ *an assignment. Then*

(i) $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]_\rho^M = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]_\rho^M$;

(ii) $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]_\rho^M = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]_\rho^M$.

Now, we can define the under-approximated semantics in a specular way.

Definition 4.5 (Under-approximated semantics) *Consider* $\underline{R}$ *being a simulation from* $\underline{M}$ *to* M *(i.e.* $\underline{M} \sqsubseteq_{\underline{R}} M$*) and* ρ *an assignment. Then, the* under-approximated semantics *of* $[\![\cdot]\!]_\rho^M$ *in* $\underline{M}$ *via* $\underline{R}$ *is the function* $\{\!|\cdot|\!\}_\rho^{\underline{R}} : \mathcal{F}^{[\Gamma;\Delta]} \to (\Omega_{\underline{M}}^{[\Gamma;\Delta]} \to \mathbb{K})$, *defined as* $\{\!|[\![\psi[\Gamma;\Delta]]\!]\!\}_\rho^{\underline{R}} = \{\underline{p} \mapsto \underline{k}(\underline{p}, \psi[\Gamma;\Delta], \underline{R}) \mid \underline{p} \in \Omega_{\underline{M}}^{[\Gamma;\Delta]}\}$, *where*

$$\underline{k}(\underline{p}, \psi[\Gamma;\Delta], \underline{R}) = \begin{cases} T & \text{if } \psi :_{\underline{R}} \rightarrow \text{ and } \underline{p} \in [\![\psi[\Gamma;\Delta]]\!]_{\underline{R}^{-1}[\cdot]\circ\rho}^{\underline{M}} \\ F & \text{if } \psi :_{\underline{R}} \leftarrow \text{ and } \underline{p} \notin [\![\psi[\Gamma;\Delta]]\!]_{\underline{R}^{-1}[\cdot]\circ\rho}^{\underline{M}} \\ ? & \text{otherwise} \end{cases}$$

Intuitively, as for the over-approximated case, the pairs of $\Omega_{\underline{M}}^{[\Gamma;\Delta]}$ are mapped depending on the type of ψ.

If ψ is preserved, then all pairs in $[\![\psi[\Gamma;\Delta]]\!]^{M}_{\underline{R}^{-1}[\cdot]\circ\rho}$ are mapped to T, since we know that their counterparts in M do certainly belong to the evaluation of ψ. Nothing can be said about the rest of the pairs, which are hence mapped to ?.

Dually, if ψ is reflected, then all pairs in $\Omega^{[\Gamma;\Delta]}_{\underline{M}} \setminus [\![\psi[\Gamma;\Delta]]\!]^{\underline{M}}_{\underline{R}^{-1}[\cdot]\circ\rho}$ are mapped to F, because we know that their counterparts in M do certainly not belong to the evaluation of ψ. Again, nothing can be said about the rest of the pairs, which are hence mapped to ?.

Finally, if ψ is not typable, then all pairs are mapped to ? since nothing can be said about the pairs in M.

As for over-approximations, we actually do not have to explicitly compute $\underline{R}^{-1}[\cdot]\circ\rho$ in practice.

We can define an under-approximated model checking procedure as follows.

Definition 4.6 (Under-check) *Let $\underline{R}$ be a simulation from $\underline{M}$ to M (that is $\underline{M} \sqsubseteq_{\underline{R}} M$) and ρ an assignment. The* under-approximated model check *(shortly,* under-check*) of $[\![\cdot]\!]^{M}_{\rho}$ in $\underline{M}$ via $\underline{R}$ is given by the function $\cdot \models_{\underline{R}} [\![\cdot]\!]^{M}_{\rho} : \Omega^{[\Gamma;\Delta]}_{M} \times \mathcal{F}^{[\Gamma;\Delta]} \to \mathbb{K}$, defined as*

$$p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = \begin{cases} ? & \text{if } \underline{R}^{-1}[p] = \emptyset \\ \bigvee\limits_{\underline{p}\in\underline{R}^{-1}[p]} \{\![\psi[\Gamma;\Delta]]\!\}^{\underline{R}}_{\rho}(\underline{p}) & \text{otherwise} \end{cases}$$

The next proposition, proved in Section 4.4, states the soundness of the under-check procedure.

Proposition 4.3 (Soundness of under-check) *Let $\underline{R}$ be a simulation from $\underline{M}$ to M (i.e. $\underline{M} \sqsubseteq_{\underline{R}} M$), $\psi[\Gamma;\Delta]$ a formula, and ρ an assignment. Then*

(i) $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$;

(ii) $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$.

We finally show how to combine the informations obtained from sets of under- and over-approximations.

Definition 4.7 (Approximated model checking) *Let $\{\underline{R}_0 \dots \underline{R}_n\}$ be simulations from the under-approximations $\{\underline{M}_0 \dots \underline{M}_n\}$ to M, and $\{\overline{R}_0 \dots \overline{R}_m\}$*

*be simulations from M to the over-approximations $\{\overline{M}_0 \dots \overline{M}_m\}$, i.e. $\underline{M_i} \sqsubseteq_{\underline{R_i}} M \sqsubseteq_{\overline{R_j}} \overline{M_j}$ for any $i \in \{0 \dots n\}$ and $j \in \{0 \dots m\}$. Then, the approximated model checking (*approximated check *in brief) of $[\![\cdot]\!]^M_\rho$ in $\{\underline{M}_0 \dots \underline{M}_n\}$ and $\{\overline{M}_0 \dots \overline{M}_m\}$ via $\{\underline{R}_0 \dots \underline{R}_n\}$ and $\{\overline{R}_0 \dots \overline{R}_m\}$ is the function $\cdot \models^{\{\overline{R}_0 \dots \overline{R}_m\}}_{\{\underline{R}_0 \dots \underline{R}_n\}} [\![\cdot]\!]^M_\rho : \Omega_M^{[\Gamma;\Delta]} \times \mathcal{F}^{[\Gamma;\Delta]} \rightharpoonup \mathbb{K}$, defined such that $p \models^{\{\overline{R}_0 \dots \overline{R}_m\}}_{\{\underline{R}_0 \dots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho$ is equal to*

$$\bigoplus_{0 \le j \le m} (p \models^{\overline{R}_j} [\![\psi[\Gamma;\Delta]]\!]^M_\rho) \oplus \bigoplus_{0 \le i \le n} (p \models_{\underline{R}_i} [\![\psi[\Gamma;\Delta]]\!]^M_\rho)$$

Note that, even if $\oplus$ is partial, the approximated check is well-defined since Propositions 4.2 and 4.3 ensure that we never have to combine contradictory results (e.g. $T \oplus F$). It is also easy to see that the soundness results of Propositions 4.2 and 4.3 allows us to conclude the soundness of the approximated check.

Theorem 4.1 (Soundness of approximated check) *Let $\psi[\Gamma;\Delta]$ be a formula, and ρ be an assignment. Let $\{\underline{R}_0 \dots \underline{R}_n\}$ be a set of simulations from the under-approximations $\{\underline{M}_0 \dots \underline{M}_n\}$ to M and $\{\overline{R}_0 \dots \overline{R}_m\}$ be a set of simulations from M to the over-approximations $\{\overline{M}_0 \dots \overline{M}_m\}$, that is $\underline{M_i} \sqsubseteq_{\underline{R_i}} M \sqsubseteq_{\overline{R_j}} \overline{M_j}$ for any $i \in \{0 \dots n\}$ and $j \in \{0 \dots m\}$. Then*

(i) $p \models^{\{\overline{R}_0 \dots \overline{R}_m\}}_{\{\underline{R}_0 \dots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$*;*

(ii) $p \models^{\{\overline{R}_0 \dots \overline{R}_m\}}_{\{\underline{R}_0 \dots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]^M_\rho$*.*

4.3 Dealing with untyped formulae

Notice how our approximated semantics and approximated check allow us to approximate the evaluation of any formula, even though this approximation may not be meaningful. Intuitively, we may obtain empty lower-bounds or unbounded upper-bounds as particular instances, that is when all the pairs are assigned to ? by the approximated semantics. Indeed, this is the case of formulae that cannot be typed with our type system. In order to obtain a more significant approximation also in the cases

of untyped formulae, we may try to enrich our approximated semantics by rules exploiting the structure of formulae.

We can thus extend both under- and over-approximated semantics (Definitions 4.3 and 4.5). In the following we present the enrichment for over-approximated semantics only, with the under-approximated case treated similarly.

Definition 4.8 (Enriched over-approximated semantics) *Let $\overline{R}$ be a simulation from M to $\overline{M}$ (i.e. $M \sqsubseteq_{\overline{R}} \overline{M}$), ρ an assignment, and $\{\![\cdot]\!\}^{\overline{R}}_{\rho}$ the over-approximated semantics of $[\![\cdot]\!]^{M}_{\rho}$ in $\overline{M}$ via $\overline{R}$. Then, the* enriched *over-approximated semantics of $[\![\cdot]\!]^{M}_{\rho}$ in $\overline{M}$ via $\overline{R}$ is given by the function ${}^{+}\{\!\{\cdot\}\!\}^{\overline{R}}_{\rho} : \mathcal{F}^{[\Gamma;\Delta]} \to (\Omega^{[\Gamma;\Delta]}_{\overline{M}} \to \mathbb{K})$ defined as*

$$
{}^{+}\{\!\{\psi[\Gamma;\Delta]\}\!\}^{\overline{R}}_{\rho} = \begin{cases} {}^{+}\{\!\{\psi_1[\Gamma;\Delta]\}\!\}^{\overline{R}}_{\rho} \vee {}^{+}\{\!\{\psi_2[\Gamma;\Delta]\}\!\}^{\overline{R}}_{\rho} & \text{if } \psi :_{\overline{R}} \bot \text{ and } \psi \equiv \psi_1 \vee \psi_2 \\ \neg\, {}^{+}\{\!\{\psi_1[\Gamma;\Delta]\}\!\}^{\overline{R}}_{\rho} & \text{if } \psi :_{\overline{R}} \bot \text{ and } \psi \equiv \neg\psi_1 \\ \{\![\psi[\Gamma;\Delta]]\!\}^{\overline{R}}_{\rho} & \text{otherwise} \end{cases}
$$

Intuitively, looking at the type system of Definition 4.2, and in particular at the rule regarding disjunction, we see that it is not possible to type disjunctions whose disjunct have (defined) opposite type, e.g. if we have $\psi_1 :_R d$ and $\psi_2 :_R d^{-1}$, then it is not possible to type their disjunction $\psi_1 \vee \psi_2$. With enriched approximated semantics we are instead able to handle those cases by dealing separately with ψ_1 and ψ_2, and then afterwards combining the obtained informations.

Example 4.4 *Consider again the predicate $\boldsymbol{p}(x, y)$ of Example 4.1, and the models M and $\overline{M}$ with $M \sqsubseteq_{\overline{R}} \overline{M}$ of Example 3.1 shown in Figure 3.2. In Example 4.3 we have seen that the over-approximated semantics of $[\![\boldsymbol{p}(u, v) \vee \neg\boldsymbol{p}(u, u)]\!]^{M}$ in $\overline{M}$ via $\overline{R}$ does not provide us any information. This happens because $\boldsymbol{p}(u, v) :_{\overline{R}} \rightarrow$ and $\neg\boldsymbol{p}(u, u) :_{\overline{R}} \leftarrow$, and hence $\boldsymbol{p}(u, v) \vee \neg\boldsymbol{p}(u, u) :_{\overline{R}} \bot$. In particular, as depicted in the third column of the first group of lines of Figure 4.3, all the pairs in $\Omega^{[z;\emptyset]}_{\overline{M}}$ are assigned to ?.*

The enriched over-approximated semantics is instead more significative. Following Definition 4.8, we evaluate separately the (enriched) over-approximated semantics of the two disjuncts (first and second column of the first group of lines of Figure 4.3), and then combine the so-obtained informations as shown

in the last column of Figure 4.3 with the Kleene's logic disjunction. Considering for example the pair $(\overline{w}_0, \lambda)$*, we have* ${}^{+}\{\!|\boldsymbol{p}(u,v)[z;\emptyset]|\!\}^{\overline{R}}_{\rho}((\overline{w}_0,\lambda)) = F$*, and* ${}^{+}\{\!|\neg\boldsymbol{p}(u,u)[z;\emptyset]|\!\}^{\overline{R}}_{\rho}((\overline{w}_0,\lambda)) = T$*, from which we obtain* ${}^{+}\{\!|\boldsymbol{p}(u,v))\vee\neg\boldsymbol{p}(u,u)[z;\emptyset]|\!\}^{\overline{R}}_{\rho}((\overline{w}_0,\lambda)) = F \vee T$*, which evaluates in* T*. Considering instead* $(\overline{w}_1, (z \mapsto e, \lambda_2))$*, we have* ${}^{+}\{\!|\boldsymbol{p}(u,v))\vee\neg\boldsymbol{p}(u,u)[z;\emptyset]|\!\}^{\overline{R}}_{\rho}((\overline{w}_1,(z\mapsto e,\lambda_2))) = ? \vee T$ *which evaluates in* T*.*

Definition 4.9 (Enriched over-check) *Let* $\overline{R}$ *be a simulation from* M *to* $\overline{M}$ *(i.e.* $M \sqsubseteq_{\overline{R}} \overline{M}$*) and* ρ *an assignment. Then, the* enriched over-approximated model check *(shortly,* enirched over-check*) of* $[\![\cdot]\!]^M_\rho$ *in* $\overline{M}$ *via* $\overline{R}$ *is given by the function* $\cdot {}^{+}\!\models^{\overline{R}} [\![\cdot]\!]^M_\rho$*, defined as*

$$p \;{}^{+}\!\models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = \bigvee_{\overline{p}\in\overline{R}(p)} {}^{+}\{\!|\psi[\Gamma;\Delta]|\!\}^{\overline{R}}_{\rho}(\overline{p})$$

We may enrich both the under-approximated semantics and under-approximated check exactly in the same way, and thus straightforwardly define an enriched version "$\cdot {}^{+}\!\models^{\{\overline{R}_j\}}_{\{\underline{R}_i\}} [\![\cdot]\!]^M_\rho$" of the approximated checking by replacing approximated under- and over-checks with their enriched variants. It is also easy to verify that enriched approximated check is sound.

4.4 Soundness proofs

In this section we prove the three Propositions 4.1, 4.2 and 4.3, implying consequently also Theorem 4.1.

We first prove that Proposition 4.1 holds, stating the soundness of our type system with respect to Definition 4.1. Namely, we show that if our type system assigns a type to a formula, then it is coherent with the preservation and/or reflection of the formula.

Proposition 4.1 (Type system soundness). *Let* R *be a simulation from* M *to* M' *(i.e.* $M \sqsubseteq_R M'$*) and* ψ *a formula. Then*

(i) $\psi :_R \rightarrow$ *implies* $\psi :_R \Rightarrow$*;*

(ii) $\psi :_R \leftarrow$ *implies* $\psi :_R \Leftarrow$*;*

(iii) $\psi :_R \leftrightarrow$ *implies* $\psi :_R \Leftrightarrow$*.*

Proof: We focus on the points (i) and (ii), since point (iii) follows directly from them. Rephrasing Definition 4.1, what we have to show is: for every $(w_1, \phi_1, w_1') \in R$, if (i) $\psi :_R \rightarrow$, then $(w_1, \sigma_{w_1}) \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$ implies $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$; while, if (ii) $\psi :_R \leftarrow$, then $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\psi[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$ implies $(w_1, \sigma_{w_1}) \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$. The proof is done on structural induction on ψ.

The proposition trivially holds for the cases tt, Z and $\psi_1 \vee \psi_2$.

[$\psi \equiv \epsilon \in_\tau Y :_R \rightarrow$] Following the semantics, we have $[\![\epsilon \in_\tau Y[\Gamma;\Delta]]\!]^M_\rho = \{(w,\sigma) \in \Omega_M^{[\Gamma;\Delta]} \mid \sigma(\epsilon) \text{ is defined and } \sigma(\epsilon) \in \sigma(Y)\}$. Hence there exists an $a : \tau \in d(w_1)$ such that $a = \sigma_{w_1}(\epsilon)$, and $a \in \sigma_{w_1}(Y)$. Clearly, $\phi_1(a) \in \phi_1 \circ \sigma_{w_1}(Y)$. A term ϵ is a variable or an operation applied to a term. From our type system we know that $\epsilon \in_\tau Y$ has type $\rightarrow$ for $\tau_{total}\ R$, which, together with the fact that morphisms preserve terms' operations, allows us to conclude $\phi_1 \circ \sigma_{w_1}(\epsilon) = \phi_1(a)$.

[$\psi \equiv \epsilon \in_\tau Y :_R \leftarrow$] From $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\epsilon \in_\tau Y[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho} \equiv \{(w',\sigma') \in \Omega_{M'}^{[\Gamma;\Delta]} \mid \sigma'(\epsilon) \text{ is defined and } \sigma'(\epsilon) \in \sigma'(Y)\}$, we know that there exists an $a' : \tau \in d'(w_1')$ with $a' = \phi_1 \circ \sigma_{w_1}(\epsilon)$, and $a' \in \phi_1 \circ \sigma_{w_1}(Y)$. From our type system we know that R is $\tau_{bijective}$, hence there exists an $a \in d(w_1)$ such that $\phi_1(a) = a'$. Clearly, $a \in \sigma_{w_1}(Y)$, and $\sigma_{w_1}(\epsilon) = a$.

[$\psi \equiv \neg\psi' :_R \rightarrow$] We want to prove that $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\neg\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$, knowing that $(w_1, \sigma_{w_1}) \in [\![\neg\psi'[\Gamma;\Delta]]\!]^M_\rho \equiv \Omega_M^{[\Gamma;\Delta]} \setminus [\![\psi'[\Gamma;\Delta]]\!]^M_\rho$. In particular, $(w_1', \phi_1 \circ \sigma_{w_1})$ may belong either to $[\![\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$ or to $\Omega_{M'}^{[\Gamma;\Delta]} \setminus [\![\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$. By absurd consider $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$. From our type system we know that $\psi' :_R \leftarrow$, which, by induction hypothesis, implies $(w_1, \sigma_{w_1}) \in [\![\psi'[\Gamma;\Delta]]\!]^M_\rho$, obtaining a contradiction.

[$\psi \equiv \neg\psi' :_R \leftarrow$] This case is specular to the $\psi :_R \rightarrow$ one.

[$\psi \equiv \exists_\tau x.\psi' :_R \rightarrow$] Following the semantics, we know that $(w_1, \sigma_{w_1}) \in [\![\exists_\tau x.\ \psi'[\Gamma;\Delta]]\!]^M_\rho = 2^{\downarrow_x}(\{(w,\sigma) \in [\![\psi'[\Gamma,x;\Delta]]\!]^M_{2^{\uparrow_x}\circ\rho} \mid \sigma(x) \text{ is defined}\})$.

From the type system we know that R is τ_{total}, hence $\phi_1 \circ \sigma_{w_1}(x)$ is defined if and only if $\sigma_{w_1}(x)$ is defined, allowing to reduce the problem in: showing that $(w_1, \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho})$ implies $(w_1', \phi_1 \circ \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^{M'}_{2^{\uparrow_x} \circ \rho'})$. We also know that $\psi' :_R \rightarrow$, hence, by induction hypothesis, $(w_1, \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho}$ implies $(w_1', \phi_1 \circ \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^{M'}_{2^{\uparrow_x} \circ \rho'}$, with $\sigma_{2w_1} \in \Omega^{[\Gamma, x; \Delta]}_{w_1}$. Noting that $2^{\uparrow_x}$ and $2^{\downarrow_x}$ are monotone, we have that $(w_1, \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho})$ implies $2^{\uparrow_x}((w_1, \sigma_{w_1})) \subseteq [\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho}$, and $(w_1, \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho}$ implies $2^{\downarrow_x}((w_1, \sigma_{2w_1})) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho})$. It is now easy to see that for every $(w_1, \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho})$ there exists a $(w_1, \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho}$ such that $(w_1, \sigma_{w_1}) = 2^{\downarrow_x}((w_1, \sigma_{2w_1}))$, for which in turn there exists a $(w_1', \phi_1 \circ \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^{M'}_{2^{\uparrow_x} \circ \rho'}$ such that $(w_1', \phi_1 \circ \sigma_{w_1}) = 2^{\downarrow_x}((w_1', \phi_1 \circ \sigma_{2w_1}))$, closing this case.

[$\psi \equiv \exists_\tau x.\psi' :_R \leftarrow$] What we want to prove is that $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\exists_\tau x.\ \psi'[\Gamma; \Delta]]\!]^{M'}_{R \circ \rho}$ implies $(w_1, \sigma_{w_1}) \in [\![\exists_\tau x.\ \psi'[\Gamma; \Delta]]\!]^M_\rho$. Clearly, $\phi_1 \circ \sigma_{w_1}(x)$ can be defined only if also $\sigma_{w_1}(x)$ is defined, hence we can again reduce the problem to $(w_1', \phi_1 \circ \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^{M'}_{2^{\uparrow_x} \circ \rho'})$ implies $(w_1, \sigma_{w_1}) \in 2^{\downarrow_x}([\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho})$. Moreover, from the type system we know that $\psi' :_R \leftarrow$, hence, by induction hypothesis, $(w_1', \phi_1 \circ \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^{M'}_{2^{\uparrow_x} \circ \rho'}$ implies $(w_1, \sigma_{2w_1}) \in [\![\psi'[\Gamma, x; \Delta]]\!]^M_{2^{\uparrow_x} \circ \rho}$, with $\sigma_{2w_1} \in \Omega^{[\Gamma, x; \Delta]}_{w_1}$. The rest of the proof is similar to the ($\exists_\tau x.\psi' :_R \rightarrow$) case.

[$\psi \equiv \exists_\tau Y.\psi' :_R \leftrightarrow$] The proofs are similar to the first-order cases.

[$\psi \equiv \Diamond\psi' :_R \rightarrow$] We are trying to prove that for every $(w_1, \phi_1, w_1') \in R$, if $\psi :_R \rightarrow$, then $(w_1, \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma; \Delta]]\!]^M_\rho$ implies $(w_1', \phi_1 \circ \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma; \Delta]]\!]^{M'}_{R \circ \rho}$. From the semantics, $(w_1, \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma; \Delta]]\!]^M_\rho$ implies the existence of a $w_2 \in W$ such that $w_1 \overset{cr}{\rightsquigarrow} w_2$ and $(w_2, cr \circ \sigma_{w_1}) \in [\![\psi'[\Gamma; \Delta]]\!]^M_\rho$. Following Def. 3.1, there exists a transition $w_1' \overset{cr'}{\rightsquigarrow} w_2'$, with (at least) an $(w_2, \phi_2, w_2') \in R$ and $\phi_2 \circ cr = cr' \circ \phi_1$. We apply the induction hypothesis: $(w, \phi, w') \in R$ and $(w, \sigma_w) \in [\![\psi'[\Gamma; \Delta]]\!]^M_\rho$ implies $(w', \phi \circ \sigma_w) \in [\![\psi'[\Gamma; \Delta]]\!]^{M'}_{\rho'}$. Hence, $(w_2', \phi_2 \circ cr \circ \sigma_{w_1}) \in [\![\psi'[\Gamma; \Delta]]\!]^{M'}_{\rho'}$. All

remains to prove is that $\phi_2 \circ cr \circ \sigma_{w_1} = cr' \circ \phi_1 \circ \sigma_{w_1}$, which follows from $\phi_2 \circ cr = cr' \circ \phi_1$.

[$\psi \equiv \Diamond\psi' :_R\leftarrow$] From the type system of Definition 4.2 we know that R is an iso-bisimulation, hence $R^{-1} \equiv \{(w', \phi^{-1}, w) \in R^{-1} \mid (w, \phi, w') \in R\}$ is defined, and is a simulation from M' to M. What we want to prove is $(w'_1, \phi_1 \circ \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$ implies $(w_1, \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma;\Delta]]\!]^{M}_{\rho}$. From the $\Diamond\psi' :_R\rightarrow$ case we know that $(w'_1, \phi_1 \circ \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma;\Delta]]\!]^{M'}_{R\circ\rho}$ implies $(w_1, \phi_1^{-1} \circ \phi_1 \circ \sigma_{w_1}) \in [\![\Diamond\psi'[\Gamma;\Delta]]\!]^{M}_{R^{-1}\circ R\circ\rho}$. It is easy to see that for a bisimulation R, $\phi_1^{-1} \circ \phi_1 \circ \sigma_{w_1} = \sigma_{w_1}$ and $R^{-1} \circ R \circ \rho = \rho$, closing the case.

[$\psi \equiv \mu Z\psi' :_R\leftrightarrow$] Consider the functions $F = \lambda Y.[\![\psi'[\Gamma;\Delta]]\!]^{M}_{\rho[^Y/_Z]}$, and $F' = \lambda Y'.[\![\psi'[\Gamma;\Delta]]\!]^{M'}_{\rho'[^{Y'}/_Z]}$ as done in the proof of Proposition 2.1. By definition, $[\![\mu Z.\psi'[\Gamma;\Delta]]\!]^{M}_{\rho} = \mathit{lfp}(F)$. We are proving $(w_1, \phi_1, w'_1) \in R$ implies $(w_1, \sigma_{w_1}) \in \mathit{lfp}(F)$ iff $(w'_1, \phi_1 \circ \sigma_{w_1}) \in \mathit{lfp}(F')$. We apply the induction hypothesis on ψ': for any Y,Y' with $Y' = R\circ Y$, then $(w, \phi, w') \in R$ implies $(w, \sigma_w) \in F(Y)$ iff $(w', \phi \circ \sigma_w) \in F'(Y')$, from which we have $F'(Y') = R \circ F(Y)$. From Kleene's theorem, $\mathit{lfp}(F) = \mathit{sup}(F^n(\emptyset) \mid n \in \mathbb{N})$, computable as the first Y_n such that $Y_n = Y_{n-1}$, with $Y_0 = \emptyset$, and $Y_i = F(Y_{i-1})$. Clearly $\emptyset = R \circ \emptyset$, implying $Y'_1 = F'(\emptyset) = R \circ F(\emptyset) = R \circ Y_1$. Iterating, $F'(lfp(F')) = R \circ F(lfp(F))$, closing the case.

□

We now prove that Proposition 4.2 holds, stating the soundness of our over-approximated model checking procedure.

Proposition 4.2 (Soundness of over-check). *Let $\overline{R}$ be a simulation from M to $\overline{M}$ (i.e. $M \sqsubseteq_{\overline{R}} \overline{M}$), $\psi[\Gamma;\Delta]$ a formula, and ρ an assignment. Then*

(i) $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$;

(ii) $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho}$.

Proof: We first focus on case (i). From Definition 4.4, $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^{M}_{\rho} = T$ iff there exists a $\overline{p} \in \overline{R}(p)$, such that $\{[\![\psi[\Gamma;\Delta]]\!\}^{\overline{R}}_{\rho}(\overline{p}) = T$. This in turn implies that $\psi :_{\overline{R}}\leftarrow$ and $\overline{p} \in [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$. Finally, from Definition 4.1 and

Proposition 4.1, we can conclude that all the pairs in $\overline{R}^{-1}[\overline{p}]$ (including p) belong to $[\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

We now consider case (ii). From Definition 4.4, $p \models^{\overline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = F$ iff there exists a $\overline{p} \in \overline{R}(p)$, such that $\{[\psi[\Gamma;\Delta]]\}^{\overline{R}}_\rho(\overline{p}) = F$, and does not exist any pair $\overline{p}' \in \overline{R}(p)$, such that $\{[\psi[\Gamma;\Delta]]\}^{\overline{R}}_\rho(\overline{p}) = T$. This in turn implies that $\psi :_{\overline{R}} \rightarrow$ and $\overline{p} \notin [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$. From Definition 4.1 and Proposition 4.1, we know that $\overline{R}([\![\psi[\Gamma;\Delta]]\!]^M_\rho) \subseteq [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$. Finally, since $\overline{p} \notin [\![\psi[\Gamma;\Delta]]\!]^{\overline{M}}_{\overline{R}\circ\rho}$, then no assignment pair in $\overline{R}^{-1}[\overline{p}]$ (including p) belongs to $[\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

□

We now prove that Proposition 4.3 holds, stating the soundness of our under-approximated model checking procedure.

Proposition 4.3 (Soundness of under-check). *Let $\underline{R}$ be a simulation from $\underline{M}$ to M (i.e. $\underline{M} \sqsubseteq_{\underline{R}} M$), $\psi[\Gamma;\Delta]$ a formula, and ρ an assignment. Then*

(i) $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$;

(ii) $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

Proof: We first focus on case (i). From Definition 4.6, $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$ iff there exists a $\underline{p} \in \underline{R}^{-1}[p]$, such that $\{[\psi[\Gamma;\Delta]]\}^{\underline{R}}_\rho(\underline{p}) = T$. This in turn implies that $\psi :_{\underline{R}} \rightarrow$ and $\underline{p} \in [\![\psi[\Gamma;\Delta]]\!]^{\underline{M}}_{\underline{R}^{-1}[\cdot]\circ\rho}$. Finally, from Definition 4.1 and Proposition 4.1, we can conclude that all the assignment pairs in $\underline{R}(\underline{p})$ (including p) belong to $[\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

We now consider case (ii). From Definition 4.6, $p \models_{\underline{R}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = F$ iff there exists a $\underline{p} \in \underline{R}^{-1}[p]$, such that $\{[\psi[\Gamma;\Delta]]\}^{\underline{R}}_\rho(\underline{p}) = F$, and does not exist any pair $\underline{p}' \in \underline{R}^{-1}[p]$, such that $\{[\psi[\Gamma;\Delta]]\}^{\underline{R}}_\rho(\underline{p}') = T$. This in turn implies that $\psi :_{\underline{R}} \leftarrow$ and $\underline{p} \notin [\![\psi[\Gamma;\Delta]]\!]^{\underline{M}}_{\underline{R}^{-1}[\cdot]\circ\rho}$. From Definition 4.1 and Proposition 4.1, we know that $\underline{R}^{-1}[[\![\psi[\Gamma;\Delta]]\!]^M_\rho] \subseteq [\![\psi[\Gamma;\Delta]]\!]^{\underline{M}}_{\underline{R}^{-1}[\cdot]\circ\rho}$. Finally, since $\underline{p} \notin [\![\psi[\Gamma;\Delta]]\!]^{\underline{M}}_{\underline{R}^{-1}[\cdot]\circ\rho}$, then $\underline{R}(\underline{p}) \cap [\![\psi[\Gamma;\Delta]]\!]^M_\rho = \emptyset$, hence, no pair in M similar to $\underline{p}$ (including p) belongs to $[\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

□

Finally, we now prove that Theorem 4.1 holds, stating the soundness of our approximated model checking procedure.

Theorem 4.1 (Soundness of approximated check). *Let $\psi[\Gamma;\Delta]$ be a formula, and ρ be an assignment. Let $\{\underline{R}_0 \dots \underline{R}_n\}$ be a set of simulations from the under-approximations $\{\underline{M}_0 \dots \underline{M}_n\}$ to M and $\{\overline{R}_0 \dots \overline{R}_m\}$ be a set of simulations from M to the over-approximations $\{\overline{M}_0 \dots \overline{M}_m\}$, that is $\underline{M_i} \sqsubseteq_{\underline{R_i}} M \sqsubseteq_{\overline{R_j}} \overline{M_j}$ for any $i \in \{0 \dots n\}$ and $j \in \{0 \dots m\}$. Then*

(i) $p \models^{\{\overline{R}_0 \cdots \overline{R}_m\}}_{\{\underline{R}_0 \cdots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$ *implies* $p \in [\![\psi[\Gamma;\Delta]]\!]^M_\rho$;

(ii) $p \models^{\{\overline{R}_0 \cdots \overline{R}_m\}}_{\{\underline{R}_0 \cdots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = F$ *implies* $p \notin [\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

Proof: We first focus on case (i). From Definition 4.7, $p \models^{\{\overline{R}_0 \cdots \overline{R}_m\}}_{\{\underline{R}_0 \cdots \underline{R}_n\}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$ iff there exist at least an $\overline{R}_j \in \{\overline{R}_0 \dots \overline{R}_m\}$ such that $p \models^{\overline{R_j}} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$, or an $\underline{R}_i \in \{\underline{R}_0 \dots \underline{R}_n\}$ such that $p \models_{\underline{R}_i} [\![\psi[\Gamma;\Delta]]\!]^M_\rho = T$. From Proposition 4.2 and Proposition 4.3 this implies that p belongs to $[\![\psi[\Gamma;\Delta]]\!]^M_\rho$.

A specular reasoning can be done for case (ii).

Moreover, it may be worth to note that from Propositions 4.2 and 4.3 we know that we never have to combine contradictory results (e.g. $T \oplus F$). □

Part II

Tool support

The use of visual specification formalisms is nowadays diffused in almost the whole spectrum of software and hardware development activities. In the particular case of analysis and verification activities, visual specifications are complemented with appropriate property specification languages and tools for checking and verifying properties. An example are graph grammars, temporal graph logics and the corresponding verification tools, which are used to reason about the possible transformations in a graph topology.

In this part we present our first steps towards the development of a tool support for our approach, aiming in particular at assessing the feasibility of our approach, and at preparing the ground for an efficient framework for verifying interesting properties of systems with dynamically evolving structure, that is where system components and their interrelations may vary over time (e.g. via (de)allocation, merging and renaming of components, and creation and breaking of relationships). The current implementation is tailored to our needs, leaving for future works concerns about efficiency and usability.

We developed our tool in Maude (CDE+07), which we introduce in Chapter 5, a high-performance formal language and execution environment based on equational and rewriting logic.

In our implementation we use algebras of a given signature to model the internal structure of system states, and sets of behavioural rules (i.e. rewrite rules) to specify systems dynamics. In Chapter 6 we present an implementation of SPO-like rewriting over algebras of a given signature, encoded in Maude's conditional term rewriting. This allows us to compositionally specify concurrent systems, in the sense that we may specify the *global* behaviour of a system by specifying the *local* behaviour of its components. Moreover, it also allows to keep trace of the evolution of system components across the system evolution. In the case in which we fix the signature of graphs, then system specifications are essentially graph transformation systems.

Basing on this machinery, in Chapter 6 we also introduce a prototypal model checker for finite counterpart models that can be used to check quantified μ-calculus formulae against system specifications. As far as we

know, our tool is one of the few model checkers for a quantified μ-calculus and one of the few ones inspired by Counterpart Theory, allowing for a neat analysis of the evolution of individual components.

Then, Chapter 7 shows how we extended our tool to support our $\mathtt{c}$-reductions approach, and discusses some state *canonizers* currently implemented in the tool. As discussed in Section 3.3, $\mathtt{c}$-reductions are a technique that we developed to reduce (Kripke and) counterpart models in behaviourally equivalent (i.e. bisimilar) ones. Interestingly, many infinite counterpart models, and in particular the class of resource-bounded ones, can be reduced to finite counterpart models. This, together with the fact that (from Chapters 3, 4) every formula of our logic is preserved and reflected in bisimilar models, allows us to use our model checker to analyze possibly infinite-state systems by checking formulae against their $\mathtt{c}$-reductions.

Finally, in Chapter 8 we validate our tool against some systems taken from the literature. Namely, the leader election system presented in Section 2.1, and an infinite-state variant of the well-known dining philosophers problem (Dij71) as presented in (DRK02). We would like to stress the fact that the aim of Chapter 8 is not that of demonstrate the efficiency and scalability of our prototypal tool framework. We instead rather want to focus on the wide applicability of our tool support, on the expressivity of our logic, and on the advantages brought by our efforts on state-space reduction techniques and sound approximated model checking.

Unfortunately, we currently do not fully support our sound approximated model checking. So far, we only deal with bisimilar models reduced following our $\mathtt{c}$-reductions approach. However, even this partial support is quite interesting, since from the results of Chapters 3, 4 we know that we can freely analyze $\mathtt{c}$-reduced models, rather than the original ones, meaning that we can reason on systems of greater size, and on infinite models, in case their reduction is finite.

Chapter 5

A gentle introduction to rewriting logic and Maude

We have decided to rely our machinery on rewriting logic (Mes12), and on its instantiation in Maude (CDE+07), due to:

- its well-developed theory based on the idea of computation as logical deduction,
- its expressiveness and generality witnessed by notable encodings of programming languages, and
- its performant, easily extensible tool support.

In this chapter we introduce some concepts related to *rewriting logic*. In Section 5.1 we provide an informal discussion in order to improve the accessibility of the more detailed one provided in Section 5.2.

5.1 Informal discussion

We already discussed the fact that worlds of our models are labelled with algebras of a given signature Σ, which fixes the set of sorts and of operation names used to represent worlds' internal structure. Actually, in our tool the worlds of our models are labelled with an explicit representation

of such Σ-algebras, defined by means of sets of elements, and by sets of operations (i.e. functions) among elements.

If we fix Σ as the signature provided in Example 2.1, then Σ-algebras are actually unlabelled directed graphs. An example of explicit representation in our framework of a Σ-algebra (for Σ the signature of unlabelled directed graphs) is

```
e(3) e(4) n(3) n(4)
s: ({e(3)} |-> n(3), {e(4)} |-> n(4))
t: ({e(3)} |-> n(4), {e(4)} |-> n(3))
```

Intuitively, the represented graph has two edges (`e(3)` and `e(4)`) and two nodes (`n(3)` and `n(4)`). While the operations source (`s:`) and target (`t:`) are specified such that `n(3)` is the source of `e(3)` and the target of `e(4)`, while `n(4)` is the source of `e(4)` and the target of `e(3)`. We obtain hence a graph connected in a ring topology, and in particular $\mathbf{G}_1$ of Figure 2.2.

In order to explicitly represent Σ-algebras, we resort to an extension of the predefined object-like signature of `Configurations` provided by Maude, where a configuration is a set of objects. More precisely, such signature provides the sort `Configuration` which we use to represent Σ-algebras. A term with sort `Configuration` can be the empty configuration `none`, a singleton object (in our case elements of the algebra), as the sort `Object` is declared as a subsort of `Configuration`, or the set union (denoted with juxtaposition) of several objects. The signature provides also the sort of messages (`Msg`), which we however do not exploit.

In particular, we defined the sorts `Element` and `Operation` as subsorts of `Object`. Indeed, `Element` can be thought of as a kind of supersort for all the sorts which will be defined by Σ to represent its elements, in the sense that every sort defined by Σ to represent the elements of a state of a system (e.g. edges and nodes for the case in which Σ is the signature of graphs) will be defined as subsorts of `Element`. A Σ-operation (e.g. source and target in the case in which Σ is the signature of graphs) is instead represented by a term with sort `Operation`, namely a mapping from an element (or sequences of n elements for n-ary operations) to another element.

Such signature of *extended configurations* will be further refined for every modelled system, by listing the set of sorts and of operation names of the considered Σ.

Last, the dynamics of a system are specified as sets of rewrite rules in SPO-like style, encoded on top of Maude's term rewriting. This chapter does not discuss in detail rules, as they are treated in Section 6.1 with the help of a running example.

Then, it is easy to understand that the structure labelling a world, i.e. a Σ-algebra representing the internal structure of a state of a system, is actually a term of sort `Configuration` of an extension of the Maude `Configurations`, which are further extended by the system-specific signature Σ. In particular, a representation of a Σ-algebra is composed by a set of terms whose sort is subsort of `Element` (i.e. `Edge` and `Node`), and by a set of terms with sort `Operation`.

This is discussed and exemplified in Section 6.1, where we provide a step-by-step specification of the leader election system considered in Section 2.1.

To conclude this informal discussion, a system specification is given by three components: the signature Σ of the algebraic structures used to represent its states (and that will label the worlds of our models), a specification of the initial state of the system (i.e. a Σ-algebra), and a set of rewriting rules in SPO-like style encoded in Maude's term rewriting. However, system specifications are actually given by means of Maude's operations (which can be thought of as functions of other programming languages), equations and rules. Intuitively, a system specification is in fact a *rewrite theory*, which is then *executed* by Maude in order to generate the counterpart model representing the state-space of the considered system. Hence, as we will see, in our framework counterpart models are terms generated by a rewrite theory.

5.2 Detailed discussion

A *rewrite theory* $\mathcal{R}$ is a tuple $\langle \Psi, E, R \rangle$ where

- Ψ is a signature, specifying the basic syntax (function symbols) and type machinery (sorts, kinds and subsorting) for terms, e.g. counterpart models and their constituents (like worlds, Σ-algebras, accessibility relation, etc);

- E is a set of possibly conditional equations which induce equivalence classes of terms, and possibly conditional membership predicates which refine the typing information;

- R is a set of possibly conditional term rewrite rules.

The Maude framework (CDE$^+$07) instantiates rewriting logic (Mes12) providing a language for describing rewrite theories by means of *modules*, and a tool built upon a rewrite engine for executing and analysing them. A very powerful functionality offered by Maude is that of *theory composition*, meaning that a module can import, extend and compose other modules.

Our implementation strongly depends on this composition functionality. In fact, it is important to notice that the signature Σ of the structures representing the states of our systems, and the signature Ψ of a rewrite theory are two distinct concepts.

Intuitively, our tool can be thought of as a specification of a *generic* rewrite theory of counterpart models, parametric in Σ and in the system specification. This means that $\mathcal{R}$ provides the machinery and the signature of counterpart models, which users refine and extend with their system specifications, providing informations about the structures labelling the worlds of the model, an initial state of the system, and its dynamics. However, the following discussion holds for any rewrite theory.

Once again, this concept is illustrated in Section 6.1, where we provide and thoroughly discuss a step-by-step specification of the leader election system considered in Section 2.1, and shown in Figures 2.1 and 2.3.

In the following, we shall use Maude's syntax, introducing the syntactic ingredients as we use them. Moreover, in order to distinguish from Maude's signatures and operations, we shall use the terms "Σ-operation" and "Σ-element" for referring to the explicit representations of the structures labelling the worlds of our models.

The signature Ψ and the equations E of a rewrite theory form a *membership equational theory* $\langle \Psi, E \rangle$, whose initial algebra is $T_{\Psi/E}$. Indeed, $T_{\Psi/E}$ is the state-space of a rewrite theory, i.e. its generated elements (e.g. in our case counterpart models) are equivalence classes of Ψ-terms t modulo the least congruence induced by the axioms in E (denoted by $[t]_E$ or just t for short).

This means that several counterpart models could be generated out of a rewrite theory, given a system specification. Intuitively, a counterpart model is generated by exploring the state-space of a system, and assigning system states (i.e. Σ-algebras) to its worlds. Depending on the strategy of rule application, the generated models may be syntactically different, yet clearly all isomorphic.

Maude operators, which can be thought of as elements constructors, and as the equivalent of functions of other programming languages, are declared in Maude notation as

```
op f : TL -> T [a] .
```

Where `f` is the name of the operator (possibly given in mixfix notation where argument placeholders are denoted with underscores), `TL` is a (possibly empty, blank separated) list of sorts defining the domain of the operator, and `T` is the sort of the co-domain of the operator. Finally, `a` is a set of equational attributes (e.g. associativity, commutativity).

For example, without giving too many details, the operator to define a counterpart model (following Definition 2.3) can be sketched as

```
op _ _ _ : Set{World} Map{World,Configuration} Set{RCEntry}
           -> CounterpartModel .
```

Hence a counterpart model is defined as the juxtaposition of a term with sort `Set{World}`, that is the set of worlds of the model, by a term with sort `Map{World,Configuration}`, that is an associative array assigning a term with sort `Configuration` (which in our framework will stand for a Σ-algebra) to every world of the model, and by a term with sort `Set{RCEntry}`, that is the accessibility relation of the model, composed by a set of entries relating a target and a source state, via a counterpart relation.

It may be interesting to notice that `Set` and `Map` are predefined parametric *data structures*, which, exactly as *c++ Templates* and *Java Generics*, can be instantiated for a specific sort. For example, `Set{World}` stands for the sort of *set of worlds*, for `World` defined as

```
sort World .
op w : Nat -> World .
```

The main equations of the theories that we use allow us to treat collections of Σ-elements and Σ-operations as multisets, i.e. modulo associativity, commutativity, and identity (all treated as equational attributes), therefore axiomatising their algebraic-theoretic nature.

Equations that cannot be declared as equational attributes must be treated as functions defined by a set of confluent and terminating (possibly conditional) equations of the form

```
ceq t = t' if c
```

where `t` and `t'` are Ψ-terms, and `c` is an application condition. When the application condition is vacuous, the simpler syntax `eq t = t'` can be used.

Roughly, an equation `ceq t = t' if c` can be applied to a term `t''` if we find a match for `t` at some place in `t''` such that `c` holds (after the application of the substitution induced by the match). The effect is that of substituting the matched part with `t'` (after the application of the substitution induced by the match).

One major advantage of Maude is that it includes tools for checking confluence, termination and completeness of equational logic specifications.

A membership predicate is of the form

```
cmb t : T if c
```

where `t` is a Ψ-term of some supersort `T'` of `T` and `c` is a predicate over `t` conditioning the membership statement. When the application condition is vacuous, the simpler syntax `mb t : T` can be used.

Roughly, a membership predicate states that if we are able to match a term `t'` with `t` such that `c` holds then `t'` has sort `T`. Hence, membership predicates provide a subtyping mechanism that we can use, for instance, to check conformance with respect to a certain meta-model (e.g. a typegraph).

Rewrite rules are of the form

```
crl t => t' if c
```

where `t` and `t'` are Ψ-terms, and `c` is an application condition, namely either a predicate on the terms involved in the rewrite, or further rewrites (whose result can be used), or membership predicates. When the application condition is vacuous, the simpler syntax `rl t => t'` can be used.

Matching and rule application are similar to the case of equations with the main difference being that rules are not required to be confluent and terminating (as they represent possibly non-deterministic concurrent actions). Equational simplification has precedence over rule application in order to simulate rule application modulo equational equivalence. Finally, as previously mentioned, rewrite rules can be used to program the behaviour of a system in a declarative way (e.g. in graph transformation style).

Σ-algebras as object collections. We summarize the previously mentioned algebra of object collections (`Configurations`) that is used to represent system states.

In our setting, a system configuration (i.e. a system state) is a collection of Σ-elements and of explicit representations of Σ-operations among them. Maude provides a signature for representing system states as multiset of objects, called `Configurations` (CDE$^+$07), which we partly use and extend (e.g. with the previously mentioned sorts `Element` and `Operation`, defined as subsorts of objects).

Each Σ-element has one of the sorts provided by Σ, which has to be defined as subsort of `Element`, and represents an entity (an individual component of a system state), while Σ-operations are terms with sort

`Operation`, which can be thought of as associative arrays mapping sequences of Σ-elements (n for n-ary operations) in another Σ-element.

For example, as shown in Section 6.1, fixing the signature of graphs for Σ, we have sorts `Edge` (whose elements are defined as `e(0),e(1)`), `Node` (whose elements are defined as `n(0),n(1)`), and the operations `source` and `target` mapping an `Edge`, respectively, in its source and target `Node`.

Sorts, ad-hoc constructors of Σ-elements (i.e. Maude operations to build them) and names of Σ-operations will be provided in the system-specific code. A possible example of constructor for the sort `Edge` is `op e : Nat -> Edge`, allowing to generate edges providing natural numbers, e.g. `e(0)` and `e(1)` are examples of edges.

In Section 8.2 we will see how it is possible to encode elements with attributes in our framework (e.g. the status of a philosopher in the well-known dining philosophers problem).

To conclude, configurations representing Σ-algebras are essentially sets of objects (Σ-elements and Σ-operations). As sketched in Section 5.1, the sort for configurations is called `Configuration` and its constructors are the empty configuration (`none`), singleton objects (in our case terms with sort `Element`, subsort of `Object`), as `Object` is in turn declared as subsort of `Configuration`, and set union (denoted with juxtaposition).

Finally, in order to distinguish a system state from the collection of Σ-elements and Σ-operations that forms it, we wrap configurations into a `State` with the operation `<< _ >> : Configuration -> State`.

Chapter 6

An explicit-state counterpart model checker for finite models

In this chapter we present a prototypal tool for the verification of *finite* counterpart models, that is counterpart models whose set of worlds is finite, and whose algebras have finite domains. In addition we shall consider *finite* formulae-in-context, i.e. finite formulae with finite contexts. As stated in Section 2.6.2, our model checking problem is decidable under those assumptions.

The tool allows to check quantified temporal properties based on our counterpart-like semantics for second-order μ-calculi. The tool can be considered as an instantiation of our approach to counterpart semantics which, as previously stated, allows for a neat handling of the evolution of system components in systems with dynamic structure.

We implemented the tool in the Maude framework (CDE$^+$07), which we introduced in Chapter 5, a language based on equational and rewriting logic, together with a rewrite engine for executing it.

The tool offers two main features:

- the automatic generation of a counterpart model provided a description of a system, specified by the initial state of the system, that is an

algebra of a given signature, and by a description of the dynamics of the system, namely a set of SPO-like rewrite rules. We discuss this system specification language in Section 6.1;

- the evaluation of logical formulae of our second-order μ-calculus in the automatically generated counterpart model. As specified by the semantics of our logic (Definition 2.11), formulae are evaluated as sets of assignments for each state, associating sorted variables to elements with the same sort.

Actually, our tool implements two different semantics for our logic. In fact, by setting a flag to true or false, it considers either the original semantics given in (GLV10), or the one proposed in (GLV12a) and discussed in Chapter 2.

In (LV11) we presented an early version of our tool considering the semantics of (GLV10), while in (GLV12a) and in this chapter (and in general in the whole thesis) we focus on the newer semantics.

Informally, as sketched in Section 2.5, the main difference between the two semantics consists in the evaluation of formulae having $\Diamond$ as main operator. In fact, in order to be adherent to Lewis Counterpart Theory, in the original semantics we do not consider (i.e. we ignore) transitions that do not preserve all the elements relevant for the formula.

6.1 System specification

In order to specify a system we need to feed the tool with three informations:

1. the signature of the system;
2. the initial state of the system;
3. the dynamics of the system.

Signature specification. Our tool is parametric with respect to the signature Σ of the structures associated to the states of the systems. However,

Listing 6.1: The Maude code to define the graph signature

```
1  --- This module provides the sorts and the operation names of a signauture
2  mod GRAPH-SIGNATURE is
3
4    pr CT-MODEL-SORTS .
5
6    sorts Edge Node . --- The sorts
7    --- Element is a "supersort" used to obtain code independent from the
         actual signature
8    subsort Edge Node < Element .
9
10   op Edge : -> Cid [ ctor ] .
11   op Node : -> Cid [ ctor ] .
12
13   op e : Nat -> Edge [ ctor ] . --- e(0), e(1) are edges
14   op n : Nat -> Node [ ctor ] . --- n(0), n(1) are nodes
15
16   --- The operations s and t are hash tables from an "NAryArgument" to an "
         Element".
17   op s:_ : Map{NAryArgument,Element} -> Operation [ ctor ] .
18   op t:_ : Map{NAryArgument,Element} -> Operation [ ctor ] .
19
20   --- I conservatively apply the renaming induced by cr to source and
         target
21   var source target : Map{NAryArgument,Element} .
22   var cr : CounterpartRelation . var conf : Configuration .
23   eq conservativelyApplyToOperations(cr, s: source conf)
24    = conservativelyApplyToOperations(cr,           conf)
25      s: applyToOperationIfDefined(cr, source) .
26
27   eq conservativelyApplyToOperations(cr, t: target conf)
28    = conservativelyApplyToOperations(cr,           conf)
29      t: applyToOperationIfDefined(cr, target) .
30
31 endm
```

we designed the tool so that almost all of its code is independent from the choice of Σ. Intuitively, we only have to provide the tool with (a Maude module containing) informations about the sorts and the names of the operations of the signature.

Listing 6.1 provides the module necessary to specify the signature of unlabelled directed graphs, shown in Example 2.1. Line `6` provides the sorts of the signature (i.e. `Node` and `Edge`). In order to obtain most of the code of the tool independent from the particular used signature, as discussed in Chapter 5 we defined a kind of *supersort* `Element`, of which every element is subsort (line `8`).

For the same reason, as shown in lines 10-11, every sort should have associated a corresponding operation with the same name and with co-domain Cid, a predefined Maude sort standing for class identifiers. As it will be clear in the next sections, those are needed to be able to restrict some computation to a subset of the sorts.

Lines 13-14 provide the constructors to build elements of the defined sorts: e(0) and e(1) are examples of edges.

Finally, in lines 17-18 are specified the names of the operations of the signature, that is *source* s: and *target* t:, while in lines 23-29 it is specified how a counterpart relation is conservatively applied to the Σ-operations of a configuration. This is used during the generation of a new state, and is discussed in paragraph **Dynamics specification** of this section. Intuitively, we only have to specify the names of the operations.

Notice that Maude provides variables that can be used as placeholders for Σ-elements or Σ-operations in the definition of equations. For example, the variables source and target represent two hash tables from n-ary sequences of Σ-elements to a Σ-element, which we use to explicitly represent Σ-operations.

We are actually currently working on a GUI to automatize the generation of this module and of the others discussed in the following.

Initial state specification. Once the signature Σ of system states has been fixed, it is necessary to provide to the tool the initial state of the considered system, that is a Σ-algebra.

Listing 6.2 shows a possible definition of two Maude operations to obtain the initial state of the leader election system considered in Section 2.1, and shown in Figures 2.1, 2.3. More detailed informations are provided in Section 8.1.

Intuitively, initLE2 (lines 7-10) generates an initial state (that is an unlabelled directed graph) with two edges (e(3), e(4)) and two nodes (n(3), n(4)) (line 8). In lines 9-10 are specified the operations source and target. Namely, n(3) is the source of e(3) and the target of e(4), while n(4) is the source of e(4) and the target of e(3). We obtain hence a graph connected in a ring topology, and in particular $\mathbf{G}_1$ of Figure 2.2.

Listing 6.2: The Maude code to define the initial state of the leader election system

```
1  mod INSTANCES-LE is
2
3    pr GRAPH-SIGNATURE .
4
5    var m size : Nat .
6
7    op initLE2 : -> Configuration .
8    eq initLE2 = e(3) e(4) n(3) n(4)
9                 s: ({e(3)} |-> n(3), {e(4)} |-> n(4))
10                t: ({e(3)} |-> n(4), {e(4)} |-> n(3)) .
11
12   op initLE : NzNat -> Configuration .
13   eq initLE(size)
14    = initLE-Elements(size) initLE-Operations(size,size) .
15
16   op initLE-Elements : NzNat -> Configuration .
17   eq initLE-Elements(s(m)) = e(m) n(m) initLE-Elements(m) .
18   eq initLE-Elements(0) = none .
19
20   op initLE-Operations : NzNat NzNat -> Configuration .
21   eq initLE-Operations(size,size)
22    = s: initSource(size,size) t: initTarget(size,size) .
23
24   op initSource : NzNat NzNat -> Map{NAryArgument,Element} .
25   eq initSource(s(m), size)
26    = initSource(  m , size), {e(m)} |-> n(m) .
27   eq initSource(0, size) = empty .
28
29   op initTarget : NzNat NzNat -> Map{NAryArgument,Element} .
30   eq initTarget(s(m), size)
31    = initTarget(  m , size), {e(m)} |-> n(s(m) rem size) .
32   eq initTarget(0, size) = empty .
33
34 endm
```

Lines 12-32 define the operation `initLE` (and the necessary utility operations), which takes as parameter a non-zero natural number `n`, and generates an unlabelled directed graph connected in a ring topology with `n` nodes and edges. The graphs in Figure 2.2 are actually three possible states of the system.

Dynamics specification. The last information that we have to provide to the tool is the dynamics of the system.

Listing 6.3 shows the specification of the dynamics of the leader election system. Intuitively, at every step, an edge is non-deterministically

Listing 6.3: The Maude code to define the dynamics of the leader election system

```
1  mod BEHAVIOUR-LE is
2
3    pr GRAPH-SIGNATURE .
4
5    var e1 : Edge . vars ns nt : Node .
6    vars source target : Map{NAryArgument,Element} .
7    var cr : CounterpartRelation .
8
9    crl [executeStep] :
10     ns e1 nt
11     s: (source, {e1} |-> ns)
12     t: (target, {e1} |-> nt)
13   =>
14   {cr}(
15     ns
16     s: applyToOperationIfDefined(cr,source)
17     t: applyToOperationIfDefined(cr,target)
18     )
19   if  cr := (ns |~> ns, nt |~> ns) .
20
21 endm
```

chosen (represented at line `10` with the placeholder variable `e1`) and deallocated, and its source and target nodes (`ns` and `nt`) are collapsed to maintain the ring topology. Clearly operations have to be accordingly updated to account for the deallocation of `e1`, and for the merging of `ns` and `nt`. Operations not matched by the application of a rule will be automatically, silently, updated.

By referring to the example in Listing 6.3, we now discuss in detail how the dynamics of a system can be specified in our tool as sets of behavioural rules.

The dynamics of a system are specified by a set of term rewrite rules given in SPO-like formalism, which we implemented on top of Maude's term rewriting by enriching the rules with the counterpart relation (similar to the trace morphism).

An example of such rules is provided in Listing 6.3. The main idea is that each rule has a left-hand side pattern (LHS), shown in lines `10-12`, and a right-hand side pattern (RHS), shown in lines `14-18`. Both the LHS and the RHS of a rule are possibly composed by a set of elements (e.g.

nodes and edges) and by a set of operations (e.g. source and target) of Σ, hence they can be thought of as incomplete Σ-algebras.

Notice that a rule is defined with the keyword `crl` (line `9`), which may be possibly followed by the name of the rule surrounded by square brackets (e.g. `executeStep`), while the LHS and the RHS are divided by the keyword `=>` (line `13`).

Intuitively, the system performs a computation step by applying a rule to a state, hence each rule specifies a particular kind of evolution step which can be performed by the system. The RHS of a rule also contains a partial mapping from the elements in the LHS to the ones of the RHS (the `cr` of lines `14` and `19`). This mapping, which is actually a partial morphisms from the LHS to the RHS of the rule, explicitly specifies how the elements of a state evolve upon the application of the rule, and it is used to build the trace morphism, which in our setting amounts to the counterpart relations labelling the transitions generated by the rule.

In our tool we implement a two-level rule scheme: at the lowest level we have a set of system-specific *local rules* (e.g. the one in Listing 6.3), while at the top level we have a uniquely defined *global rule* that takes care of local rule application at the global level, in order to extend the counterpart relation (generated applying a local rule to only part of a state) with the (preserved) elements not considered by the local rule. Intuitively, this two-level infrastructure allows to specify the behaviour of a system by specifying the (local) behaviour of its components, or of sets of them.

A local rule r can be applied to a (part of a) state s of the system (i.e. a Σ-algebra) only if it is possible to find a total matching m for the items of the LHS of r with part of the ones in s. The matching m has to be injective (the items in the LHS have to be associated to distinct items of s) and has to respect sorts and operations of Σ, i.e. it has to preserve them. In other words, m is a total injective morphism from the LHS of r to s.

By applying the local rule r to s via m we obtain a new (incomplete) Σ-algebra defined applying m to the RHS of r, labelled by a counterpart relation obtained composing the mapping component of r (e.g. the `cr` of line `19` of Listing 6.3) with m.

The role of the global rule is that of lift to the global level, i.e. to the

whole state, the application of a local rule to part of a state. Hence, the global rule can be applied to a state s whenever a local rule can be applied to part of it.

In particular, the application of the global rule to a state s consists in first applying a local rule r to part of s via a matching m. Then, the state s' successor of s computed applying r to s via m is obtained by removing from s its items matched by m, and by adding to it the ones obtained applying m to the RHS of r.

As we will see, during the generation of the model, the so-obtained (this time total) Σ-algebra s' is used to label a new world of the model (e.g. w'). Moreover, a new transition will be added to the model from the world labelled by s to w'. Such transition will be labelled with the counterpart relation obtained extending the one generated by the application of the local rule r (obtained composing the mapping component of r with m), extended with the identity for all elements of s not matched by m.

We recall once more that cr correlates the items of s preserved by the transition from w to w', allowing for their renaming and merging. Moreover, cr is undefined for the items of s deallocated during the transition, while its image does not contain the newly created items of w'.

Looking back at Listing 6.3 it is now easy to understand the behaviour of the system. By applying the rule to a state of the system, we match an edge with `e1` and two nodes with `ns` and `nt` (line `10`). Moreover, from line `11` we have that `ns` is the source of `e1`, while from line `12` we have that `nt` is its target. Then, from the definition of `cr` (line `19`) we see that `e1` is deallocated, while `ns` and `nt` are merged in a node with the name of `ns`.

In lines `16-17` we update the two Σ-operations to account for the deallocation of `e1`. Namely, we remove the entries regarding source and target of `e1`. Moreover, in those two lines we also conservatively apply `cr` to the two Σ-operations, to account for possible renamings induced by `cr`. In this particular case it is necessary to change any reference to the node matched by `nt` in `ns`.

Thanks to the global rule, `cr` is extended with the identity for all the items not considered by `executeStep`, meaning that they are preserved.

Listing 6.4: The global rule which applies the system-specific local rules

```
1  --- This module defines how a system can evolve. It contains a rule
       regarding the whole state. Such rule utilizes the system-specific rules
        provided by the user
2  mod GLOBAL-RULE is
3
4    pr CT-MODEL .
5
6    var cr extendedCR : CounterpartRelation .
7    vars conf conf3 conf4 remainingConf : Configuration .
8
9    crl [global] : << conf >> => {extendedCR}<< conf3 conf4 >>
10    if conf => (remainingConf ({cr}conf3))
11    /\ conf4 := conservativelyApplyToOperations(cr, remainingConf)
12    /\ extendedCR := extend(cr,remainingConf) .
13
14 endm
```

Moreover, the global rule will also take care of applying `cr` to the other Σ-operations possibly not matched by the local rule.

We conclude this paragraph by discussing a simplified version of our above mentioned global rule, where we omit the automatic assignment of names to newly generated Σ-elements, and the application of state canonizers, which, following our `c`-reduction approach of Section 3.3, allows to obtain reduced state-spaces. The first aspect will be exemplified in Section 8.2, while the second aspect is treated in Chapter 7, and exemplified in Sections 8.1, 8.2.

The simplified global rule is provided in Listing 6.4. From line `9` we know that a state (i.e. a Σ-algebra) matched by `conf` evolves in the state `conf3 conf4`, with counterpart relation `extendedCR`.

In particular, `conf3` is obtained by the application of a local rule (e.g. `executeStep`) to part of the state matched by `conf`, with counterpart relation `cr`.

The variable `remainingConf` matches the part of `conf` not matched by the application of the local rule, hence the Σ-elements (and the Σ-operations) in `remainingConf` are not affected by the application of the local rule, and should then be preserved. For this reason we compute `extendedCR` by extending `cr` with the identities for the Σ-elements in `remainingConf`.

Finally, we obtain `conf4` by *conservatively* applying the counterpart relation `cr` to `remainingConf`. This allows to apply the renamings imposed by `cr` to the Σ-operations in `remainingConf`, if any. By saying *conservatively*, we mean that only references to Σ-elements for which `cr` is defined are updated, while the other are left unchanged. Intuitively, this corresponds to applying `extendedCR`, but it is more efficient.

6.2 Counterpart model generation

The tool does not perform on-the-fly model checking: it first generates the (finite) counterpart model for a given rule-based specification, and then evaluates formulae over such model.

The procedure of generation of a counterpart model is quite intuitive. It starts from a counterpart model composed by a world associated to an algebraic structure representing the initial state of the system, and by the empty accessibility relation. Then it keeps adding worlds and entries of the accessibility relation to the model, up to completion of the state-space. This is done by applying exhaustively the global rule to the algebraic structures labelling the generated worlds.

As previously discussed, the application of the global rule to a Σ-algebra s generates a new Σ-algebra s' and a counterpart relation cr. In particular, two cases can arise after the application of the rule, supposing that s labels the world w:

1. the model does not already contain a world labelled with s', in which case a new world w' labelled with s' it is added to the model. Moreover, also an accessibility relation entry from w to w' labelled with cr is added to the model;

2. the model already contains a world labelled with s', thus only the corresponding accessibility relation entry is added, if not already present.

It is worth to mention that, other than identifying syntactically identical algebras, we could exploit techniques to reduce the size of the state-space obtaining *equal* (i.e. bisimilar) or *approximated* (i.e. similar) models.

For example, when creating new elements we could reuse names of previously deallocated elements, allowing to obtain finite counterpart models for infinite-state systems with bounded resource allocation. Moreover, also more powerful strategies, based e.g. on identifying *symmetric* states could be applied.

In (LMV12) we studied how to apply some of those state-space reduction techniques to Kripke models, developing an approach named `c`-reductions, based on state canonizers. In Section 3.3 we lift these results to counterpart models, while in Chapter 7 we present some state canonizers currently implemented in our tool.

Moreover, by studying how the semantics of our logic is related to model approximations, in Chapter 4 we gave the formal relation between the evaluation of formulae in the original models and reduced ones.

6.3 Formulae evaluation

Given a counterpart model M and an assignment for fix-point variables (we usually initially fix the empty one), our tool evaluates a formula-in-context as the set of pairs (w, σ_w) satisfying it under the semantics defined in Section 2.4 [1]. With w being a world of M, and σ_w a variable assignment for w defined for a subset of the variables in the first-order context, and exactly for the variables in the second-order context of the formula.

The set of pairs satisfying the formula is computed by relying on the function `valid`, taking as arguments a formula-in-context, a pair, a fix-point variable assignment and a counterpart model. A call to function $\texttt{valid}(\psi, (w, \sigma_w), \rho, M)$ returns `true` if the pair (w, σ_w) validates the formula ψ in M under fix-point variable assignment ρ, and `false` otherwise. Finally, we evaluate the semantics of a formula-in-context with a function $[\![\_]\!]_{\_,\_}$ taking as arguments a formula-in-context $\psi[\Gamma; \Delta]$, an initial state w_0 of the system (from which the counterpart model will be built), and an assignment ρ for fix-point variables. The operation generates all the pairs

[1] Or the one proposed in (GLV10). However, as previously mentioned, we will focus on the semantics proposed in Section 2.4.

in the set $\Omega^{[\Gamma;\Delta]}$, and adds to the semantics of the formula only those pairs for which $\texttt{valid}(\psi, (w, \sigma_w), \rho, M)$ is true.

Note that the described procedure is a simple brute-force algorithm implementing the semantics of Section 2.4. However, as we previously explained, our aim was to develop a tool to test and debug our semantics, and to demonstrate the feasibility of our approach, rather than to provide an efficient verification engine.

So far we focused on developing techniques to generate and exploit smaller models, which is a very important aspect, since the complexity of any model checking procedure depends on the size of the considered state-spaces.

Indeed, in Chapter 7 we see how it is possible to extend the tool we presented to support `c`-reductions.

In (LV11) we have tested our tool over a system implementing the well-known stable marriage problem (GI89). Moreover, the properties discussed in Example 2.5 and in Example 2.6 have been checked automatically using our tool. In Chapter 8 we will consider other interesting cases.

Chapter 7

Tool support for c-reductions

In Section 3.3, we presented c-reductions, a state-space reduction technique for counterpart models. The approach is based on state *canonizers*, namely functions mapping the worlds of a model in a (non necessarily unique) canonical representative of their equivalence class, provided by an iso-bisimulation. The approach is general enough to allow for recasting in it other reduction techniques like name reusing and symmetry reduction.

In this chapter we discuss how our prototypal model checker presented in Chapter 6 has been extended to support c-reductions, and sketch some currently implemented canonizers.

Then, in Chapter 8, with the help of appropriate running examples, we will discuss in more detail the implemented canonizers, and we will exemplify how them, and their combinations, can be exploited to enhance system analysis capabilities.

We recall that canonizers are functions applied to the Σ-algebras labelling the worlds of our counterpart models in order to compute smaller but semantically equivalent (i.e. iso-bisimilar) models, provided an iso-bisimulation.

Intuitively, the c-reduction of a counterpart model, that is the reduced

model resulting from the application of a canonizer to its worlds, corresponds to the model obtained by (partly) collapsing the equivalence classes of its worlds.

We distinguish between strong and weak canonizers. Strong canonizers provide unique representatives for each equivalence class of worlds (i.e. all equivalence classes are collapsed in single worlds) and, hence, the maximal reduction which can be obtained for a given equivalence. In the case of non-strong (weak) canonizers, we might have different representatives for equivalent worlds (i.e. equivalence classes are possibly not collapsed in single worlds). Weak canonizers can hence be thought of as heuristic canonizers providing weaker state-space reductions.

However, weak canonizers may enjoy advantages over strong ones, like easiness of definition, and less expensive computations. Meaning that in some cases they are easier to be defined, and that their exploitation can be more computationally efficient in terms of runtime cost.

Strong canonizers provide smaller models, while weak canonizers require less computations. Depending on the different computation requirements and reduction ratios, we may have that the generation of a counterpart model reduced with a weak canonizer may require either more or less time than with a strong canonizer.

We now move our attention on how we can exploit `c`-reductions in the tool presented in Chapter 6. Practically, we obtain an on-the-fly reduction technique (i.e. computed during the generation of a counterpart model) by simply applying a canonizer to every newly generated world (and to the counterpart relation generated with it), before storing it. Notice that we directly build the reduced model, meaning that we can generate a reduced model if it is finite, independently on the finiteness of the original model.

Listing 7.1 shows how we easily extend the global rule depicted in Listing 6.4 in order to support `c`-reductions.

Notice how the rule of Listing 6.4 has been extended to obtain the one of Listing 7.1. In Listing 6.4 we generate the Σ-algebra `conf3 conf4` with counterpart relation `extendedCR`. In Listing 7.1 we instead further apply the function `reducer` to the result of the former, obtaining the

Listing 7.1: An enriched global rule to support c-reductions

```
1  mod GLOBAL-RULE is
2
3    pr CT-MODEL .
4
5    var cr extendedCR crRed : CounterpartRelation .
6    vars conf conf3 conf4 remainingConf confRed : Configuration .
7
8    crl [global] : << conf >> => {crRed}confRed
9     if conf => (remainingConf ({cr}conf3))
10    /\ conf4 := conservativelyApplyToOperations(cr, remainingConf)
11    /\ extendedCR := extend(cr,remainingConf)
12    /\ {crRed}confRed := reducer({extendedCR}<< (conf3 conf4) >>).
13
14 endm
```

Σ-algebra `confRed` and the counterpart relation `crRed` (line `12`).

Where `reducer` can be thought of as a *generic* canonizer (having as domain and co-domain Σ-algebras labelled with a counterpart relation), defined as

```
op reducer : LabelledState -> LabelledState .
```

By resorting to the extended rule of Listing 7.1, we can easily exploit any provided *concrete* canonizer, just by adding a simple equation connecting the generic and the concrete canonizers. For example, suppose to have defined a concrete canonizer `compactNames`. Then, by providing the following equation, we obtain counterpart models reduced with `compactNames`:

```
var lState : LabelledState .
eq reducer(lState) = compactNames(lState) .
```

In our tool we currently implemented three state canonizers:

1. `compcatNames`,
2. `freeTranps` and
3. `rotation`

For easiness of presentation, we now only informally sketch the three canonizers, as we will discuss them in more detail in Chapter 8, with the help of the system specifications there provided.

Canonizer `compactNames`. The canonizer `compactNames` is defined such that, for each sort provided by Σ, it simply renames the elements of a Σ-algebra by *compacting* their names towards the bottom of a provided ordering for the identifiers of elements of that sort. This canonizer may be useful to *compact* names after elements deallocation, merging and renaming, implementing a sort of garbage collection of identifiers, and facilitating the reuse of currently unused ones.

Intuitively, the canonizer is based on total orderings for elements names (one for each sort), and simply pushes down elements names towards the bottom of the orderings, starting from the minimal one.

In Section 6.1 we have seen how it is possible to specify the leader election system of Section 2.1. Then, suppose to have an ordering for `Edge` identifiers such that `e(i)<e(j)` if `i<j`, having hence `e(0)` as bottom. Suppose moreover to have a state of the system composed by the three edges {`e(1)`,`e(3)`,`e(4)`}. Applying the canonizer to this state we obtain that `e(1)` is renamed in `e(0)`, `e(3)` is renamed in `e(1)`, and `e(4)` is renamed in `e(2)`. The same procedure is applied to the elements of every sort.

As discussed in Section 8.1, this generates a huge reduction of the size of the state-space of the leader election system. In particular, every state with three edges will contain the edges {`e(0)`,`e(1)`,`e(2)`}. However, notice that this does not imply that all the states with the same number of elements are identified, because states are Σ-algebras, and hence we also have operations among elements. Consider the simple example of the two graph algebras:

```
e(1) n(1) n(3) s: ({e(1)} |-> n(1)) t: ({e(1)} |-> n(3))
e(1) n(1) n(3) s: ({e(1)} |-> n(3)) t: ({e(1)} |-> n(1))
```

Then, if we apply `compactNames` to them we obtain the two distinct Σ-algebras:

```
e(0) n(0) n(1) s: ({e(0)} |-> n(0)) t: ({e(0)} |-> n(1))
e(0) n(0) n(1) s: ({e(0)} |-> n(1)) t: ({e(0)} |-> n(0))
```

We do not provide the code implementing such canonizer. Instead, it may be interesting to show the code that the user has to provide in order

Listing 7.2: The Maude code to exploit `compactNames` (1)

```
1  mod CODEFOR-NAME-COMPACT-LE is
2
3    pr COMPACT-NAMES .
4    pr ABSTRACTIONS .
5    pr GRAPH-SIGNATURE .
6
7    vars el1 el2 : Element . var i j : Nat .
8    var source target : Map{NAryArgument,Element} .
9
10   --- Sorts to be compacted
11   eq CLASSES-2BECOMPACTED = (Edge,Node) .
12   eq getCid(e1:Edge) = Edge .
13   eq getCid(n1:Node) = Node .
14
15   --- minimal name
16   eq minId-2CompactAndAllocate(Edge) = e(0) .
17   eq minId-2CompactAndAllocate(Node) = n(0) .
18
19   --- next name
20   eq successor-2CompactAndAllocate(e(i)) = e(i + 1) .
21   eq successor-2CompactAndAllocate(n(i)) = n(i + 1) .
22
23   --- Name comparison
24   eq e(i) <#El e(j) = i < j .
25   eq n(i) <#El n(j) = i < j .
26
27   --- propagation of compacting to operations
28   eq (s: source)[ el1 -> el2 ] =  s: compactOp(source,el1,el2) .
29   eq (t: target)[ el1 -> el2 ] =  t: compactOp(target,el1,el2) .
30
31 endm
```

to use this canonizer. As done in Section 6.1, we will again consider the case of the leader election system.

Listings 7.2 and 7.3 provide the Maude code necessary to exploit the canonizer `compactNames`. As result, the tool will generate models reduced with the canonizer `compactNames`.

The listings provide two modules: `CODEFOR-NAME-COMPACT-LE` and `TEST-COUNTERPARTMODELGENERATION-LE`. In the first one we provide the informations necessary to the canonizer, while the second one imports all the modules necessary to the generation of a counterpart model, and specifies the possibly used canonizer.

We first consider Listing 7.2. In lines `3-4` we import the modules defining the canonizer, while in line `5` we import the signature of graphs.

Listing 7.3: The Maude code to exploit `compactNames` (2)

```
1  mod TEST-COUNTERPARTMODELGENERATION-LE is
2
3    pr BEHAVIOUR-LE .
4    pr INSTANCES-LE .
5    pr CT-MODEL-BUILDER .
6    pr META-CONNECTOR .
7    pr CODEFOR-NAME-COMPACT-LE .
8
9    var lState : LabelledState .
10
11   --- which reduction you want to use
12   --- eq reducer(lState) = lState .  --- use this for none
13   eq reducer(lState) = compactNames(lState) .
14
15 endm
```

In line `11` we specify the sorts to which we want to apply the canonizer. For efficiency reasons, here should be listed only the sorts of elements which are deallocated (and relloacted), merged and renamed, because those are the only cases in which it makes sense to compact their names. In this case both `Edge` and `Node`, because the first ones are deallocated, while second ones are merged. For the same reasons, lines `12-13` provide the equations to obtain the class corresponding to the sorts that we want to compact. As previously mentioned, this is a redundant information provided to make it easier to write code independent from the considered Σ.

Lines `16-25` provide informations about the orderings of the names of edges and nodes. In particular, in lines `16-17` we provide the minimal names, in lines `20-21` we state how it is possible to generate the *next* name in the ordering, while in lines `24-25` we specify how to compare two names.

Finally, in lines `28-29` we specify how the compacting is propagated to the elements referenced by Σ-operations. Intuitively, we need an equation for every operation.

The only interesting line of the module Listing 7.3 is `13`, where we specify the canonizer to be used. Interestingly, the unreduced state-space is obtained decommenting line `12`, and commenting `13`.

Canonizers `freeTransps` and `rotations`. The other two canonizers `freeTransps` and `rotations` implement a state-space reduction technique well known in the field of model checking, namely symmetry reduction (WD10). In particular, `freeTranps` allows to reduce systems characterized by *full symmetry*, while `rotations` allows to reduce systems presenting *rotational symmetry*.

This deserves some comments. In the context of symmetry reduction (WD10) there exist several possible *kind* of symmetries. The most general one is the so called *full symmetry*, where we can freely transpose (i.e. pairwise swap) the names of symmetric entities, obtaining an equivalent (i.e. an iso-bisimilar) state. Intuitively, this is the case of systems composed by several replications (copies) of components independently performing the same computations, as for example the processes (i.e the edges) of our leader election system.

Clearly, in a state with n of such components, n distinct steps can be performed towards distinct but symmetric states, depending on which component executes.

Then it is possible to consider only one state for every equivalence class of symmetric states, e.g. a *minimal* one for a given total ordering, which can be obtained from every state in the equivalence class by applying a sequence of transpositions.

In the other cases of symmetries, constraints are imposed on the kind of manipulation that can be done to the names of the elements. The well known dining philosophers problem that we treat in Section 8.2 is an emblematic example of *rotational symmetry*, where the transposition of names could possibly lead to not bisimilar states. Instead, by rotating them all at once, we obtain bisimilar states. The intuition is that it is not important the *absolute position* of a philosopher in the table, what matters is instead its relative position with respect to its neighbours (the two philosophers with which it shares its two forks), in the sense that they have to be preserved, so that it will not happen that two eating philosophers become neighbours.

The canonizer `freeTransps` transposes (i.e. swaps pairwise) the names of the elements (preserving sorts), in order to obtain a minimal state.

The canonizer `rotations` instead computes all the possible rotations of names (which are actually n for systems with n rotationally symmetric elements), and computes the minimal state.

Hence, for those two canonizers we do not only require an ordering among elements names, but also among system states. A very intuitive one may be the one in which we order states lexicographically, in the sense that we first compare a Σ-operation of the two states (e.g. source), and if they are equal we compare another Σ-operation (e.g. target), and so on. In order to compare two Σ-operations, we again follow a lexicographical approach, i.e. we compare pairs of entries starting from the minimal ones (e.g. the ones regarding the minimal elements).

Notice that it is not important to compare elements, since all the states resulting from a transposition or from a rotation have the same elements.

For example, the following equation (where `<#Conf` is the ordering for Σ-algebras, while `<#Op` is the ordering for Σ-operations) sketches an ordering for Σ-algebras:

```
eq (s: s1 t: t1 conf1) <#Conf (s: s2 t: t2 conf2)
 = if s1 <#Op s2
    then true
    else if s1 == s2
          then t1 <#Op t2
          else false
         fi
   fi .
```

We recall that `compactNames` does not collapse the two following Σ-algebras

```
e(1) n(1) n(3) s: ({e(1)} |-> n(1)) t: ({e(1)} |-> n(3))
e(1) n(1) n(3) s: ({e(1)} |-> n(3)) t: ({e(1)} |-> n(1))
```

In fact, they have the two distinct minimal representatives

```
e(0) n(0) n(1) s: ({e(0)} |-> n(0)) t: ({e(0)} |-> n(1))
e(0) n(0) n(1) s: ({e(0)} |-> n(1)) t: ({e(0)} |-> n(0))
```

If we instead apply the `freeTranps` canonizer to the resulting Σ-algebras, then they are collapsed in the following minimal representative

```
e(0) n(0) n(1) s: ({e(0)} |-> n(0)) t: ({e(0)} |-> n(1))
```

Listing 7.4: The Maude code to exploit `freeTransps` (1)

```
mod CODEFOR-STATE-ORDERING-LE is

  pr STATES-ORDERING . pr ABSTRACTIONS . pr GRAPH-SIGNATURE .

  vars conf conf2 : Configuration . var i j size : Nat .
  var source target source2 target2 : Map{NAryArgument,Element} .

  --- State comparison
  eq (s: source t: target conf) <#Conf (s: source2 t: target2 conf2) =
   if source <#Op source2
    then true
    else if source == source2 then target <#Op target2 else false fi
   fi .

  --- Compare two elements
  eq e(i) <#El e(j) = i < j .
  eq n(i) <#El n(j) = i < j .

endm

mod CODEFOR-TRANSPOSTIONS-LE is

  pr TRANSPOSITIONS . pr ABSTRACTIONS . pr GRAPH-SIGNATURE .

  var conf : Configuration . vars el1 el2 : Element .
  var source target : Map{NAryArgument,Element} .

  --- Sorts to be transposed
  eq CLASSES-OF-SORTS-TO-BE-TRANSPOSED = (Edge,Node) .
  eq getCid(e1:Edge) = Edge . eq getCid(n1:Node) = Node .

  --- Application of transposition to the state
  eq (s: source t: target conf)[ el1 <-> el2 ]
   =  s: applyTranspToOperation(source,el1,el2)
      t: applyTranspToOperation(target,el1,el2) conf .

endm
```

This leads to the idea of combining canonizers. For example we just exemplified the combined canonizer obtained combining `freeTranps` with `compactNames`.

We obtain the canonizer `freeTranpsocompactNames` which first compacts the names of a Σ-algebra, and then transposes them. Interestingly, for the case of graphs (i.e. when we fix as Σ the signature of graphs presented in Example 2.1), with this composed canonizer we consider the equivalence class of isomorphic states, and map each graph in a minimal isomorphic one, obtaining thus isomorphism reduction.

Listing 7.5: The Maude code to exploit `freeTransps` (2)

```
1  mod TEST-COUNTERPARTMODELGENERATION-LE is
2
3    pr BEHAVIOUR-LE .
4    pr INSTANCES-LE .
5    pr CT-MODEL-BUILDER .
6    pr META-CONNECTOR .
7
8    pr CODEFOR-STATE-ORDERING-LE .
9    pr CODEFOR-TRANSPOSTIONS-LE .
10   pr STATE-ORDERING-TRANSPOSESOMESORTS .
11
12   var lState : LabelledState .
13
14   eq reducer(lState) = freeTransps(lState) . --- the canonizer to be used
15
16 endm
```

Other combinations of canonizers are possible. In Chapter 8, and in particular in Section 8.2, we show some interesting examples.

We now exemplify the code necessary to exploit `freeTranps`, while we will exemplify the one for `rotations` in Section 8.2.

Listings 7.4 and 7.5 provide the Maude code necessary to exploit the canonizer `freeTransps` for the case of the leader election system. As result, the tool will generate models reduced with this canonizer.

The listings regard three modules. `CODEFOR-STATE-ORDERING-LE` is the first one, which provides informations necessary to compare Σ-algebras. The second one is `CODEFOR-TRANSPOSTIONS-LE`, providing informations about transpositions. Finally, the third one imports all the modules necessary to the generation of a counterpart model, and specifies the used canonizer (`TEST-COUNTERPARTMODELGENERATION-LE`).

We first focus on Listing 7.4. Lines `3-4` import the modules defining the canonizer, while line `5` imports the signature of graphs. In lines `9-13` we provide an equation to order Σ-algebras, while in lines `16-17` we order elements names.

In module `CODEFOR-TRANSPOSTIONS-LE` we first indicate the sorts to which we want to apply the canonizer (lines `29-30`), and then we specify how transpositions are propagated to the operations of a Σ-algebra (lines `33-35`).

Listing 7.6: The implementation of the canonizer `freeTransps`

```
1  mod STATE-ORDERING-TRANSPOSESOMESORTS is
2
3    pr STATES-ORDERING .
4    pr TRANSPOSITIONS .
5
6    vars conf : Configuration . var cr : CounterpartRelation .
7    var cid : Cid . var otherCid : Set{Cid} . vars el1 el2 : Element .
8
9    --- Minimize a state transposing names
10   op freeTransps : LabelledState -> [LabelledState] .
11
12   eq freeTransps({cr}<< conf >>)
13    = $freeTransps({cr}<< conf >>, CLASSES-OF-SORTS-TO-BE-TRANSPOSED) .
14
15   op $freeTransps : LabelledState Set{Cid} -> [LabelledState] .
16
17   ceq $freeTransps({cr}<< el1 el2  conf >>,(cid,otherCid))
18     = $freeTransps({cr[el1 <-> el2]}<< el1 el2 (conf[el1 <-> el2]) >>,(cid,
         otherCid))
19    if getCid(el1) == cid /\ getCid(el2) == cid
20    /\ el1 <#El el2
21    /\ ((conf[el1 <-> el2]) <#Conf (conf)) .
22
23   eq $freeTransps({cr}<< conf >>,otherCid) = {cr}<< conf >> [ owise ] .
24
25 endm
```

The only interesting line of Listing 7.5 is `14`, where we specify the canonizer to be used.

We conclude the chapter by sketching the implementation of the canonizer `freeTransps`. Listing 7.6 sketches the implementation of the above considered canonizer `freeTransps`. In line `3` we import a module (partially) defining the operations to compare Σ-algebras, while in line `4` we import the module (partially) defining the operations to apply transpositions to configurations, Σ-operations, Σ-elements and counterpart relations.

Line `10` defines the canonizer, which is then implemented in lines `12-13`. In particular, the canonizer is reduced in `$freeTransps`, to which it is provided the set of sorts to which the canonizer should be applied (`CLASSES-OF-SORTS-TO-BE-TRANSPOSED`).

Line `15` defines `$freeTransps`, which is implemented in lines `17-23`. Intuitively, we see that the canonizer keeps on pairwise swapping

elements of a state, as long as it is possible to obtain a smaller state.

In particular, at each iteration a Σ-algebra is matched by the variables `el1 el2 conf`, where `el1` and `el2` are two Σ-elements, while the counterpart relation generated with the state is matched by `cr`.

From line `19` we know that the sorts of the two matched elements are equal and belong to the set of sorts considered by the canonizer. From line `20` we know that the element matched by `el1` is minor than the one matched by `el2`. This is necessary in order to avoid infinite cyclic repetitions of applications of opposite transpositions.

Finally, in line `21` we actually check if the state obtained after the transposition is minor than the matched one.

If all of those conditions hold, then the transposition is applied to the matched labelled state, which is used as argument of the next iteration of the canonizer.

Line `23` provides the base case of the canonizer. Namely, the flag `owise`, standing for *otherwise*, tells Maude to apply this equation if it is not possible to apply any other equation to reduce `$freeTransps`, i.e. if it is not possible to obtain a state minor than the current one by transposing its names. The resulting labelled state {`cr`}`<< conf >>` is the reduction of the original state, and it is actually the one considered during the generation of the counterpart model.

Notice that with this canonizer we reduce a state in the minimum of its equivalence class, where the considered equivalence class is given by all the states which can be obtained by transposing elements names.

In particular, we transpose only parts of the elements (the ones whose sorts are specified by `CLASSES-OF-SORTS-TO-BE-TRANSPOSED`). Instead, depending on the user-provided implementation of the state comparison operator, either the whole state, or part of it is considered to compare a state with the transposed one.

Then, depending on the ordering provided by the user, this canonizer can be either strong if the provided ordering is total (i.e. if the the whole state is considered in the ordering), or weak if the ordering is partial (i.e. if only part of the state is considered in the ordering).

Chapter 8

Model checker at work

In this chapter we validate our tool support over some well known systems, or particular versions of them. We consider the leader election system presented in Section 2.1, and an infinite-state implementation of the well known *dining philosophers problem* (Dij71) where, as proposed in (DRK02), forks are consumed and recreated upon usage. In this last case we only focus on the performances in the generation of the counterpart models.

Once more, the aim of this chapter is not to demonstrate the efficiency and scalability of our prototypal tool, but rather that of showing the wide applicability of our framework, the expressivity of our logic, and the performance gains brought by our efforts in defining techniques to reduce the sizes of the considered state-spaces and in dealing with reduced systems.

8.1 Leader election system

System description. In this section we discuss the running example of Chapter 2, that is a simple system modelling a leader election algorithm. The system has been presented in Section 2.1, and further discussed in Chapters 2 and 6. We recall that it consists in a sort of leader survivor game, in which there are a set of processes connected in a ring topology

via input and output ports. The game evolves via elimination rounds, in which a process is deallocated, and its ports are merged to maintain the ring topology. The game ends with the election of a leader, that is the only remaining process, which hence has same input and output ports. Figure 2.3 shows two possible executions of a leader election system with three processes (`e0`, `e1` and `e2`) connected via the three communication ports `n0`, `n1` and `n2`. We actually abstract from the particular algorithm used at each round to choose the process to be deallocated, and we focus instead on the evolution of the topology.

System specification. We already exemplified the specification of the system in our tool in Chapter 6. In particular, as shown in Listing 6.1, we represent states of the system as unlabelled directed graphs. Processes are modelled as edges (e.g. `e(0)`, `e(1)`), while communication ports are modelled as nodes (e.g. `n(0)`, `n(1)`). Intuitively, the source and target nodes of an edge represent, respectively, its input and output ports.

In Listing 6.2 it is instead shown the Maude operation `initLE` which allows us to generate initial states of the system. Given a natural number `n`, the operation generates a graph with `n` edges and `n` nodes connected in ring topology. As shown in lines `13-14` of Listing 6.2, a call to `initLE(size)`, where `size` is the natural number specifying the initial number of edges and nodes, reduces to `initLE-Elements(size) initLE-Operations(size)`. As shown in lines `16-18`, the first one generates `size` edges and nodes, namely `e(0) n(0) ...e(size-1) n(size-1)`.

As shown in lines `20-32`, `initLE-Operations(size)` generates the operations source and target of the `size` created edges, assigning, respectively, nodes `n(i)` and `n(i+1)` as source and target of edge `e(i)`. We actually defined `initSource` (lines `24-27`) and `initTarget` (lines `29-32`) to generate the two operations.

Finally, in Listing 6.3 we find the definition of the dynamics of the system. In lines `9-19` we provide the simple rule governing the behaviour of the system. Namely, at each step, an edge and its source and target nodes are selected, i.e. they are matched respectively with `e1`, `ns` and `nt`.

Then (the edge matched with) `e1` is deallocated, i.e. it does not appear neither in the right-hand side of the rule (the part of the rule following `=>`) nor in the counterpart relation `cr` labelling the rule.

Different is the case of `nt`. Even if it does not appear in the right part of the rule, it is not deallocated. In fact, as indicated in `cr` (line `19`), it is merged with `ns`. Hence, in the target states obtained applying this rule, will exist a node which represents both the one matched with `nt` and the one matched with `ns`.

As a last note, line `11` tells us that the node matched with `ns` is the source of the edge matched with `e1`, while line `12` tells us that the node matched with `nt` is its target. Then, in lines `16-17`, we *update* the operations source and target to reflect the deallocation of `e1`, and the fusion of `nt` and `ns`. In particular, the entries related to `e1` are removed, and `cr` is conservatively applied to the remaining parts of the operations, so to rename any reference to the node matched by `nt` to the one matched by `ns`.

Interesting properties. In Section 2.1 we enumerated several interesting properties of the leader election system which, as shown in Section 2.5, are expressible in our logic. We recall that we considered seven properties:

p1: *Will a leader be elected in all possible executions?*;

p2: *Can two distinct leaders be elected in the same execution?*;

p3: *Is there a process that necessarily becomes the leader?*;

p4: *In which states do we have a leader?*;

p5: *For which processes does an execution leading to its election exist?*;

p6: *Which communication ports will eventually merge?*;

p7: *Are processes connections correctly updated after each round?*.

We already provided the formulae expressing those properties in Examples 2.3 and 2.6. In particular, we first defined the derived predicates $\mathbf{present}_{\tau}(x)$, regarding the presence of an entity with sort τ in a world,

and **leader**(x) characterizing edges with same source and target node, i.e. leader processes. Then, by resorting to the temporal operators derived in Section 2.3, we expressed the listed properties as

p1 $\equiv AF[\exists x.(\textbf{leader}(x))]$,

p2 $\equiv \exists x.\exists y.[\neg \textbf{atMostOneLeader}(x,y)]$,

p3 $\equiv \exists x.[AF(\textbf{leader}(x))]$,

p4 $\equiv \exists x.(\textbf{leader}(x))$,

p5 $\equiv EF(\textbf{leader}(x))$,

p6 $\equiv (x \neq y) \ \wedge \ AF[\textbf{present}(x) \wedge \textbf{present}(y) \wedge (x = y)]$,

p7 $\equiv \textbf{present}(x_E) \wedge (x_N = s(x_E)) \wedge (y_N = t(x_E)) \wedge \Diamond[(\neg \textbf{present}(x_E)) \wedge (x_N \neq y_N)]$.

In Example 2.6 we discussed in detail the actual meaning of these formulae, and the used derived operators.

In the next paragraphs we will use our model checker to evaluate those formulae against an instance of an automatically generated counterpart model. As we will see, due to the size of the state-space, we are not able to build models with size greater than 15. Fortunately, by resorting to our c-reductions we are able to handle systems up-to size equal to 100. However, for easiness of presentation we will evaluate these formulae against a model with size fixed to 4 initial edges. In fact, fixing this size we obtain a model small enough to allow to depict it and its pairs.

Counterpart model generation. As discussed in Sections 6.2 and 6.3, our prototypal model checker does not perform on-the-fly model checking. It first generates the (finite) counterpart model of a given system specification, and then evaluates formulae over such a model. Thanks to the Maude attribute `memo`, which let store the result of the reduction of a function for a given set of parameters, the counterpart model is generated only once, and then it is stored and reused for every formula evaluated against it.

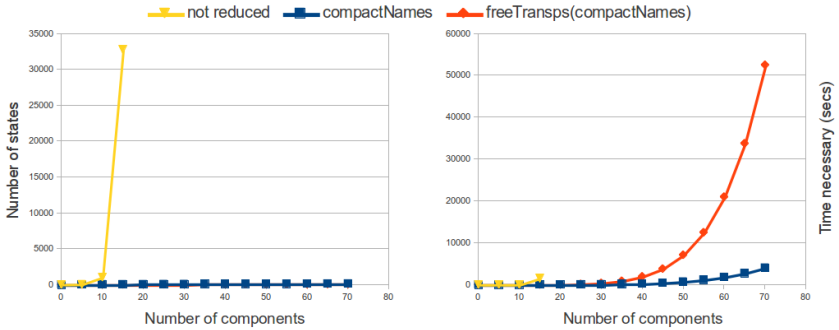

Figure 8.1: State-space sizes (left) and time necessary for their generation (right) at the varying of system size

In this paragraph we hence first discuss the performances of our tool in generating the counterpart models coming from system specifications with a varying initial number of edges (i.e. size). In particular, due to the size of the state-space, we are not able to build models with size greater than 15. However, by resorting to our c-reductions, we are able to generate the counterpart model for much greater size. In particular, we exploit the canonizer `compactNames`, and `freeTranspsocompactNames` (i.e. the composition of `compcatNames` with `freeTransps`) of Chapter 7.

We considered the size of the system varying from 5 to 100, however we report data from 5 to 70 (actually 100 for the `compactNames` case).

Figure 8.1 (left) reports informations about the number of worlds of the counterpart models generated at the varying of the size of the system (i.e. the initial number of edges), ranging from 5 to 70, while Figure 8.1 (right) reports informations about the time necessary to generate such models. The graphics regard three cases: the original (unreduced) counterpart models (*not reduced*), the ones obtained resorting to the canonizer `compactNames`, and the ones obtained by exploiting the canonizer `freeTranspsocompactNames` (depicted as `freeTransps(compactNames)`).

Before analyzing the performances obtained in generating the counterpart models, it is interesting to discuss about the size of the state-space of

this system. The dynamics of the leader election system are quite simple. From every state, an edge is non-deterministically selected and deallocated, while its source and target nodes are merged (having assigned the name of the source node).

Then, a state with n edges can perform n different one-step evolutions in n distinct states containing $n-1$ edges, that is all the edges except the one selected for deallocation. Each of those states with $n-1$ edges will generate $n-2$ states, and so on (even if with there will be repeated states). Then it is easy to understand that the size of the state-space of a system with $size$ initial edges is $2^{size}-1$.

The previous analysis is confirmed by the curve `not reduced` in the graphic on the left of Figure 8.1. Where, for $size = 5$ we have a counterpart model with 31 worlds, for $size = 10$ we have 1023 worlds, while for $size = 15$ we obtain 32767 worlds. Then it is not surprising that we are not able to generate (in a reasonable amount of time on a standard laptop) the counterpart model for $size = 20$, consisting of more than a million of worlds.

The situation changes dramatically by resorting to the two mentioned canonizers. We recall from Chapter 7 that the canonizer `compactNames` simply compacts the names of the elements in a state. Intuitively, the canonizer is based on total orderings for elements names (one for each sort), and simply pushes down elements names towards the bottom of the orderings, starting from the minimal one.

As we will discuss, the application of the canonizer `compactNames` generates a huge reduction of the state-space of this system specification, because all the $n-1$ states reachable from a state with n edges will have the same sets of elements. However, as discussed in Chapter 7, the application of the canonizer to the states having $n-1$ edges could still results in several distinct states, due to the fact that states are algebras, and hence we also have operations among elements. Consider the simple example of the two graph Σ-algebras proposed in Chapter 7:

```
e(1) n(1) n(3) s: ({e(1)} |-> n(1)) t: ({e(1)} |-> n(3))
e(1) n(1) n(3) s: ({e(1)} |-> n(3)) t: ({e(1)} |-> n(1))
```

Then, if we apply `compactNames` to them we obtain the two distinct

Σ-algebras:

```
e(0) n(0) n(1) s: ({e(0)} |-> n(0)) t: ({e(0)} |-> n(1))
e(0) n(0) n(1) s: ({e(0)} |-> n(1)) t: ({e(0)} |-> n(0))
```

The two Σ-algebras have same elements, namely `e(0)`, `e(1)`, and `n(1)`, but distinct operations. In fact, in the first algebra the source and target of `e(0)` are `n(0)` and `n(1)`, while in the second one are inverted, i.e. `n(1)` and `n(0)`.

Clearly, by swapping the names of the two nodes in the second Σ-algebra, we would obtain the first one. As discussed in Chapter 7, this is actually what the canonizer `freeTransps` does, searching for the minimal state which can be obtained by applying sequences of transposition, given an ordering for Σ-algebras.

In fact, by applying `freeTransps` to the two Σ-algebras reduced by `compactNames` (considering the ordering provided in Listing 7.4), we obtain the unique minimal representative

```
e(0) n(0) n(1) s: ({e(0)} |-> n(0)) t: ({e(0)} |-> n(1))
```

In particular, it should be easy to understand that the composition of the two canonizers allows to collapse all the states of the considered system having the same number of edges (and nodes). Moreover, since in every evolution step of the system we deallocate an edge, and the game ends when there remains only one edge, then the `c`-reduced counterpart model of this system under `freeTranspsocompactNames` has $size$ worlds, where $size$ is the initial number of edges.

Interestingly, for the case in which Σ is the signature of graphs, we have that the reduction obtained with `freeTranspsocompactNames` corresponds to reducing the state-space up-to isomorphic graphs.

In Chapter 7 we exemplified the system-specific code that is necessary to exploit both `compactNames` (Listings 7.2 and 7.3) and `freeTransps` (Listings 7.4 and 7.5). The code necessary to exploit the composite canonizer consists in the modules defined in the four listings, except for the module `TEST-COUNTERPARTMODELGENERATION-LE`, whose new version is in Listing 8.1. The only noteworthy line is `21`, where we specify the canonizer to be used. In the case depicted in the listing we adopt

Listing 8.1: The code to exploit the composition of `freeTransps` with `compactNames`

```
1  mod TEST-COUNTERPARTMODELGENERATION-LE is
2
3    pr BEHAVIOUR-LE .
4    pr INSTANCES-LE .
5    pr CT-MODEL-BUILDER .
6    pr META-CONNECTOR .
7
8    pr CODEFOR-NAME-COMPACT-LE .
9
10   pr CODEFOR-STATE-ORDERING-LE .
11
12   pr CODEFOR-TRANSPOSTIONS-LE .
13   pr STATE-ORDERING-TRANSPOSESOMESORTS .
14
15   var lState : LabelledState .
16
17   --- which reduction you want to use
18   --- eq reducer(lState) = lState .  --- use this for none
19   --- eq reducer(lState) = freeTransps(lState) .
20   --- eq reducer(lState) = compactNames(lState) .
21   eq reducer(lState) = freeTransps(compactNames(lState)) .
22
23 endm
```

the composed canonizer `freeTranspsocompactNames`. Instead, by commenting line `21` and uncommenting, respectively, lines `18`, `19`, or `20`, we would obtain the unreduced model, the model reduced with `freeTransps` or the model reduced with `compactNames`.

We can now discuss the graphics of Figure 8.1, where we notice that we are able to generate the unreduced counterpart model up-to systems with initially 15 edges (and nodes), obtaining 32767 states in 1500 seconds.

For what regards the models obtained exploiting the two canonizers, the graphics show instead informations up-to size 70, for which we obtain, respectively, 70 worlds in 52500 seconds and 138 worlds in 4021 seconds for the `freeTranspsocompactNames`, and `compactNames` canonizers. In particular we notice a linear growth of the sizes of the counterpart models, and much better time performances.

In order to better understand the different performances obtained with the two canonizers, in Figure 8.2 we propose the same graphics omitting the informations regarding the unreduced counterpart models.

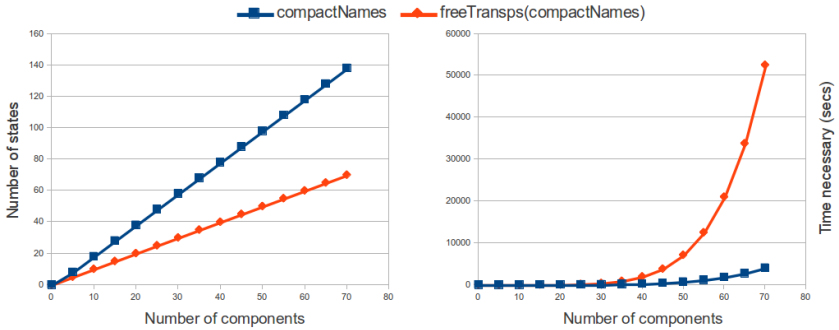

Figure 8.2: Reduced state-space sizes (left) and time necessary (right) at the varying of system size

Here we notice that both canonizers provide a linear growth of the sizes of the models at growing of the size of the system. More precisely, as expected (since we first apply `compactNames`, and then we further reduce with `freeTransps`), the composed canonizer offers better reductions, even if this gain is not enough to compensate for the extra computation introduced by `freeTransps`. As result, the canonizer `compactNames` offers slightly worse reductions but with much better performances.

In both the cases, the amount of time necessary to build counterpart models appears to grow exponentially, even if it grows much slower in the `compactNames` case.

In Figure 8.3 we focus on the canonizer `compactNames`, and we show informations about the generation of counterpart models with size varying from 5 to 100.

System analysis. We conclude this section by checking the considered properties against an instance of the leader election system. For easiness of presentation, we consider the model obtained fixing the initial number of edges to 4, reduced with canonizer `freeTranspsocompactNames`.

The Maude term representing such model is provided in Listing 8.2. Following Definition 2.3, the term is divided in three parts: the set of

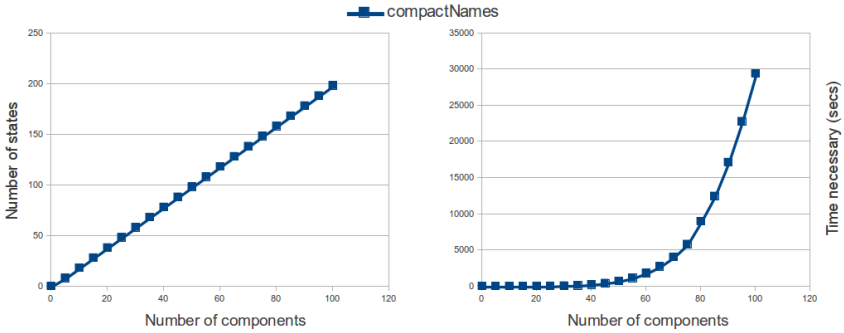

Figure 8.3: State-space sizes (left) and time necessary (right) at the varying of system size, reducing with `compactNames`

worlds W of the model (line `1`), the function d assigning a Σ-algebra to each world (lines `3-12`) and the accessibility relation $\rightsquigarrow$ over W, enriched with the counterpart relations (lines `14-29`).

In order to give an example of an entry of d, from lines `9-11` we know that the world `w(2)` has assigned the Σ-algebra containing the edges `e(0)` and `e(1)`, and nodes `n(0)` and `n(1)`. The operation source (line `10`) is defined such that `n(0)` is the source of `e(0)`, and `n(1)` is the source of `e(1)`. Conversely, the operation target (line `11`) is defined such that `n(0)` is the target of `e(1)`, and `n(1)` is the target of `e(0)`.

In order to give an example of an entry of $\rightsquigarrow$, from lines `26-27` we see that the world `w(1)` evolves in `w(2)` by renaming `e(1)` in `e(0)` and `e(2)` in `e(1)`. Interestingly, `n(0)` and `n(1)` are merged in `n(0)`, while `n(2)` is renamed in `n(1)`, i.e. it is reused the name `n(1)` which has just become free. Moreover, notice that from line `6` we know that `w(1)` contains an edge `e(0)` for which the counterpart relation is undefined. Hence edge `e(0)` is deallocated by the transition. Notice moreover that the only state with a leader is `w(3)` (line `12`), which has an edge `e(0)` with same source and target (`n(0)`). Finally, notice that, as required in Section 2.2.2 for the definition of derived temporal operators, the model has no deadlock worlds, i.e. worlds without outgoing transitions. In fact we added a self-loop on `w(3)` which preserves all its elements.

Listing 8.2: The counterpart model for four processes reduced with canonizer `freeTranspsocompactNames`

```
w(0), w(1), w(2), w(3)

w(0) |-> << e(0) e(1) e(2) e(3) n(0) n(1) n(2) n(3)
  s: {e(0)} |-> n(0), {e(1)} |-> n(1), {e(2)} |-> n(2), {e(3)} |-> n(3)
  t: {e(0)} |-> n(1), {e(1)} |-> n(2), {e(2)} |-> n(3), {e(3)} |-> n(0) >>,
w(1) |-> << e(0) e(1) e(2) n(0) n(1) n(2)
  s: {e(0)} |-> n(0), {e(1)} |-> n(1), {e(2)} |-> n(2)
  t: {e(0)} |-> n(1), {e(1)} |-> n(2), {e(2)} |-> n(0) >>,
w(2) |-> << e(0) e(1) n(0) n(1)
  s: {e(0)} |-> n(0), {e(1)} |-> n(1)
  t: {e(0)} |-> n(1), {e(1)} |-> n(0) >>,
w(3) |-> << e(0) n(0) s: {e(0)} |-> n(0) t: {e(0)} |-> n(0) >>

w(0) = e(0) |~> e(0), e(1) |~> e(1), e(3) |~> e(2),
       n(0) |~> n(0), n(1) |~> n(1), n(2) |~> n(2), n(3) |~> n(2) => w(1),
w(0) = e(0) |~> e(0), e(2) |~> e(1), e(3) |~> e(2),
       n(0) |~> n(0), n(1) |~> n(1), n(2) |~> n(1), n(3) |~> n(2) => w(1),
w(0) = e(0) |~> e(2), e(1) |~> e(0), e(2) |~> e(1),
       n(0) |~> n(2), n(1) |~> n(0), n(2) |~> n(1), n(3) |~> n(2) => w(1),
w(0) = e(1) |~> e(0), e(2) |~> e(1), e(3) |~> e(2),
       n(0) |~> n(0), n(1) |~> n(0), n(2) |~> n(1), n(3) |~> n(2) => w(1),
w(1) = e(0) |~> e(0), e(2) |~> e(1),
       n(0) |~> n(0), n(1) |~> n(1), n(2) |~> n(1) => w(2),
w(1) = e(0) |~> e(1), e(1) |~> e(0),
        n(0) |~> n(1), n(1) |~> n(0), n(2) |~> n(1) => w(2),
w(1) = e(1) |~> e(0), e(2) |~> e(1),
       n(0) |~> n(0), n(1) |~> n(0), n(2) |~> n(1) => w(2),
w(2) = e(0) |~> e(0), n(0) |~> n(0), n(1) |~> n(0) => w(3),
w(2) = e(1) |~> e(0), n(0) |~> n(0), n(1) |~> n(0) => w(3),
w(3) = e(0) |~> e(0), n(0) |~> n(0) => w(3)
```

So far we focused on the generation of counterpart models. However, before analyzing properties against the generated models we have to provide a further module to *connect* the counterpart model generator with the model checker itself.

Intuitively, the formulae of our logic are parametric with respect to the signature Σ of the structures assigned to the worlds of the model, i.e. in this case graphs. Hence we have to provide informations about the first- and second-order variables and the other *terms* which can appear in formulae. In particular where we define the first- and second-order variables and the terms for the signature of graphs.

We exemplify the module, named `TEST-CTMODELCHECKER-LE`, in Listing 8.3. For easiness of presentation we omit irrelevant details.

Listing 8.3: The Maude code to connect the counterpart model generator with the model checker

```
1  mod TEST-CTMODELCHECKER-LE is
2
3    pr TEST-COUNTERPARTMODELGENERATION-LE .
4
5    eq NOVEL-SEMANTICS = true . --- if true, then uses the novel semantics
6
7    *** --- First-order variables with sort
8    sorts FONodeVariable FOEdgeVariable .
9    subsorts FONodeVariable FOEdgeVariable < FOVariable .
10   op xE : Nat -> FOEdgeVariable [ ctor ] .
11   op xN : Nat -> FONodeVariable [ ctor ] .
12
13   *** --- Second-order variables with sort
14   sorts SONodeVariable SOEdgeVariable .
15   subsorts SONodeVariable SOEdgeVariable < SOVariable .
16   op XN : Nat -> SONodeVariable [ ctor ] .
17   op XE : Nat -> SOEdgeVariable [ ctor ] .
18
19   *** --- terms with sort
20   sorts NodeTerm EdgeTerm .
21   subsorts NodeTerm EdgeTerm < Term .
22   subsorts Node FONodeVariable < NodeTerm .
23   subsorts Edge FOEdgeVariable < EdgeTerm .
24
25   --- Compound terms
26   sort CompoundNodeTerm .
27   subsort CompoundNodeTerm < NodeTerm .
28   subsort CompoundNodeTerm < CompoundTerm .
29   ops s t : EdgeTerm -> CompoundNodeTerm [ ctor ] .
30
31   vars xe1 xe2 xe3 xe4 : FOEdgeVariable . vars xn1 xn2 : FONodeVariable .
32   op presentEdge : FOEdgeVariable FOEdgeVariable -> Formula .
33   eq presentEdge(xe1,xe2) = exists xe2 . xe1 = xe2 .
34   op presentNode : FONodeVariable FONodeVariable -> Formula .
35   eq presentNode(xn1,xn2) = exists xn2 . xn1 = xn2 .
36
37   op leader : FOEdgeVariable FOEdgeVariable -> Formula .
38   eq leader(xe1,xe2) = presentEdge(xe1,xe2) and s(xe1) = t(xe1) .
39
40   op atMostOneLeader : FOEdgeVariable FOEdgeVariable FOEdgeVariable
         FOEdgeVariable -> Formula .
41   eq atMostOneLeader(xe1,xe2,xe3,xe4)
42    = ((leader(xe1,xe3) and leader(xe2,xe4)) -> (xe1 = xe2)) .
43
44 endm
```

In line `5` we choose the semantics to be used, i.e. either the *novel* semantics of Section 2.4.2, or the original one of (GLV10).

The informations regarding the variables of the logic are provided in lines `8-17`. Intuitively, we define first- and second-order variables for the sort `Edge` (`FOEdgeVariable` and `SOEdgeVariable`) and for the sort `Node` (`FONodeVariable` and `SONodeVariable`). For example, `xE(0)`

and `XE(0)` are, respectively, examples of first- and second-order `Edge` variables.

Informations regarding the terms that may appear in the membership operator are instead provided in the lines `20-23`. As it may be expected, in line `20` we define terms for edges and for nodes, respectively `EdgeTerm` and `NodeTerm`. Clearly, elements with sort `Node` and first-order node variables are `NodeTerm` (line `22`), and similarly for edges (line `23`). Moreover, we can also have *compound terms*, namely the *source* and the *target* of a term with sort `EdgeTerm` are (compound) `NodeTerm` (lines `26-29`). Finally, in lines `31-42` we define some derived operators from Example 2.6 which will help us in defining the seven considered formulae. Namely, **present** (`presentEdge` and `presentNode` of lines `32-35`), **leader** (`leader` of lines `37-38`), and **atMostOneLeader** (`atMostOneLeader` of lines `40-42`).

We can now finally evaluate the seven mentioned properties. By evaluating **p1** $\equiv AF[\exists x.(\textbf{leader}(x))]$ we search for those worlds such that for all departing paths from them, eventually a world containing a leader is met. Evaluating the formula with empty context we expect to obtain the set of pairs composed by all the worlds (paired with empty context). In fact, from every world there exists a path to `w(3)`. This is confirmed in the lines `1-4` of Listing 8.4.

By evaluating **p2** $\equiv \exists x.\exists y.[\neg\textbf{atMostOneLeader}(x,y)]$ we search for erroneous states, namely for states having more than a leader. We expect to obtain the empty set of pairs, meaning that no world of the model falls in this case. This is confirmed by Listing 8.4, lines `5-8`.

By evaluating **p3** $\equiv \exists x.[AF(\textbf{leader}(x))]$ we search for those worlds having a process which will eventually become leader following any of the outgoing transitions. The only such state is `w(3)`, which has a leader (`e(0)`), and self-loop preserving it as unique outgoing transition. This is confirmed by the lines `9-12` of Listing 8.4.

By evaluating **p4** $\equiv \exists x.(\textbf{leader}(x))$ we search for those worlds having a leader. As confirmed in lines `13-16` of Listing 8.4, the only world having assigned a structure with a leader is `w(3)`.

Listing 8.4: Evaluating formulae against the model of Listing 8.2

```
1  *** p1
2  red in TEST-CTMODELCHECKER-LE :
3   [| AF exists xE(0) . leader(xE(0),xE(1)) |] << initLE(3) >> .
4  result: (w(0), empty), (w(1), empty), (w(2), empty), (w(3), empty)
5  *** p2
6  red in TEST-CTMODELCHECKER-LE :
7   [| exists xE(0) . exists xE(1) . not atMostOneLeader(xE(0),xE(1)) |] <<
        initLE(3) >> .
8  result: (empty).Set{Pair}
9  *** p3
10 red in TEST-CTMODELCHECKER-LE :
11  [|  exists xE(0) . AF(leader(xE(0),xE(1))) |] << initLE(3) >> .
12 result: (w(3), empty)
13 *** p4
14 red in TEST-CTMODELCHECKER-LE :
15  [| exists xE(0) . leader(xE(0),xE(1)) |] << initLE(3) >> .
16 result: (w(3), empty)
17 *** p5
18 red in TEST-CTMODELCHECKER-LE :
19  [| EF(leader(xE(0),xE(1))) |] << initLE(3) >> .
20 result: (w(0), (xE(0) |-> e(0))), ... (w(0), (xE(0) |-> e(3))),
21         (w(1), (xE(0) |-> e(0))), ... (w(1), (xE(0) |-> e(2))),
22         (w(2), (xE(0) |-> e(0))), (w(2), (xE(0) |-> e(1))),
23         (w(3), (xE(0) |-> e(0)))
24 *** p6
25 red in TEST-CTMODELCHECKER-LE :
26  [| (xN(0) =/= xN(1)) and (AF(presentNode(xN(0),xN(2)) and
27      presentNode(xN(1),xN(3)) and (xN(0) = xN(1))))  |] << initLE(3) >> .
28 result: (w(0), (xN(0) |-> n(0), xN(1) |-> n(1))),
29         (w(0), (xN(0) |-> n(0), xN(1) |-> n(2))),
30         (w(0), (xN(0) |-> n(0), xN(1) |-> n(3))),
31         (w(0), (xN(0) |-> n(1), xN(1) |-> n(2))),
32         (w(0), (xN(0) |-> n(1), xN(1) |-> n(3))),
33         (w(0), (xN(0) |-> n(2), xN(1) |-> n(3))), ...
34 *** p7
35 red in TEST-CTMODELCHECKER-LE :
36   [| presentEdge(xE(0),xE(1)) and (xN(0) = s(xE(0))) and (xN(1) = t(xE(0)))
37      and <>( (not presentEdge(xE(0),xE(1))) and (xN(0) =/= xN(1)) ) |] <<
           initLE(3) >> .
38 result: (empty).Set{Pair}
```

From the evaluation of **p5** $\equiv EF($**leader**$(x))$ we expect to find those processes for which there exists an execution leading to their election to leader. For every process there exists a path leading to its election. Hence we expect to obtain a set of pairs whose assignment components map the variable to each of the edges in the model. This is confirmed by Listing 8.4, lines `17-23`

Evaluating **p6** $\equiv (x \neq y) \wedge AF[$**present**$(x) \wedge$ **present**$(y) \wedge (x = y)]$ we search for worlds having two nodes which will collapse for any possible outgoing path. All the nodes get merged in the node `n(0)` of world `w(3)`,

hence for each world, all the nodes will eventually merge. Evaluating the formula (with context composed by the two node variables) we obtain, for each world, the set of pairs whose assignment components map the two node variables to all the possible pairs of nodes. In lines `24-33` of 8.4 we exemplify only the pairs having `w(0)` as first component (and moreover we omit the symmetric pairs with inverted assignments for the two node variables).

Finally, evaluating **p7** we search for worlds that can evolve deallocating a process without merging its communication ports. Hence we search for erroneous evolutions in which communication ports (nodes) are not correctly updated after the deallocation of a process (i.e. an edge). As depicted in lines `34-38` of 8.4, the formula is evaluated as the empty set of pairs, meaning that no world of the model falls in this case.

8.2 Dining philosophers with disposable forks

System description. In this section we consider as test case a particular version of the well-known dining philosophers problem (Dij71) along the lines of the case study used in (DRK02), where *Allocational Temporal Logic*, a first-order extension of LTL, is proposed to reason about the allocation and dealloactionn of entities.

The dining philosophers problem is a classical example often used to illustrate the synchronization issues of concurrent algorithms, where participants compete for the exclusive use of resources. Following the classical formulation, the problem consists of five philosophers that sit at a table. On the table there are five forks, and each philosopher has one on right and one on left. Intuitively, the two forks on the left and on the right are shared, respectively, with the neighbour philosopher on the left and on the right. This configuration is illustrated in Figure 8.4 [1].

The evolution of the system consists in the philosophers alternately thinking and eating, with the requirement that a philosopher can eat only when he/she owns both the left and the right forks. Each fork can be owned by only one philosopher at time, hence a philosopher can grab one

[1] From `http://en.wikipedia.org/wiki/Dining_philosophers_problem`

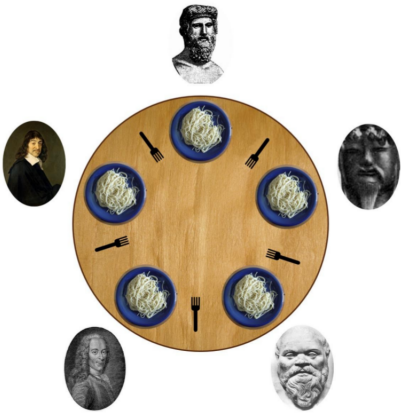

Figure 8.4: An illustration of the dining philosophers problem

of its two left and right forks only if it is not currently used by another philosopher.

Initially all the philosophers think, and the forks are on the table. Once a philosopher owns two forks, he/she eats for a while, and then puts back the forks on the table, making them available to neighbour philosophers, and starts thinking. In particular, a philosopher can grab a fork on right or left as soon as they are available, but he/she can't start eating yet, because he/she has to wait to get both of them.

The amount of food in each plate is considered to be infinite, however, considering the status of the philosophers (i.e. `think`, `wait` and `eat`), and the number and distribution of forks on the table, it is easy to see that the system has only a finite number of distinct states which are possibly repeated infinitely often.

Then, the problem is that of defining a concurrent algorithm that guarantees that no philosopher will starve, i.e. that each philosopher will alternate between eating and thinking infinitely often, hence avoiding deadlock states, that is states from which a philosopher will not be able to evolve. Intuitively, this happens in the case in which all the philosophers own a fork, and are waiting for the second one.

The variant proposed in (DRK02) regards as usual philosophers and forks. However, forks have their own identifier, and are considered as messages which are *consumed* every time they are grabbed by a philosopher. In particular, the action of taking a fork from the table performed by a philosopher is represented by the destruction of the fork, while putting the fork back on the table is modelled by creating a new fork. Newly generated forks are actually new entities, meaning that once a fork is used, it is destroyed, as the original purpose of (DRK02) was to reason about dynamic resource allocation.

Due to the creation of infinitely many new forks, this version of the dining philosopher problem has an infinite number of distinct states, leading to infinite-state models, which hence at first sight appear not verifiable in our framework.

However, fortunately, following our approach it is possible to precisely represent this infinite-state system with a finite model. The intuition is very simple: in our setting names are local to single states, meaning that the same name does not identify entities (e.g. forks) across different states. Even more importantly, names may represent distinct entities even in the same state of a model, all that matters is the counterpart relation of the considered incoming transition.

Consider the example of Figure 3.3 (right), where it is depicted a counterpart model having only one state w_0 containing only one element u_1, and a loop on w_0 with empty counterpart relation as unique transition. Then, it is easy to understand that, thanks to the empty counterpart relation, this model actually represents an infinite-state system which evolves by deallocating the current element (u_1), and creating a new distinct one, to which it is reassigned the name u_1.

We can hence model the state-space of the above discussed variant of the dining philosophers problem by assigning the names of the previously deallocated forks to the newly generated ones. Note that the represented behaviour is exactly the discussed one. Namely, forks are consumed after usage, and new ones are created to replace them. As we will see, intuitively, we defined a total ordering among forks names (e.g. `fork(i)` $<$ `fork(j)` if $i < j$), and a minimal one (e.g. `fork(0)`). Then,

every time we create a new fork, we assign to it the minimal currently unused fork name.

Notice moreover that the system is resource bounded, as a new fork can be created only after that another one has been previously deallocated, and in particular there will never be more than n forks, with n the number of philosophers. This, together with the mentioned reusing of forks names, guarantees the finiteness of the model, which is hence amenable to verification.

As we will see, we can also further exploit other name reusing techniques to reduce the obtained model. In a system with n philosophers we can have at most n forks with names ranging from the minimal one, to its n successors. At each step, at most a fork can be consumed by a philosopher. Then we can use the *compactName* canonizer, adopted also for the leader election case, to compact the names of the forks. By resorting to this technique we obtain a model such that in a state with m forks we will have their names going from the minimal one, to its m successors (rather than to n), independently from which forks have been previously deallocated.

More interestingly, the system presents a couple of regularities (i.e. symmetries) that can be exploited in the form of c-reductions and that happen to yield bisimulations which we can exploit to further reduce the size of the state-space of the model. Namely, the full symmetry of forks, and the rotational symmetry of philosophers.

The names of the forks do not play any role in the definition of the dynamics of the system. Hence we can freely transpose them, obtaining bisimilar states. Moreover, the dining philosopher problem is an emblematic example of *rotational symmetry*, where the transposition of the names of the philosophers could possibly lead to not bisimilar states. Instead, by rotating them all at once, we obtain bisimilar states. The intuition is that it is not important the *absolute position* of a philosopher in the table, what matters is instead its relative position with respect to its neighbours (the two philosophers with which it shares its two forks), in the sense that they have to be preserved.

System specification. We can easily represent states of this system as algebras of the signature used for the leader election case, that is unlabelled directed graphs. We use nodes to represent philosophers, and edges to represent forks. The two philosophers surrounding a fork are actually represented as the source and target nodes of the fork. Intuitively, for one of the two philosophers it will be the left fork, while for the other it will be the right one.

Then, the initial state of the system is, as for the case of the leader election example, a graph where edges are connected in a ring topology. During the execution of the system, edges (forks) are deallocated and reallocated. Hence, differently from the leader election case, during the evolution of the system the ring topology will get broken.

Actually, we have to encode also the statuses of the philosophers in the state representation. Namely, `thinking` when it is idle, `waiting` when it owns a fork and it is waiting for the other one, and `eating` when it owns two forks, and can hence eat. In the current version of our framework we treat plain algebras, hence we do not have entities with attributes (e.g. we cannot directly model attributed graphs). In order to encode statuses informations we extend the signature of graphs with a third sort `Status` whose elements are {`think`,`wait`,`eat`}, and with an unary operation `status` going from the sort of nodes (philosophers) to the one of statuses.

Notice that statuses are now first-class elements exactly as nodes and edges. Hence we can express properties regarding them. Notice moreover that statuses will be preserved by every transition. This shows the generality of our approach.

Finally, for implementation choice, we decided to explicitly encode informations regarding the left- and right-neighbours of a philosopher. We then further extended the considered signature with two unary operations `leftNeighbour` and `rightNeighbour` going from the sort of nodes to the one of nodes (i.e. between philosophers).

As done in sections 8.1 and 6.1 for the case of the leader election system, we now discuss in detail the Maude code that has to be provided to define the above described system.

Listing 8.5: The Maude code to define the signature of the dining philosophers

```
1  mod DP-SIGNATURE is
2
3    pr CT-MODEL-SORTS .
4
5    --- the sorts: nodes = philosophers, edges = forks
6    sorts Node Edge Status .
7    subsorts Node Edge Status < Element .
8    ops Node Edge Status : -> Cid [ctor] .
9
10   --- elements constructors
11   op e : Nat -> Edge [ctor] .
12   op n : Nat -> Node [ctor] .
13   ops think wait eat : -> Status [ctor] .
14
15   --- the names of the five operations
16   ops s:_ t:_   : Map{NAryArgument,Element} -> Operation [ctor] .
17   op st:_       : Map{NAryArgument,Element} -> Operation [ctor] .
18   ops lN:_ rN:_ : Map{NAryArgument,Element} -> Operation [ctor] .
19
20   var source target : Map{NAryArgument,Element} .
21   var statuses lNeigh rNeigh : Map{NAryArgument,Element} .
22   var cr : CounterpartRelation . var conf : Configuration .
23
24   --- Conservatively apply cr to the operations
25   eq conservativelyApplyToOperations(cr, s: source conf)
26    = conservativelyApplyToOperations(cr,           conf)
27      s: applyToOperationIfDefined(cr, source) .
28   eq conservativelyApplyToOperations(cr, t: target conf)
29    = conservativelyApplyToOperations(cr,           conf)
30      t: applyToOperationIfDefined(cr, target) .
31   eq conservativelyApplyToOperations(cr, st: statuses conf)
32    = conservativelyApplyToOperations(cr,              conf)
33      st: applyToOperationIfDefined(cr, statuses) .
34   eq conservativelyApplyToOperations(cr, lN: lNeighbours conf)
35    = conservativelyApplyToOperations(cr,                 conf)
36      lN: applyToOperationIfDefined(cr, lNeighbours) .
37   eq conservativelyApplyToOperations(cr, rN: rNeighbours conf)
38    = conservativelyApplyToOperations(cr,                 conf)
39      rN: applyToOperationIfDefined(cr, rNeighbours) .
40
41   --- Singleton with sort SystemInfo providing the size of the system (i.e
         number of philosophers). It is necessary to exploit rotational
         symmetry.
42   sort SystemInfo . subsort SystemInfo < Element .
43   op SystemInfo : -> Cid [ctor] .
44   op info : NzNat -> SystemInfo [ctor] .
45
46 endm
```

Listing 8.5 shows the Maude code necessary to define the discussed signature for dining philosophers. As discussed in Chapter 6, we provide the sorts of the signature (lines 6-8), namely nodes (philosophers), edges (forks) and the statuses of the philosophers.

Then lines 11-13 provide the constructors of the elements of the algebras. Terms `e(0)`, `e(1)` and `n(0)`, `n(1)` exemplify possible edges and nodes. The constructors for statuses (line 13) are actually the three constants `think`, `wait` and `eat`, meaning that those are the only existing elements with sort `Status`.

The rest of the module regards the operations of the signature. The names of the operations of graphs are provided in lines 16 (source and target), while lines 17-18 provide the names of the three newly introduced operations, namely `st` (i.e. status), associating a status to each philosopher, and `lN` and `rN` (i.e. leftNeighbour and rightNeighbour) associating to each philosopher, respectively, its left and right neighbour.

Finally, the simple equations in lines 25-39, one for each operation, specify how a counterpart relation is conservatively applied to a configuration (i.e. a state of the system). As discussed in Chapter 6, this is used during the generation of a new state, and, intuitively, we again only have to specify the names of the operations.

The module concludes with lines 42-44, providing the definition of the sort `SystemInfo`, and the constructor `info`, taking as argument a non-zero natural number. This sort is meant to be instantiated by a singleton providing the size of the system, that is the number of philosophers in the system. This is not important right now, as we will use it to exploit the rotational symmetry of the philosophers.

The next step in defining the system is that of providing the necessary Maude module to specify its initial state. An example is proposed in Listing 8.6. Lines 7-10 define the operation `initDP` which takes as parameter a natural number `size` greater than 2, and generates the initial state of the discussed dining philosophers system with `size` philosophers. We actually defined `initDP-Elements` (lines 12-14) to generate `size` philosophers, `size` forks, the three statuses `think`, `wait` and `eat`, and the singleton `info(size)`.

In order to build the Σ-operations of the algebras, in lines 16-20 we defined `initDP-Operations`, which in turn resorts on the utilities `initSource`, `initTarget`, `initStatus`, `initLNeighbours`, as well as `initRNeighbours`, specified in the rest of the module (lines 22-43).

Listing 8.6: The Maude code to define the initial state of the dining philosophers system

```
1  mod INSTANCES-DP is
2
3    pr DP-SIGNATURE .
4
5    var m size : Nat .
6
7    op initDP : NzNat -> Configuration .
8    ceq initDP(size)
9     = initDP-Elements(size) initDP-Operations(size,size) info(size)
10   if size > 2 .
11
12   op initDP-Elements : NzNat  -> Configuration .
13   eq initDP-Elements(s(m)) = e(m) initDP-Elements(m) n(m) .
14   eq initDP-Elements(0) = think wait eat .
15
16   op initDP-Operations : NzNat NzNat -> Configuration .
17   eq initDP-Operations(size,size)
18    = s: initSource(size) t: initTarget(size,size)
19      lN: initLNeighbours(size,size) rN: initRNeighbours(size,size)
20      st: initStatus(size) .
21
22   ops initSource initStatus : NzNat -> Map{NAryArgument,Element} .
23   op initTarget : NzNat NzNat -> Map{NAryArgument,Element} .
24   op initLNeighbours : NzNat NzNat -> Map{NAryArgument,Element} .
25   op initRNeighbours : NzNat NzNat -> Map{NAryArgument,Element} .
26
27   eq initSource(s(m)) = initSource(m), {e(m)} |-> n(m) .
28   eq initSource(0) = empty .
29
30   eq initTarget(s(m),size)
31    = initTarget(m,size), {e(m)} |-> n(s(m) rem size) .
32   eq initTarget(0, size) = empty .
33
34   eq initStatus(s(m)) = initStatus(m), {n(m)} |-> think .
35   eq initStatus(0) = empty .
36
37   eq initRNeighbours(s(m),size)
38    = initRNeighbours(m,size), {n(m)} |-> n(s(m) rem size) .
39   eq initRNeighbours(0, size) = empty .
40
41   eq initLNeighbours(s(m),size)
42    = initLNeighbours(m,size), {n(s(m) rem size)} |-> n(m) .
43   eq initLNeighbours(0, size) = empty .
44
45 endm
```

We already discussed `initSource` and `initTarget` in the case of the leader election system: we now focus on the newly introduced ones. With `initStatus` (lines 34-35) we assign the status `think` to every philosopher, while with `initLNeighbours` and `initRNeighbours` (lines 37-43) we assign to each philosopher the *following* one as right neighbour, and the *preceding* one as left neighbour. Where, considering

philosopher `n(m)`, by *following* we mean the philosopher `n(m+1)` (and `n(0)` for the last philosopher). Conversely, by `preceding` we mean `n(m-1)` (or `n(size-1)` for `n(0)`).

Notice that, differently from other formalizations of the problem, the choice of the neighbourhood and of the names of the forks is arbitrary, as we do not use names of elements to implicitly indicate their positions. We instead explicitly represent this information in the operations of the algebra. For this reason, the forks present full symmetry, meaning that we can freely transpose their names without affecting the behaviour of the system. This allows us to exploit the canonizer `freeTransps` for the sort of forks.

Finally, the five rules specifying the dynamics of the system are given in Listing 8.7.

The first four are intuitively pairwise symmetric. The first two regard the acquisition of the first fork by a philosopher in status `think`. In particular, with the first one (`think2wait-source`) a philosopher matched with the variable `ns` grabs the fork of which it is the source node, while with the second one (`think2wait-target`), a philosopher matched with `nt` grabs the fork of which it is the target node. Intuitively, in a case the philosopher will grab its left fork, while in the other he will grab its right one.

We discuss in more detail only `think2wait-source`. In order to apply the rule on a state it is necessary to match `ns` with a node, and `el` with an edge (line `11`). More precisely, from `statuses, ns |-> think` of line `11`, we know that the matched node has to be in status `think`. Moreover, from line `12` we know that the node has to be the source of the edge `el`.

Then, in line `14` we see that the application of the rule involves the deallocation of `el`, and the changing of the status of `ns` in `wait`. In line `15` we see that the operations source and target are updated to handle the deallocation of `el`. Notice that, in general, it is necessary to explicitly handle only the operations matched by the LHS of a local rule, while the others are automatically updated by the application of the global rule. Finally, from line `16` we see that `ns` is preserved.

Listing 8.7: The Maude code to specify the dynamics of the dining philosophers system

```
mod BEHAVIOUR-DP is
  pr DP-SIGNATURE .
  var e1 : Edge . vars ns nt nr : Node . var cr : CounterpartRelation .
  vars source target : Map{NAryArgument,Element} . vars i l : Nat .
  vars statuses lNeigh rNeigh : Map{NAryArgument,Element} .

  eq minId-2CompactAndAllocate(Edge) = e(0) .
  eq successor-2CompactAndAllocate(e(i)) = e(i + 1) .

  crl [think2wait-source] :
    ns e1 st: (statuses, {ns} |-> think)
    s: (source, {e1} |-> ns) t: (target, {e1} |-> nt)
  => {cr}(
    ns    st: (statuses, {ns} |-> wait )
    s: (source             ) t: (target             ) )
  if cr := (ns |~> ns) .

  crl [think2wait-target] :
    nt e1 st: (statuses, {nt} |-> think)
    s: (source, {e1} |-> ns) t: (target, {e1} |-> nt)
  => {cr}(
    nt    st: (statuses, {nt} |-> wait)
    s: (source             ) t: (target             )  )
  if cr := (nt |~> nt) .

  crl [wait2eat-source] :
    ns e1 st: (statuses, {ns} |-> wait)
    s: (source, {e1} |-> ns) t: (target, {e1} |-> nt)
  => {cr}(
    ns st: (statuses, {ns} |-> eat)
    s: (source             ) t: (target             ) )
  if cr := (ns |~> ns) .

  crl [wait2eat-target] :
    nt e1 st: (statuses, {nt} |-> wait)
    s: (source, {e1} |-> ns) t: (target, {e1} |-> nt)
  => {cr}(
    nt st: (statuses, {nt} |-> eat)
    s: (source             ) t: (target             ) )
  if cr := (nt |~> nt) .

  crl [eat2think] :
    n(i)                          st: (statuses, {n(i)} |-> eat)
    lN: ({n(i)} |-> n(l), lNeigh) rN: ({n(i)} |-> nr, rNeigh)
    s: source
    t: target
  => {cr}(
    n(i) new(Edge,0) new(Edge,1) st: (statuses, {n(i)} |-> think)
    lN: ({n(i)} |-> n(l), lNeigh) rN: ({n(i)} |-> nr, rNeigh) )
    if l < i --- i == (size -1)
     then s: (source, {new(Edge,0)} |-> n(l), {new(Edge,1)} |-> n(i))
          t: (target, {new(Edge,0)} |-> n(i), {new(Edge,1)} |-> nr)
     else s: (source, {new(Edge,0)} |-> n(i), {new(Edge,1)} |-> n(l))
          t: (target, {new(Edge,0)} |-> nr, {new(Edge,1)} |-> n(i)) fi
  if  cr := (n(i) |~> n(i)) .
endm
```

The rules `wait2eat-source` and `wait2eat-target` are very similar to, respectively, `think2wait-source` and `think2wait-target`, the only difference being that those rules model the acquisition of the second fork, rather than of the first one. In fact, looking at the rules, the only differences are that the statuses of the matched nodes have to be `wait`, and are updated to `eat`. Hence, the application of these two rules change the status of a philosopher from `waiting` for the second fork to `eating`.

More interesting is the last rule `eat2think` of lines `42`-`55`. The application of this rule allows an eating philosopher to get back to status `think`, and to release the two forks. Actually, as previously discussed, forks are not *released*, because they get destroyed when a philosopher grabs them. Instead, two new forks are created and laid on the table.

From line `43` we see that the application of the rule involves a matching of `n(i)` with a node in status `eat`. Notice that here the variable to be matched is `i`. While from line `44` we know that the two neighbours of (the node matched by) `n(i)` are the ones matched by `n(l)` and `nr`. Notice that here we match `n(l)` with the left neighbour, and `nr` with the right one. Here the matched variables are `l` and `nr`.

Then, from line `48` we know that the application of the rule involves the creation of two new forks, in fact `new` can be thought of as a function that creates a new element with sort provided as parameter. The chosen name will be the least currently unused one, hence we will reuse one of the names of the forks previously deallocated. The second parameter of `new` (i.e. a natural number) is used to distinguish the many elements that could be created by a single rule, and the ones with smaller parameter will have assigned a smaller name, because will be treated first. From line `48` we also now that the status of `n(i)` is changed in `think`.

Also lines `50`-`54` are important: there we update the operations source and target to account for the creation of the two new edges. Intuitively, `n(i)` will be the source of one edge and the target of the other one, while one of its two neighbours will be the source of one, and the other neighbour will be the target of the other one.

Intuitively, any (admissible) combination could be chosen without

affecting the behaviour of the system. However, in order to obtain smaller state-spaces, we use the `if_then_else_fi` statement to obtain states where edges are ordered from the one having the minor node as source to the one having the greater one as target, as it happens for the initial state (considering the order given by the indexes of the names of the elements, e.g. `n(i)` $<$ `n(j)` and `e(i)` $<$ `e(j)` if $i < j$).

From the definition of `initLNeighbours` and `initRNeighbours` of Listing 8.6, we know that a node is minor than its left neighbour except for one limit case: the one in which `n(i)` is matched with `n(size-1)`, whose left neighbour is the smaller `n(0)`. Hence, the `then` case (for `l` $<$ `i`) corresponds to the case in which `n(l)` is matched with `n(0)` (and hence `n(i)` is matched with `n(size-1)`). Conversely, the `else` case corresponds to the other cases.

Finally, in line `53` we see that we leave unmodified `lN` and `rN`, while the counterpart relation defined in line `62` preserves `n(i)`.

We still have to discuss the two equations in line `7-8` of Listing 8.7. Those are two equations that have to be provided for every sort of which new elements can be created as result of the application of a rule. As we just discussed, the only elements which are created (rule `eat2think`, line `48`) are edges.

We previously intuitively explained that every time we create a new element, the application of the global rule assigns to it the *minimal* currently unused name for its sort. Then, it should be provided a (total) ordering for the names of the interested sorts.

In the first equation (`minId-2CompactAndAllocate`, line `7`) we provide the minimal name of the sort `Edge`, in this particular example `e(0)`. While with the equation of line `8` we provide the successor name of the one given as parameter, in this particular example we simply increment by one the index of the name given as parameter.

It may be worth to point out that the dynamics of Listing 8.7 (as well as the ones proposed in (DRK02)) implement a *naive* solution to the dining philosopher problem, where deadlocks are not prevented. In fact, if every philosopher grabs a fork, and changes its status in wait, then no philosopher will be anymore able to evolve, and the system will be in

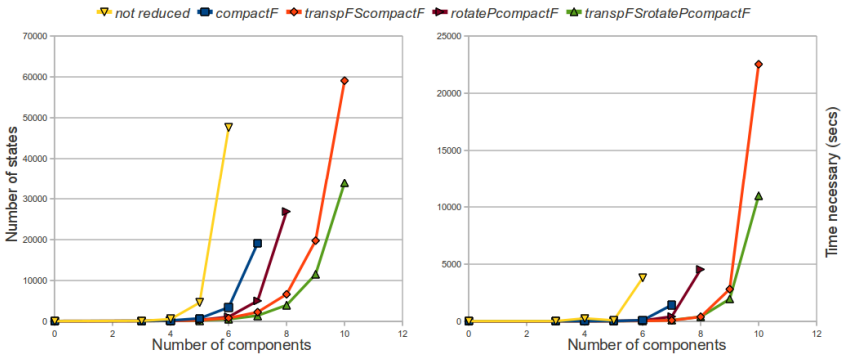

Figure 8.5: State-space sizes (left) and time necessary for their generation (right) at the varying of system size

deadlock.

The problem of dining philosophers is very well studied, and has several solutions. However this is not the aim of this section, as we just want to show how it is possible to model such scenario in our framework.

Counterpart model generation. In this paragraph we discuss the performances of our tool in generating the counterpart models at the varying of the size of the system, i.e. the number of philosophers. In particular, due to the size of the state-space, we are not able to build unreduced models with size greater than 6, by using with a standard laptop. However, by resorting to our c-reductions, we are able to generate counterpart models for greater sizes. In particular, we exploit the canonizers `compactNames`, `freeTransps`, `rotations` and their combinations.

Figure 8.5 (left) reports informations about the number of worlds of the counterpart models generated varying the size of the system from 3 to 10. While, Figure 8.5 (right) reports informations about the time necessary to generate such models. The graphics regard five cases: the unreduced counterpart models (*not reduced*), the ones obtained resorting to the canonizer `compactNames` to compact the names of the forks (*compactF*), the ones obtained by first compacting the names of the forks and then transposing forks and statuses, resorting to canonizer

`freeTranspsocompactNames` (*transpFScompactF*), the ones obtained by first compacting the names of the forks and then rotating philosophers, resorting to canonizer `rotationsocompactNames` (*rotatePcompactF*), and, finally, the ones obtained applying the three reductions, i.e. by using `freeTranspsorotationsocompactNames` (*transpFSrotatePcompactF*).

Before discussing the graphics of Figure 8.5, in Listings 8.8 and 8.9 we provide the system-specific code to exploit the three canonizers. For easiness of presentation we focus on the canonizer `rotations`, as in Chapter 7 and Section 8.1 we already provided the (similar) code to exploit the other two canonizers for the case of the leader election system.

In Chapter 7, and more precisely in Listings 7.4 and 7.5, we have seen that in order to exploit the canonizer `freeTransps` for the leader election system, we have to provide a module to compare states (`CODEFOR-STATE-ORDERING-LE`), a module to provide informations about transpositions (i.e. `CODEFOR-TRANSPOSITIONS-LE`), and a module importing all the necessary modules and specifying the canonizer to be used. As depicted in Listings 8.8 and 8.9, the case of rotations is specular. We in fact provide a module to compare states (`CODEFOR-STATE-ORDERING-DP`), a module to provide informations on rotations (`CODEFOR-ROTATIONS-DP`), and the module `TEST-COUNTERPARTMODELGENERATION-DP` importing the necessary modules and specifying the used canonizer.

We first focus on Listing 8.8. We do not report the body of `CODEFOR-STATE-ORDERING-DP`, as it is similar to the one of the leader election case. In module `CODEFOR-ROTATIONS-DP` we first indicate the sorts to which we want to apply the canonizer, in this case `Node` (lines `20`-`21`). Then we specify how rotations are applied to single elements (line `24`), and how they are propagated to a state (lines `27`-`32`). Intuitively, in this last case we only have to specify the operations composing a state of the dining philosophers system.

Considering module `TEST-COUNTERPARTMODELGENERATION-DP` of 8.9, the only interesting lines are `19`-`28`, where we choose which canonizer (or their combination) we want to use. In the case depicted in Listing 8.9 we use the combination of the canonizer `compactNames` with the canonizer `rotations` (line `26`).

Listing 8.8: The Maude code to exploit `freeTransps rotations` and `compactNames`(1)

```
mod CODEFOR-NAME-COMPACT-DP is
   ...
endm

mod CODEFOR-STATE-ORDERING-DP is
  ...
endm

mod CODEFOR-TRANSPOSTIONS-DP is
  ...
endm

mod CODEFOR-ROTATIONS-DP is
  pr ROTATIONAL-SIMMETRY . pr DP-SIGNATURE .

  vars i nSymElts nextRot : Nat . var cid : Cid . var conf : Configuration.
  var source target statuses lNeigh rNeigh : Map{NAryArgument,Element} .

  --- Sorts to be transposed
  eq CLASS-OF-SORTS-TO-BE-ROTATED = Node .
  eq getCid(n1:Node) = Node .

  --- Application of rotations to elements
  eq rotateElement(nextRot,nSymElts,cid,n(i)) = n((i + nextRot) rem
       nSymElts) .

  --- Application of rotations to states
  eq rotateConfiguration(nextRot,nSymElts,cid,(s: source t: target st:
       statuses lN: lNeigh rN: rNeigh conf))
   = s:  rotateOperation(nextRot,nSymElts,cid,source)
     t:  rotateOperation(nextRot,nSymElts,cid,target)
     st: rotateOperation(nextRot,nSymElts,cid,statuses)
     lN: rotateOperation(nextRot,nSymElts,cid,lNeigh)
     rN: rotateOperation(nextRot,nSymElts,cid,rNeigh) .
endm
```

We can now discuss the graphics of Figure 8.5, where we notice that we are able to generate the unreduced counterpart model up-to systems with 6 philosophers, obtaining 47527 states in 3783 seconds.

Considering the reduced models, we see that we are able to generate the model up-to size 7 (19034 states in 1407 seconds) for the *compactF* case. By combining the canonizers `compactNames` and `rotations` (case *rotatePcompactF*), we instead manage to generate the model up-to 8 philosophers (26819 states in 4517 seconds). In the case *transpFScompactF*, where we combine `freeTransps` and `compactNames`, we spend

Listing 8.9: The Maude code to exploit `freeTransps rotations` and `compactNames`(2)

```
mod TEST-COUNTERPARTMODELGENERATION-DP is

  pr BEHAVIOUR-DP .
  pr INSTANCES-DP .
  pr CT-MODEL-BUILDER .
  pr META-CONNECTOR .

  pr CODEFOR-NAME-COMPACT-DP .
  pr CODEFOR-STATE-ORDERING-DP .
  pr CODEFOR-TRANSPOSTIONS-DP .
  pr STATE-ORDERING-TRANSPOSESOMESORTS .
  pr CODEFOR-ROTATIONS-DP .

  eq ROOTMODULENAME = 'TEST-COUNTERPARTMODELGENERATION-DP .

  var lState : LabelledState . var size : Nat .
  var cr : CounterpartRelation . var conf1 : Configuration .

  --- 1) non reduced
  *** eq reducer(lState) = lState .
  --- 2) compact edges
  *** eq reducer(lState) = compactNames(lState) .
  --- 3) compact edges + transpose edges & status
  *** eq reducer(lState) = minimize-LexicographicalOrdering-
       TransposingSomeSorts(compactNames(lState)) .
  --- 4) compact edges + rotate nodes
  eq reducer({cr}<< info(size) conf1 >>) = rotations(compactNames({cr}<<
       info(size) conf1 >>), CLASS-OF-SORTS-TO-BE-ROTATED, size) .
  --- 5) compact edges + transpose edges & status + rotate nodes
  *** eq reducer({cr}<< info(size) conf1 >>) = minimize-
       LexicographicalOrdering-TransposingSomeSorts(minRotation(compactNames
       ({cr}<< info(size) conf1 >>), CLASS-OF-SORTS-TO-BE-ROTATED, size)) .

endm
```

22523 seconds to generate the model with 10 philosophers, containing 59048 states. Finally, by composing the three canonizers (case *transpFS-rotatePcompactF*) we are again able to generate the model regarding 10 philosophers, but in this case we obtain, roughly, half of the states in half of the time with respect to the *transpFScompactF* case (33828 states in 10953 seconds).

From these experimental results we notice that the canonizers allow to generate counterpart models with greater size in much less time. In particular, we can argue that `compactNames` is less effective than for the leader election case, as it allows to generate the model with only

one more philosopher (7, case *compactF*) with respect to the unreduced model. However, we can obtain better performances by resorting to composed canonizers. For example, by combining `compactNames` with `rotations` and then with `freeTransps` we are able to generate the model for the double of the philosophers with respect to the unreduced case (10, case *transpFSrotatePcompactF*). However, we still have an exponential (but slower) growth of the number of worlds of the generated models.

Part III

Closing part

Chapter 9

Discussion

In this thesis we proposed a framework based on a novel approach to the semantics of quantified μ-calculi inspired by Counterpart Theory of David Lewis (Lew68). Our logic allows to reason about several kinds of evolutions of system components.

We proposed counterpart models as semantic domain for our logic, namely transition systems where states are algebras, and transitions are labeled with partial morphisms between the algebras of the source and target state. Our models come as a direct generalization of graph transitions systems, where states are labelled with graphs. Counterpart models are well-suited to model systems with dynamic structure, that is systems whose components and their interconnections may vary over time.

Then we proposed a general formalization of counterpart model approximations, together with a sound approximated model checking procedure which exploits sets of under- and over-approximations to approximate the evaluation of formulae in a model.

Moreover, we presented a state-space reduction technique to reduce counterpart models to smaller behavioural equivalent ones.

Finally we validated our approach through a prototypal tool framework.

Following the structure of the thesis, in this chapter we first critically overview some quantified modal logics proposed in the literature to

reason about the evolution of system components (Section 9.1).

Then, in Section 9.2 we discuss some techniques and tools to enable verification of visual specification formalisms in general, and graph transformation in particular.

9.1 Quantified modal logics

As we mentioned in the Introduction, many authors considered quantified modal logics and have addressed their decidability and complexity issues. For instance, many efforts have been focused on defining logics (or identifying fragments) that sacrifice expressiveness in favour of efficient computability.

This section reviews some proposals for quantified modal logics, trying to sum up the differences with our own contribution, with a specific focus on those approaches developed for the verification of visual specification formalisms in general. We will treat in more detail the approaches related to graph transformation in Section 9.2.

Description logics. Logics for reasoning about knowledge change (e.g. temporal description logics) have been proposed by various authors (see e.g. (FT03; HWZ01)), either as first-order extensions of classical linear- and branching-time temporal logics such as LTL and CTL (HWZ01), or as extensions of the modal μ-calculus (FT03). The semantics is typically given in Kripke-style with a unique domain of interpretation that allows neither for the merging nor for the renaming of elements.

Decidability results are given for some fragments, e.g. the *monodic* ones, roughly consisting of equality-free formulae with a restricted number of free variables under temporal operators.

Our current approach to decidability focuses instead on the class of models considered (essentially finite-state) rather than on the logic fragments that guarantee decidability for any possible class of models. For this reason we discussed how it is possible to obtain finite-state models for the important class of resource-bounded systems. This can in fact

be obtained by resorting on state-space reduction techniques based on name-reusing, captured by our c-reductions (Section 3.3).

In the future works we might consider to extend our work to identify decidable fragments to enable verification even for some infinite-state models. In the same way, we could define less expressive logics encodable in the one proposed in this thesis, so to be able to reuse all the here presented results and techniques.

Graph transformation logics. Another interesting setting where quantified temporal logics have raised interest are graph transformation systems (Roz97; EEKR99; EEPT06), where software systems exhibiting features such as component or resource allocation, deallocation, reallocation or fusion are conveniently modelled using graph morphisms.

An example can be found in (GL07), where a graph logic was developed for encoding a spatial logic for the π-calculus (Cai04) in a graph-based setting. The logic extends the μ-calculus with a node-binding modal operator "$\Diamond_{\langle p,Y\rangle}$", basically stating that the transition between worlds is caused by a specific rule p, that may create a chosen set Y of new elements. The logic also concerns quantifiers and other ingredients along the ones in (Cou97) to describe the graphical structure of configurations. Merging and renaming is allowed for some restricted cases only.

Several other approaches exist. In Section 9.2 we discuss the ones proposed in the two research lines of (BCK04; BCKL07), and (BRKB07; Ren03), related, respectively, to the tools Augur and GROOVE.

Software model checking. Some of the authors of (BRKB07) have investigated first-order temporal logics for various other structures as well, other than graphs.

For example, in (Ren06a) they propose an extension of CTL with first- and (monadic) second-order quantification. The semantics is given in terms of *algebra automata*, i.e. automata enriched with an algebraic structure of states, and with a morphism-like transition relation that allows for renaming elements, but not for their merging.

The model checking problem over finite automata is shown to be reducible to the ordinary model checking of CTL formulae over Kripke structures, while preserving the necessary structure to exploit name symmetries. However two transformations have to be applied in order to check a property against an automata: to the formula and to the automata. The transformation on the formula is done to remove quantifiers, and leads to a blow up of the size of the formula linear in the number of quantifiers and temporal operators. The automaton is instead transformed via a *skolemization* which may result in an exponential blow-up in the maximum size of the algebras of the automata and in the nesting depth of the quantifiers in the formula.

A similar approach is followed in (DRK02), but based on LTL and including predicates to reason about allocation, deallocation and reallocation of objects. The notion of name-equipped automata allows for injective renaming, but forbids merging. The idea follows the tradition of HD-automata (MP05b), which enable name reuse for the sake of verification and bisimulation checks.

As in our proposal, the semantics of the next time modality does not discard accessible worlds where elements assigned to variables are deleted. The assignments of deallocated variables become undefined, so that the logic allows for expressing deallocation. However, equality over undefined variables become false (even the simple case $x = x$), leading to a non-reflexive equivalence predicate. We follow the same idea, that is the membership predicate (which is our only predicate) is falsified over undefined variables. However, since equality is derived as $\epsilon_1 =_\tau \epsilon_2 \equiv \forall_\tau Y.\ (\epsilon_1 \in_\tau Y \leftrightarrow \epsilon_2 \in_\tau Y)$, we have that our equality predicate is reflexive.

Another similar proposal can be found in (DKR04), concerned with the approximation of special kinds of graphs and the verification of a similar logic to analyze pointer structures on the heap.

Finally, *evolution logic* (YRSW06), a first-order extension of LTL, is another interesting logic to reason about the dynamics featured in object-oriented programming languages. The model checking approach focuses on abstract interpretation.

Dynamic logic. Another interesting trend of research centres around *dynamic logic* (HKT00), a modal logic for which both propositional and first-order extensions exist.

Recent works consider the study of propositional fragments of dynamic logics with quantifiers (e.g. (Lei08)) and the application to graph transformations systems (BEH10). In particular, aiming to reason about data-structures defined by means of pointers, and processes manipulating them, the work in (BEH10) tries to fill the lack of appropriate termgraph rewriting proof methods, in order to take full advantage of the expressivity of termgraphs. A termgraph is a ground term with possibly cycles and sharing of sub-terms, able to model data structures with pointers like circular lists and doubly-linked lists. Termgraph rewrite rules can hence intuitively model manipulations of such data structures.

The authors propose an extension of dynamic logic tailored to fit termgraph rewriting, whose modal formulae are capable to specify termgraphs and rewrite rules' matching and application. The logic is shown to be undecidable in general, but a few decidable fragments are presented.

Spatio-temporal logics. Spatio-temporal logics form another track of formalisms for describing the evolution of processes and data structures.

Early works aimed at reasoning about networks of processes (e.g. the *multiprocess network logic* of (RS85)), and were based on extensions of classical linear- and branching-time logics with first-order quantifiers. In these works, the set of processes was considered to be fixed (i.e. no dynamic creation or deletion was considered) so that the elimination of quantifiers was possible.

In the last years spatial logics evolved, and have been mostly defined for algebraically presented systems. We cite among others spatial logics for process calculi like the π-calculus (Cai04; CC03; CC02; LV10) and mobile ambients (CG00), for protocol specification languages like Promela (Llu07), for rewrite theories (Mes08; BM10), for graph-based computational models like bigraphs (CMS07), and for data structures like graphs (CGG02; HLT06; FL06), heaps (Rey02; BDL09; BDL08), and trees (CGG03; CG04).

The common idea in such approaches is to mix temporal modalities with spatial operators that represent the dual of the operators of the algebra, like parallel (de)composition of processes or graphs, and various forms of (name) quantification. Renaming and merging of elements is typically restricted to some special cases like α-renaming and name extrusion.

To conclude this section, in our current work we developed a general framework based on the notion of counterpart relations, counterpart models and counterpart model approximations. We instantiated the approach with a simple μ-calculus syntax, and an intuitive and streamlined semantics, leaving for future works concerns about efficiency. One of our future goal is hence that of defining ad-hoc and less expressive logics in the same line of the one presented in (Ren03), and encodable in our μ-calculus, so that we will be able to lift all our current proposals and results to them.

9.2 Techniques and tools for the verification of visual specification formalisms

In this section we discuss some techniques and tools to make more effective the verification of visual specification formalisms, focusing on graph transformations.

Graph transformation systems. By resorting to graph transformation as system specification mechanism it is possible to model systems with infinite state-space, high degree of dynamism, dynamic creation and deletion of components, mobility and variable topology. Concrete examples are pointer structures on the heap, distributed, concurrent and mobile systems, and network protocols with varying number of participants. Moreover, graph transformation is nowadays widely used in the context of model transformation, where the model of a system is *manipulated* in order, for example, to reflect updates in the actual system, or in its specifications.

Attracted by these features, various authors proposed to verify software systems using graph transformation. The idea is that of modelling a system by a so-called graph transformation system (GTS), and then verify it with suitable ad-hoc logics, techniques and tools.

This is a challenging and active research area because, as it usually happens, a high level of expressiveness makes verification problematic.

Several approaches to graph rewriting exist, among which we find the algebraic one, divided in several sub-approaches, with double- and single- pushout (CMR+97; EHK+97) being the most common ones. In our prototypal model checker presented in Chapter 6 we implement an SPO-like formalism built on top of Maude's term rewriting by enriching the rules with the counterpart relation (similar to the trace morphism). An example of such rules is provided in Listing 6.3. However, our results are independent from the particular approach used, since the results apply to models, independently from how they have been generated.

In order to model a system as a graph transformation system it is necessary to define a graph representation of its initial state, and a set of graph rewrite rules specifying its behaviour in a declarative way. Applying a rewrite rule we create a new graph out of an original one, modeling the state obtained after the evolution step defined by the rule.

The state-space of a graph transformation system is computed applying exhaustively its rules starting from the initial state, obtaining *graph transition systems* (BCKL07). Those models are transition systems whose states are labelled with graphs, and transitions with partial morphisms between the graphs of the source and target state. As previously discussed, our counterpart models can be seen as a direct generalization of graph transition systems.

For a given initial state and a given set of rewrite rules, the generated graph transition system can have a finite or infinite state-space. Different approaches and techniques have been proposed to obtain efficient GTS verification, depending on the finiteness of the state-space of the systems.

Standard model checking. For finite-state systems it has been first proposed to encode graph transformation systems into the input language of

existing, optimized model checkers (BRRS08; DFRdS03; Var02), with the advantage of using their power. Unfortunately existing model checkers usually do not handle efficiently dynamic allocation and deallocation of system components, because often their logics are propositional and their algorithms are tuned for finite-state systems, while dynamism can easily lead to infinite state-spaces. Some partial solutions exist, like pre-computing the maximal number of nodes in all reachable graphs, and then reusing node names. Otherwise infinitely many nodes might be generated, obtaining infinite state-spaces. This technique is applicable only for systems with bounded resource allocation.

Another solution to model check graph transformation systems consists in applying reduction techniques like partial-order reduction, or symmetry reductions, decreasing the size of the state-space. The next paragraphs discuss some of the approaches falling in this category.

Unfolding-based approach. In graph transformation, the state-space explosion problem is mainly caused by the high level of concurrency and dynamism characterizing the systems modelled with GTSs. Intuitively, every different combination in the interleaving between concurrent events generate a different state. Partial-order reduction (Pel98) is a technique aiming at decreasing the degree of concurrency, thus reducing the state-space of concurrent systems. The idea is to not consider all possible interleavings of concurrent events, but to (partially) order them, and to consider only one of them (i.e. the minimal one).

The already mentioned approach of (BCKL07) and (BKK03) actually correspond to partial-order reduction, and aims at building a verification setting where graph transformation systems are abstracted into Petri graphs, an extension of Petri nets with additional graph structure.

The property specification language is a logic that mixes the modal μ-calculus with Monadic Second-Order logic for graphs (Cou97). The syntax of our logic is reminiscent of the one proposed in this approach.

Systems are specified as graph transformation systems, and the semantical domain of the logic is given in terms of graph transition systems. Actually, strong constraints are imposed on the class of admissible models.

In fact, the admissible graph transition systems are not allowed to introduce merging or renaming of graph items, and moreover, the semantics is defined over the *unravelling* of a graph transition system, that is a tree that represents the unfolded state-space and that guarantees some additional properties such as no-reuse of item names.

Intuitively, the unfolding of a graph transition system is an acyclic branching structure representing the possible computations of the system beginning from the initial graph, where cycles are expanded, then leading to infinite trees.

The tool support for the approach of (BCKL07) is under development. As far as we know, the tool AUGUR2 (KK08; aug) is limited to the propositional fragment of the logic only: quantifiers are allowed but cannot be interleaved with modal or fixpoint operators, resulting hence in state propositions.

The key technique of this approach is the *approximated unfolding technique*, described in (BCK01). Essentially the idea is that, given a graph transformation system, a Petri graph is obtained, that is a Petri net with additional graph structure. Then, it is shown that the graph transition system generated from the Petri graph is an over-approximation of the one obtained from the original graph transformation system. Intuitively, every graph reachable from the original model can be mapped homomorphically to a corresponding one in the approximated unfolding, but not necessarily vice-versa. Therefore, if a property over graphs is reflected by graph morphisms, then if it holds on the approximated unfolding it also holds in the original model.

Our framework for the approximation of counterpart models took inspiration from the work of (BCKL07), and generalized it. As a result, our concepts of over-approximation coincide. Hence, we could easily lift this technique to our framework.

In (BCK04), the authors of (BCKL07) show that in the case of finite-state graph transformation systems, it is possible to compute a *finite prefix* of its unfolding (i.e. the unfolding of its graph transition system), which is enough to represent all reachable graphs. The obtained approximated model is actually an over-approximation of the original one. The idea

behind this technique is based on cut-off events. Intuitively, an event is a cut-off if there is another event with a *smaller history* that generates an isomorphic graph. Then, the unfolding can be pruned after cut-offs.

Using this technique, unfoldings can be much smaller than the state-space of the system, especially if the system has a high degree of concurrency. The approach is inspired by early works on unfolding and complete prefix techniques for Petri Nets (BCM$^+$92; ERV96).

Symmetry reduction of graph transition systems. In (Ren03) it is proposed another interesting graph-based approach. Here formulae of a second-order linear-time logic are evaluated against (executions paths and assignments defined over) a kind of graph transition systems where, differently from the graph algebra we give in Example 2.1, states are labelled directed graphs. Our states are parametric with respect to the signature Σ, hence we can obtain labelled directed graphs by simply adding the sort "τ_L" (for labels), and the operation "$label : \tau_E \to \tau_L$" to the graph algebra presented in Example 2.1.

In order to navigate through the graph structure of a state, the logic provides regular path expressions (evaluated in sequences of labels of connected edges) and set expressions (evaluated in sets of nodes). Examples of set expressions are first- and second-order node variables. The logic also provides a set expression to evaluate the set of nodes reachable from a set of source nodes (defined by a set expression) through a set of paths (defined by a path expression).

As for our proposal, the logic presented in (Ren03) has (only) the membership predicate, used to establish if (the evaluation of) a first-order variable belongs to (the evaluation of) a set expression. Moreover, the logic is decidable for finite-state models. Finally, the logic has first- and second-order node quantifiers, the standard boolean connectives, and LTL's temporal operators.

There are hence quite a few similarities with our approach, even if we treat also edges (and elements of any other sort possibly defined in Σ) as first-class entities, allowing to quantify and evaluate predicates over them. Moreover we do not restrict just to the case of graphs, but allow for

different structures as states (depending on Σ).

In the line of research of (Ren03) falls the development of the tool GROOVE (GdMR+12; gro), a state-space generator and model checker for graph transformation systems. Among the features supported by the tool, one that surely inspired our work is reduction up-to-isomorphism. The technique, in turn inspired by research on HD-Automata (MP05b), allows to generate state-spaces up-to-isomorphism. Namely, during the generation of the state-space, before storing a new state it is first checked if an isomorphic one has been already considered (Ren06b). As known, isomorphism check is expensive, hence heuristics based on hashing-like graph certificates have been introduced to reduce the number of comparisons (Ren06b). Interestingly, in the formalism used to model systems in GROOVE (i.e. a particular flavour of graph transformation), we have that a system reduced up-to-isomorphism is bisimilar to the original one, meaning that we can use the reduced model to check properties of standard logics like CTL or LTL on the original one. Clearly, this gives rise to an automatic state-space reduction technique which can be applied to any system modelled in GROOVE.

Our approach to state-space reduction is instead based on `c`-reductions, and it is realized via state canonizers that map states in the (not necessarily unique) representative of their equivalence class, given an equivalence which is also a bisimulation. In Chapter 7 we presented three canonizers to obtain full symmetry reduction (`freeTransps`) and rotational symmetry reduction (`rotations`) for two particular systems, and a technique that *compacts* the names of the states (`compactNames`).

In particular, considering the case in which we use graphs to model the states of a system (i.e. we fix as Σ the signature of graphs presented in Example 2.1), in Chapter 7 we have shown how it is possible to combine the canonizers `freeTranps` and `compactNames` to obtain a canonizer for the equivalence class of states with isomorphic graphs. Then, isomorphic states are mapped in a minimal isomorphic one, obtaining thus isomorphism reduction. However, in order to exploit the reduced models for verification we need that the equivalence class induced by the combined canonizer is also a bisimulation. Namely that the dynamics of the

system are preserved by isomorphism reduction. Intuitively, this surely happens when the rules specifying the dynamics of the system (e.g. the ones in Listing 6.3 for the leader election system) does not depend on the identities of the elements of the states, but only on the interrelations between them (i.e. only on the structure of the graphs labelling the states rather than on the names of their edges and nodes). This is exactly what happens in GROOVE.

To sum up, since the composed canonizer has been defined once and for all, if we restrict to the modelling language of GROOVE, then we have a completely automatic technique for isomorphism reduction. The main difference is that rather than checking if an isomorphic graph has been already considered, we apply state canonizers to each newly created state, and then we check if the *canonized* state has been already considered. Noteworthy, isomorphism reduction is only one of the reductions that we can obtain with c-reductions, as the user can define its own canonizers to exploit the regularities of each modeled system.

Approaches based on abstract graph transformation and shape analysis. In order to deal with infinite state-space systems, it has been proposed to import to graph transformation techniques based on abstract interpretation (CC77). Namely, it has been introduced the concept of *abstract graph transformation* (BRKB07; RD05), where, intuitively, given some equivalence relation, graphs are quotiented into abstract graphs of bounded, finite size.

Then, from abstract graph transformation, and from shape analysis (SRW02), a technique where groups of *similar* states are abstracted in shapes, derives another interesting state-space reduction technique implemented in GROOVE, i.e. *neighbourhood abstraction* (BBKR08; RZ10).

The idea is to abstract infinitely many graphs by a finite number of *shapes*, where a shape is a graph enriched with a *node (edge) multiplicity function* indicating, for each node (edge) of the shape, how many nodes (edges) it summarises. In order to do so, edges and nodes are collapsed modulo *neighbourhood similarity*. For example, edges with the same source node, and ending into nodes in the same group (or, respectively, edges

with the same target node, and starting in nodes in the same group) cannot be distinguished. Only the number of such edges is indicated with the help of the edge multiplicity functions of the shape.

The obtained model is an over-approximation of the original one. By increasing the *radius* of the considered neighbourhood (i.e. edges are collapsed if all their neighbours distant at most *radious* are equal), it is possible to obtain more precise approximations at the cost of bigger state-spaces. Still, in (BRKB07) it is shown that shapes can themselves be further abstracted.

Other than neighbourhood abstraction, in the literature there exist other contributions for abstract graph transformation inspired by the shape analysis of (SRW02). A recent one is proposed in (SWW11). Since the proposal is inspired by shape analysis techniques, abstract graphs are called shape graphs (as for the case of neighbourhood abstraction). As for the original proposal of (SRW02), shape graphs are enriched with *shape constraints*, namely first-order predicate formulae which provide further informations about the concrete graphs represented by a shape.

The main difference with respect to (SRW02) lies in the transformation handling. In fact the authors of (SWW11) defined an automatic sound and complete transformation system (execution steps) on abstract graphs. This means that they capture exactly the transformations on the concrete graph level. In this way, as discussed by the authors, an effectively computable *best transformer* for abstract graphs is obtained, which can be employed in verification techniques for GTSs based on abstraction.

We are confident that our framework for counterpart model approximation may provide interesting insights for techniques based on abstract graph transformation like the ones in (BBKR08; SWW11). In fact, shape graphs can be easily seen as graph algebras extended with multiplicity operations. Moreover, the *abstraction morphisms* that are used to coalesce nodes and edges of concrete states in abstract states according to some notion of similarity are actually surjective graph morphisms, similar to the morphisms of our notion of counterpart model simulation.

The logic adopted in (BBKR08) appears less expressive than ours (as well as of the one used in (BCKL07)), as it does not have temporal

modalities, but it offers the advantage that all formulae are strongly preserved by the approximation.

More recently, the authors of (RZ10), proposers of neighbourhood abstraction, developed a novel abstraction technique based on abstract graph transformation, named *pattern-based graph abstraction* (RZ12).

The novelty introduced with respect to neighbourhood abstraction is that now the concept of shape is not used to represent abstract graphs, but *abstract pattern graphs*. A pattern is a simple graph describing structures of interest that should be preserved by the abstraction. Then, pattern graphs are just layered graphs obtained by hierarchical composition of patterns.

Similarly to how graphs were abstracted to shape graphs, here pattern graphs are abstracted to pattern shapes by collapsing equivalent patterns. The difference stands in the fact that neighbourhood abstraction collapses only simple nodes and edges.

This flexibility in specifying which structures should be collapsed actually represents the main novelty and advantage of the pattern-based abstraction with respect to the neighbourhood one. As previously mentioned, it is possible to tune the precision of neighbourhood abstraction by modifying its radius parameter (the maximal distance up to which neighbours are considered in neighbourhood similarity). However, this radius is then applied indistinctly to all the nodes and edges. The pattern-based method allows instead for a finer tuning of the precision of the approximation, because patterns provide greater informations rather than neighbourhood, and more importantly, several distinct patterns can be used to compute a similarity. In particular, the authors claim that a neighbourhood abstraction with a given radius can be simulated by a corresponding pattern abstraction, from which it follows that the latter may generalize the former.

Backward anlysis. Another approach to the analysis of infinite-state graph transformation systems relies on a technique named *backward analysis* (ABC$^+$08; JK08; SWJ08). The idea is that of doing a backward search starting from error graphs, taking note of the encountered graphs by symbolically representing them. The result is a symbolically representation of

all graphs reachable by this backward search. Then it is sufficient to check whether the start graph is represented. If the start graph is represented, then it means that the start graph can reach an error graph.

Summary. As far as we know, existing graph transformation tools are not yet equipped with model checking capabilities for quantified modal μ-calculus, or more generally for quantified temporal logics. Amongst them, the already cited GROOVE and AUGUR2 seem the most promising ones, since their authors have already produced very interesting contributions to the theoretical foundations of model checking systems with dynamic structure using quantified temporal logics.

Our (theoretical) contribution represents a further step in this direction. Moreover, we have discussed how some of the techniques proposed in the research lines regarding the two tools are related to our proposals. Hence we are confident that our proposal may also help in integrating the results belonging to the research lines.

It may be worthwhile to state once more that the tool presented in Part II has to be intended as a proof-of-concept tool, mainly aimed at assessing the feasibility of our approach. It should be intended as preparing the ground for an efficient framework for verifying interesting properties of systems with dynamically evolving structure, that is where system components and their interrelations may vary over time (e.g. via (de)allocation, merging and renaming of components, and creation and breaking of relationships). The current implementation is tailored to our needs, leaving for future works concerns about efficiency and usability. Needless to say, from a comparative analysis it results that our tool is outperformed by GROOVE. It could anyhow be interesting to integrate some of our proposals in such a mature tool.

Chapter 10

Conclusions

In this thesis we presented a framework for the analysis of systems with dynamically evolving components and resources: they may join or leave the system, or even get combined during its evolution. We refer to this kind of systems as "systems with dynamic structure".

We proposed a novel approach to the semantics of quantified μ-calculi, that is languages combining quantifiers with the fix-point and modal operators of temporal logics, inspired by Counterpart Theory of David Lewis (Lew68). With respect to other approaches, including those commented in Chapter 9, our proposal allows for a mathematically elegant definition of the semantic universe by means of counterpart models, i.e. labeled transition systems where states are algebras representing their structure (their components and the relations between them), and transitions are labeled with counterpart relations (partial morphisms) between states, which provide informations about the evolution of system components. The idea of associating sets of pairs of states and variable assignments to (open) formulae, instead of just states, allows for a straightforward interpretation of fixed points and for their smooth integration with the evaluation of quantifiers, which often asked for a restriction of the class of admissible models. Moreover, our logic allows to reason about allocation, deallocation, renaming and merging of components, and it is hence well suited to reason about systems with dynamic structure.

Unfortunately, systems with dynamic structure often have huge or infinite state-spaces even for very simple cases. Verification can hence become intractable, calling for the development of state-space reduction and approximation techniques that may ease the verification at the acceptable cost of losing in preciseness and completeness.

In this direction, we proposed *c-reductions*, a state-space reduction technique for counterpart models capturing symmetry reduction, name reusing and name abstraction. The technique is based on state canonizers, namely functions mapping states in the (possibly not unique) representative of their equivalence class, given an equivalence which is also a bisimulation. Then, a model is bisimilar, i.e. behaviourally equivalent, to its c-reduction.

However, in order to exploit state-space reduction (or approximation) techniques, it is first necessary to understand how the semantics of the logic is related to reduced or approximated models. For this reason we proposed a general notion of counterpart model approximation based on standard behavioural pre-orders for transition systems, and in particular to similarities, which we extended to counterpart models. We considered both *under-approximations*, namely models that express *less behaviours* than the original one, and *over-approximations*, namely models that express *more behaviours* than the original one. Intuitively, an under-approximation $\underline{M}$ is simulated by the original model M (i.e. M is similar to $\underline{M}$), while an over-approximation $\overline{M}$ simulates M (i.e. $\overline{M}$ us similar to M). In the particular case in which a model is bisimilar to M, as it is the case of the ones obtained with c-reductions, then it is both an under- and an over-approximation, and it represents exactly the same behaviours of the original one.

Interestingly, our type system is complete for bisimilar models, in which case it assigns the type *strongly preserved* to every formula, that is preserved and reflected.

Our approach to model approximations can be seen as an evolution of the verification technique for graph transformation systems based on temporal graph logics and unfoldings (BKK03; BCKL07), which is extended on the kind of models and of simulations under analysis. We are

also confident that our proposal may provide interesting insights for other approximation techniques, such as *neighbourhood abstraction* (BBKR08), where states are *shapes* (i.e. graph algebras extended with an operation for abstraction purposes), and suitable abstraction morphisms (i.e. surjective graph morphisms, similar to the morphisms of our simulations) coalesce nodes and edges of concrete states according to their neighbourhood similarity. The logic adopted is less expressive than ours (as well as of the one used in (BCKL07)), but it offers the advantage that all formulae are strongly preserved by the approximation.

Finally, we instantiated our approach implementing a prototypal explicit-state model checker for our logic, with support for $\subset$-reductions. The tool has been implemented to prove the feasibility of our approach, and to prepare the ground to build an efficient tool framework for it.

Our general framework for the analysis of systems with dynamically evolving structure opens other possible future interesting lines of research.

As a start, we would like to investigate if the correspondence results between quantified μ-calculi and Petri nets logics proposed in (BCKL07) could be lifted to our framework, and its richer family of counterpart relations. We would also like to better understand the relationship with spatial logics, along the lines of (GL07), possibly adopting a family of labeled counterpart relations, and the richer modal operators $\Diamond_{\langle p,Y\rangle}$, basically stating that the transition between worlds is caused by a specific action p (e.g. a rule in rewriting-based settings), that may create a chosen set Y of new elements.

Then, it would be interesting to define less expressive logics encodable in the one presented in this thesis, allowing to define effective model checking algorithms, and at the same time reuse all the results presented in this thesis.

Finally, we would like to further investigate the enrichment of our approximated semantics (Section 4.2), in order to deal with more untyped formulae. An interesting question in this regard is whether we can use both an under- and an over-approximation *simultaneously*, by translating assignment pairs back and forth via the composition of the corresponding abstraction and concretization functions.

References

[ABC+08] Parosh Aziz Abdulla, Ahmed Bouajjani, Jonathan Cederberg, Frédéric Haziza, and Ahmed Rezine. Monotonic abstraction for programs with dynamic memory heaps. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV'08)*, LNCS, pages 341–354. Springer, 2008. 151

[aug] AUGUR2, `www.ti.inf.uni-due.de/research/augur/`. 146

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A modal-logic based graph abstraction. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *LNCS*, pages 321–335. Springer, 2008. 149, 150, 155

[BCK01] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In Kim Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001. 146

[BCK04] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: An unfolding-based approach. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 83–98. Springer, 2004. 140, 146

[BCKL07] Paolo Baldan, Andrea Corradini, Barbara König, and Alberto Lluch Lafuente. A temporal graph logic for verification of graph transformation systems. In José Luiz Fiadeiro and Pierre-Yves Schobbens, editors, *WADT*, volume 4409 of *LNCS*, pages 1–20. Springer, 2007. 5, 6, 8, 12, 18, 23, 37, 140, 144, 145, 146, 150, 154, 155

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98:142–170, June 1992. 147

[BDH01] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. A heuristic for symmetry reductions with scalarsets. In *International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME'01)*. Springer, 2001. 51

[BDH02] Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):92–106, 2002. 51

[BDL08] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. On the almighty wand. In Michael Kaminski and Simone Martini, editors, *Proceedings of the 17th Annual Conference of the EACSL on Computer Science Logic (CSL'08)*, volume 5213 of *LNCS*, pages 323–338. Springer, 2008. 142

[BDL09] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. Reasoning about sequences of memory states. *Annals of Pure and Applied Logic*, 161(3):305–323, 2009. 142

[BEH10] Philippe Balbiani, Rachid Echahed, and Andreas Herzig. A dynamic logic for termgraph rewriting. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of the 5th International Conference on Graph Transformations (ICGT'10)*, volume 6372 of *LNCS*, pages 59–74. Springer, 2010. 142

[Bel06] Francesco Belardinelli. *Quantified Modal Logic and the Ontology of Physical Objects*. PhD thesis, Scuola Normale Superiore of Pisa, 2006. 5

[BKK03] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In Radhia Cousot, editor, *SAS*, volume 2694 of *LNCS*, pages 255–272. Springer, 2003. 55, 56, 57, 145, 154

[BM10] Kyungmin Bae and José Meseguer. The linear temporal logic of rewriting maude model checker. In Peter Csaba Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA'10)*, volume 6381 of *LNCS*, pages 208–225. Springer, 2010. 142

[BRKB07] I.B. Boneva, A. Rensink, M.E. Kurban, and J. Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, University of Twente, 2007. 140, 149, 150

[BRRS08] Luciano Baresi, Vahid Rafe, Adel Torkaman Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 213(1):3–21, 2008. 145

[Cai04] L. Caires. Behavioral and spatial observations in a logic for the π-calculus. In I. Walukiewicz, editor, *Proceedings of the 7th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*, pages 72–87. Springer, 2004. 140, 142

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977. 149

[CC02] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part ii). In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2002. 142

[CC03] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194–235, 2003. 142

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All about Maude*, volume 4350 of *LNCS*. Springer, 2007. 13, 15, 49, 72, 74, 77, 80, 82

[CE12] B. Courcelle and J. Engelfriet. *Graph structure and monadic second-order logic, a language theoretic approach*. Cambridge University Press, 2012. 4

[CG00] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 365–377, 2000. 142

[CG04] Luca Cardelli and Giorgio Ghelli. TQL: A query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004. 142

[CGG02] L. Cardelli, Ph. Gardner, and G. Ghelli. A spatial logic for querying graphs. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo,

editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'02)*, volume 2380 of *LNCS*, pages 597–610. Springer, 2002. 142

[CGG03] L. Cardelli, Ph. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In A. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, volume 2620 of *LNCS*, pages 216–232. Springer, 2003. 142

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. 1

[CMR+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In Rozenberg (Roz97), pages 163–246. 144

[CMS07] Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. Static bilog: a unifying language for spatial structures. *Fundamenta Informaticae*, 80(1-3):91–110, 2007. 142

[Cou89] Bruno Courcelle. The monadic second-order logic of graphs, ii: Infinite graphs of bounded width. *Mathematical Systems Theory*, 21(4):187–221, 1989. 4

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. 4

[Cou97] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 313–400. World Scientific, 1997. 4, 140, 145

[CPR06] Manuel Clavel, Miguel Palomino, and Adrián Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. 49

[CRe] C-Reducer, `http://sysma.lab.imtlucca.it/tools/c-reducer`. 49

[DFRdS03] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi dos Santos. Verification of distributed object-based systems. In Elie Najm, Uwe Nestmann, and Perdita Stevens, editors, *FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2003. 145

[DGG07] Anuj Dawar, Philippa Gardner, and Giorgio Ghelli. Expressiveness and complexity of graph logic. *Information and Computation*, 205(3):263–310, 2007. 4

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informaticae*, 1:115–138, 1971. 73, 106, 120

[DKR04] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? In K. Lodaya and M. Mahajan, editors, *Proceedings of the 32nd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004. 141

[DM06] Alastair F. Donaldson and Alice Miller. A computational group theoretic symmetry reduction package for the SPIN model checker. In *Algebraic Methodology and Software Technology*, pages 374–380, 2006. 50

[DM10] Francisco Durán and José Meseguer. A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In Peter Csaba Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA'10)*, volume 6381 of *LNCS*, pages 69–85. Springer, 2010. 49

[DRK02] Dino Distefano, Arend Rensink, and Joost-Pieter Katoen. Model checking birth and death. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *Procedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS'02)*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer, 2002. 3, 6, 7, 34, 36, 73, 106, 120, 122, 131, 141

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. 3, 140

[EEKR12] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*. Springer, 2012. 162, 165

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 3, 140

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach*, pages 247–312. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 144

[EMS03] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *SPIN*, pages 230–234, 2003. 49

[ERV96] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106. Springer, 1996. 147

[FL06] Gianluigi Ferrari and Alberto Lluch Lafuente. A logic for graphs with QoS. In Fabio Gadducci and Maurice ter Beek, editors, *Proceedings of the 1st International Workshop on Views on Designing Complex Architectures (VODCA'04)*, volume 142 of *ENTCS*, pages 143–160. Elsevier, 2006. 142

[FT03] E. Franconi and D. Toman. Fixpoint extensions of temporal description logics. In Diego Calvanese, Giuseppe De Giacomo, and Enrico Franconi, editors, *Proceedings of the 16th International Workshop on Description Logics (DL'03)*, volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003. 4, 139

[GdMR+12] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *Software Tools for Technology Transfer*, 14(1):15–40, 2012. 148

[GI89] Dan Gusfield and Robert W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989. 93

[GL07] Fabio Gadducci and Alberto Lluch Lafuente. Graphical encoding of a spatial logic for the π-calculus. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Proceedings of the 2nd International Conference on Algebra and Coalgebra in Computer Science (CALCO'07)*, volume 4624 of *LNCS*, pages 209–225. Springer, 2007. 140, 155

[GLV10] Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Counterpart semantics for a second-order μ-calculus. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of the 5th International Conference on Graph Transformation*

(ICGT'10), volume 6372 of *LNCS*, pages 282–297. Springer, 2010. xiii, 8, 9, 13, 35, 83, 92, 117

[GLV12a] Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Counterpart semantics for a second-order -calculus. *Fundam. Inform.*, 118(1-2):177–205, 2012. xiii, 9, 13, 83

[GLV12b] Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Exploiting over- and under-approximations for infinite-state counterpart models. In Ehrig et al. (EEKR12), pages 51–65. xiii, 10, 12

[gro] GROOVE, `http://groove.cs.utwente.nl/`. 148

[Haz04] Allen Hazen. Counterpart-theoretic semantics for modal logic. *The Journal of Philosophy*, 76(6):319–338, 2004. 5

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, 2000. 142

[HLT06] Dan Hirsch, Alberto Lluch Lafuente, and Emilio Tuosto. A logic for application level QoS. In *Proceedings of the 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, volume 153(2) of *ENTCS*, pages 135–159. Elsevier, 2006. 142

[HWZ01] I. M. Hodkinson, F. Wolter, and M. Zakharyaschev. Monodic fragments of first-order temporal logics: 2000-2001 a.d. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01)*, volume 2250 of *LNCS*, pages 1–23. Springer, 2001. 4, 139

[Inv] The Maude Invariant Analyzer Tool (InvA), `camilorocha.info/software/inva`. 49

[ITP] Maude Interactive Theorem Prover, `maude.cs.uiuc.edu/tools/itp/`. 49

[JK08] Salil Joshi and Barbara König. Applying the graph minor theorem to the verification of graph transformation systems. In *Proceedings of the 20th international conference on Computer Aided Verification (CAV'08)*, pages 214–226. Springer, 2008. 151

[KK08] Barbara König and Vitali Kozioura. Augur 2 - a new version of a tool for the analysis of graph transformation systems. *Electr. Notes Theor. Comput. Sci.*, 211:201–210, 2008. 146

[Lei08] Daniel Leivant. Propositional dynamic logic with program quantifiers. In Andrej Brauer and Michael Mislove, editors, *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV)*, volume 218 of *ENTCS*, pages 231–240. Elsevier, 2008. 142

[Lew68] David Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, 65:113–126, 1968. 7, 18, 22, 138, 153

[Llu07] Alberto Lluch Lafuente. Towards model checking spatial properties with SPIN. In Dragan Bosnacki and Stefan Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 223–242. Springer, 2007. 142

[LMV12] Alberto Lluch Lafuente, José Meseguer, and Andrea Vandin. State space c-reductions of concurrent systems in rewriting logic. In Toshiaki Aoki and Kenji Taguchi, editors, *ICFEM*, volume 7635 of *Lecture Notes in Computer Science*, pages 430–446. Springer, 2012. xiii, 10, 11, 45, 49, 52, 92

[LV10] Étienne Lozes and Jules Villard. A spatial equational logic for the applied π-calculus. *Distributed Computing*, 23(1):61–83, 2010. 142

[LV11] Alberto Lluch Lafuente and Andrea Vandin. Towards a Maude tool for model checking temporal graph properties. In Fabio Gadducci and Leonardo Mariani, editors, *GT-VMT*, volume 42 of *ECEASST*. EAAST, 2011. xiii, 13, 83, 93

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996. 51

[Mes08] José Meseguer. The temporal logic of rewriting: A gentle introduction. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*, pages 354–382. Springer, 2008. 142

[Mes12] José Meseguer. Twenty years of rewriting logic. *J. Log. Algebr. Program.*, 81(7-8):721–781, 2012. 15, 49, 74, 77

[MP05a] Ugo Montanari and Marco Pistore. History-dependent automata: An introduction. In Marco Bernardo and Alessandro Bogliolo, editors, *SFM*, volume 3465 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005. 6

[MP05b] Ugo Montanari and Marco Pistore. Structured coalgebras and minimal hd-automata for the π-calculus. *Theoretical Computer Science*, 340(3):539–576, 2005. 141, 148

[Pel98] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998. 145

[RD05] Arend Rensink and Dino Distefano. Abstract graph transformation. In *Proceedings of the 3rd International Workshop on Software Verification and Validation (SVV'05)*. ENTCS, 2005. 149

[Ren03] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *AVOCS*, volume 2003-2 of *DSSE-TR*. University of Southampton, 2003. 140, 143, 147, 148

[Ren06a] A. Rensink. Model checking quantified computation tree logic. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 110–125. Springer, 2006. 6, 140

[Ren06b] Arend Rensink. Isomorphism checking in GROOVE. In Albert Zündorf and Daniel Varró, editors, *GraBaTs*, volume 1 of *ECEASST*. EAAST, 2006. 57, 148

[Rey02] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74. IEEE Computer Society, 2002. 142

[Rod09] Dilia E. Rodríguez. Combining techniques to reduce state space and prove strong properties. In *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA 2008)*, volume 238(3) of *ENTCS*, pages 267 – 280, 2009. 49

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997. 3, 140, 159

[RS85] John Reif and A. P. Sistla. A multiprocess network logic with temporal and spatial modalities. *International Journal of Computer and System Sciences*, 30(1):41–53, 1985. 142

[RZ10] Arend Rensink and Eduardo Zambon. Neighbourhood abstraction in GROOVE. In Juan de Lara and Daniel Varró, editors, *GraBaTs*, volume 32 of *ECEASST*. EAAST, 2010. 149, 151

[RZ12] Arend Rensink and Eduardo Zambon. Pattern-based graph abstraction. In Ehrig et al. (EEKR12), pages 66–80. 151

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. 149, 150

[SWJ08] Mayank Saksena, Oskar Wibling, and Bengt Jonsson. Graph grammar modeling and verification of ad hoc routing protocols. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'08)*, LNCS, pages 18–32. Springer, 2008. 151

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and complete abstract graph transformation. In Adenilso da Silva Simão and Carroll Morgan, editors, *SBMF*, volume 7021 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2011. 150

[Var02] Dániel Varró. Towards symbolic analysis of visual modelling languages. In Paolo Bottoni and Mark Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 (3) of *ENTCS*, page 57–70, Barcelona, Spain, October 11-12 2002. Elsevier, Elsevier. 145

[WD10] Thomas Wahl and Alastair F. Donaldson. Replication and abstraction: Symmetry in automated formal verification. *Symmetry*, 2(2):799–847, 2010. 11, 49, 100

[YRSW06] Eran Yahav, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. *Logic Journal of the IGPL*, 14(5):755–783, 2006. 7, 34, 37, 141