

DINO DISTEFANO

On model checking the dynamics
of object-based software

a foundational approach

Distefano, Dino

On model checking the dynamics of object-based software: a foundational approach / Dino Distefano - Ph.D. thesis, University of Twente, 2003
ISBN 90-365-1975-6



IPA Dissertation Series No: 2003-09

CTIT Ph.D. thesis series No: 03-58

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and within the context of the Centre for Telematics and Information Technology (CTIT).

Publisher: Twente University Press, P.O. Box 217, 7500 AE Enschede, the Netherlands, www.tup.utwente.nl

Typeset: $\text{\LaTeX} 2_{\epsilon}$

Print: Ocè Facility Services, Enschede

Cover design by Daniela Distefano

Copyright © Dino Distefano, Enschede, 2003

No part of this work may be reproduced by print, photocopy or any other means without the permission in writing from the publisher.

ISBN 90-365-1975-6

ISSN 1381-3617

ON MODEL CHECKING THE DYNAMICS
OF OBJECT-BASED SOFTWARE:
A FOUNDATIONAL APPROACH

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 7 november 2003 te 16.45 uur

door

Dino Salvo Distefano

geboren op 20 juli 1973
te Catania, Italië

Dit proefschrift is goedgekeurd door:

prof. dr. H. Brinksma (promotor)

dr. ir. J.-P. Katoen (assistent-promotor)

dr. ir. A. Rensink (assistent-promotor)

Thanks to...

Thanks to Ed Brinksma, my promotor, for giving me the possibility to start as Ph.D. student in his FMT group four years ago. I have really enjoyed the sparkling scientific environment which Ed has created in the group. Although he is usually extremely busy, he has always been ready to find time to discuss with me all kinds of issues.

Thanks to Joost-Pieter Katoen and Arend Rensink, my daily supervisors. Working on a Ph.D. thesis is often a lonely and frustrating exercise. Mine would not have been an exception if Arend and Joost-Pieter were not there. As a matter of fact, the existence of this thesis is mostly due to the continuous care that I have received from both of them during these years. I will never forget our short meetings that always turned into long hours brainstorming at the white-board. After each of them I was busy for weeks trying to work out new problems. It is fair to say that almost the entirety of the results contained in this thesis originated from those meetings. During the years, Joost-Pieter and Arend patiently cured my unfortunate, sloppy attitude. In many difficult occasions they helped me and — despite all the disasters I was able come up with — they continued to believe that I could reach the end. Also, they taught me how to translate those obscure ideas I had in my mind into science, and how to communicate them to others. If you are one of the (un)fortunate readers that will go beyond Chapter 2, and find it unreadable, try to imagine how it was before Arend and Joost-Pieter struggled years convincing me to make things easier. I consider myself fortunate to have had the opportunity to learn from them.

Thanks to Mehmet Aksit, John Hatcliff, Anneke Kleppe, John-Jules Meijer, and Mooly Sagiv for accepting the duty of being members of the promotion commission and read the dissertation.

Thanks to Holger Hermanns and Pedro D'Argenio my “paranymphs”. Unfortunately, they have not been directly involved in my research. However, I like to think that the long trip to the completion of this thesis started once upon a time (one night in February/March 1999) in a Mexican restaurant in Enschede. There Pedro and Holger invited me for dinner after the formal interview with

the senior members of the group. Although after some time I learned that this is the standard hiring procedure at the FMT group, that dinner was still special¹. That famous night, helping themselves with several Coronas², they convinced me that FMT was the right place for my doctoral studies. True. Fortunately, the fun shared together on the Corona-night did not turn out to be an isolated case. On the contrary, it was only the beginning. Since that night, they have been for me the epitome of true scientists. In this respect, and on several occasions Holger and Pedro have influenced me and contributed towards my development as a scientist.

Thanks to Theo Ruys who has been always ready to answer a wide range of questions; from science, to bureaucracy, to my thesis. Even the most silly questions. He was even able to make me feel unashamed of “mijn ABN nederlands”. Theo has provided me with some nice L^AT_EX styles that have improved the appearance of the thesis and translated the summary³.

Thanks to Joke Lammerink whose help has been useful for all those bureaucratic matters a computer nerd has to face in everyday life. Do you have any idea what usually happens when twenty or so computer scientists are abandoned in those few weeks the secretary is on holiday leave? It is always a catastrophe. To use the standard FMT example, consider those deadlock situations like: entire weeks spent *with coffee without sugar*; or *sugar without coffee*; or *with coffee and sugar but no coffee-filters*; etc. If this is not enough you can try to make some fancy permutations of the terms: *coffee, sugar, with, without*, — and also — *reisdeclaratie, werkgeversverklaring*.

Thanks to Ric Klaren, my office mate, who saved me in several occasions from my well-established inability to cope with computers, operating systems, `find dirwhereithastocomefrom | xargs ln -s`, shell scripts, installations, fancy tools, and so on.

Thanks to John Hatcliff and Matt Dwyer for having provided me with the possibility to spend three inspiring months at Kansas State University. There, I became aware and interested in abstraction, which broadened my view on this research field. Thanks to Radu Iosif and Georg Jung for interesting research discussions and for being great companions during my stay in Manhattan.

Thanks to Laura Stevens who kindly checked my English in Chapters 1 and 7 during a trip from Italy to the USA.

Thanks to Daniela Distefano for designing the cover of the book with some tricky computer-aided manipulations of one of my drawings.

Thanks to Monica Brivio who has been like a sister (yet another) during my Dutch sojourn. She is the first Italian I met in Enschede four years ago and for some strange reason she has always been around to listen to my complaints

¹I am pretty sure of it since I attended several such dinners later on playing the same role as Pedro and Holger.

²Apparently, they understood immediately one of my weak points.

³If you are Dutch and your name is not Theo, you do not want to read a *samenvatting in mijn nederlands*.

in our mother tongue. As a matter of fact, when it comes to complaints, I personally consider Italian a much more suitable language than English.

Thanks to Claudia who, although faraway, has provided constant support in several ways, particularly in this last period. I have really appreciated it.

Thanks to the macandrians⁴ who made me enjoy living in this small town. Many people still wonder why, at the time of writing these acknowledgements, I am still living among them (actually sometimes I wonder too). However, so far, I can hardly imagine my life in Enschede away from them. Among all, some experienced macandrians have been the closest to me. Thanks to Azita, Maria, Peter, Brian, Uwe, Katrin, Ruth, and Sander who have always been macandrians at the right time.

Thanks to Antonio and Marcello who, during my trips back to my home town, were there. Always.

Grazie ai miei genitori per la loro continua comprensione in tutti questi anni lontani. A loro desidero dedicare questo lavoro.

⁴*Macandrians* are a rare tribe populating the suburb of Enschede. They are so rare that even Google has some difficulties to find them in the Internet if not provided with some tricky keywords.

Summary

This dissertation is concerned with software verification, in particular automated techniques to assess the correct functioning of object-based programs. We focus on the dynamic aspects of these programs and consider model-checking based verification techniques. The major obstacle to the design of model-checking algorithms is the *infinite state-space explosion* caused by the dynamic constructs supported by object-based languages. On the one hand, unbounded *allocation* and *deallocation* (birth and death) of objects and threads give rise to unbounded state spaces. On the other hand, the capability of objects *to refer to* each other by pointers (references) yields a heap — where objects are allocated — with a dynamic topological structure that evolves in an unpredictable and intricate manner.

In order to tackle the aforementioned issues, in the first part of this thesis, we define a temporal logic — based on linear temporal logic (LTL) — that is aimed at specifying a wide range of properties of object-based systems. Subsequently, we define and study two main subsets of the logic. The first is a restriction to a core subset suitable to reason about allocation and deallocation of objects. The second subset allows, in addition, reasoning about dynamic pointers between objects. These temporal logics are interpreted on appropriate (Büchi) automata-based models which provide finite-state abstractions of infinite-state systems. These automata are employed for the definition of the operational semantics of programming languages and constitute the basis for our model-checking algorithms. For the case of allocation and deallocation we achieved a sound and complete algorithm. For the more involved case of dynamic references, we obtained a sound algorithm. However, the latter may occasionally return false negatives.

Moreover, in this dissertation, we provide examples demonstrating how finite automata can be automatically extracted from a program. Finally, we illustrate the usage of the developed theory and algorithms for the verification of security properties for Mobile Ambients.

*è una questione di qualità
o una formalità
non ricordo più bene una formalità
come decidere di radersi i capelli
di eliminare il caffè, le sigarette
di farla finita con qualcuno
o qualcosa, una formalità una formalità
o una questione di qualità*

[CCCP (former Italian punk band), 1985]

Contents

Thanks to...	i
Summary	v
1 Introduction	1
1.1 The challenges of model checking object-based software	1
1.2 Contributions	6
1.3 Outline	7
1.4 Related work: the jungle of software model checkers	8
2 Preliminaries	11
2.1 Model checking and temporal logics	11
2.1.1 Kripke structures and Büchi automata	12
2.1.2 Linear Temporal Logic	16
2.1.3 Computation Tree Logic	17
2.1.4 A few formal notions relating formulae and models . .	19
2.1.5 The automata theoretic approach to model checking .	20
2.1.6 The tableau approach to model checking	24
2.1.7 Automata theoretic versus tableau approach	26
2.2 History-Dependent Automata	27
2.3 Object-based systems	29
2.3.1 The Unified Modelling Language	31
2.3.2 Dynamic allocation and deallocation	32
2.3.3 Dynamic references	35
3 A logic for object-based systems	37
3.1 Introduction	37
3.2 Syntax of BOTL	38
3.2.1 Data types and values	38
3.2.2 Syntax of BOTL	40

3.3	Semantics of BOTL	44
3.3.1	BOTL operational models	44
3.3.2	Semantics of BOTL static expressions	48
3.3.3	Semantics of BOTL temporal formulae	49
3.4	Object Constraint Language	50
3.4.1	An informal and concise summary of OCL basic concepts	51
3.4.2	OCL syntax	53
3.4.3	Some OCL restrictions	54
3.5	Translating OCL into BOTL	56
3.5.1	Translation issues	56
3.5.2	Translating OCL expressions into BOTL	59
3.5.3	Translating OCL constraints into BOTL	60
3.5.4	How to employ BOTL for OCL tools	62
3.6	Related work	65
3.6.1	The Bandera Specification Language	65
3.6.2	Others	67
4	Dynamic Allocation and Deallocations	69
4.1	Introduction	69
4.2	Allocational temporal logic	71
4.2.1	Syntax	71
4.2.2	Semantics	72
4.2.3	Folded allocation sequences	74
4.2.4	Relating unfolded and folded allocation sequences . .	75
4.3	Automata for dynamic allocation and deallocation	79
4.3.1	Allocational Büchi Automata	79
4.3.2	High-level Allocational Büchi Automata	80
4.3.3	The duality between ABA and HABA	87
4.4	Programming allocation and deallocation	89
4.4.1	Syntax	90
4.4.2	Concrete semantics	94
4.4.3	Symbolic semantics	97
4.4.4	Relating the concrete and symbolic semantics	101
4.5	Model checking $\mathcal{A}llTL$	101
4.5.1	Duplication	102
4.5.2	Valuations	104
4.5.3	Tableau graph for $\mathcal{A}llTL$	106
4.5.4	Complexity	120
4.6	Related work	123
5	Dynamic References	127
5.1	Introduction	127
5.2	A logic for navigation	129
5.2.1	Semantics	129
5.3	ABA and HABA with references	131

5.3.1	Morphisms	132
5.3.2	Allocational Büchi Automata	136
5.3.3	Reallocations and HABA	137
5.4	Relating HABA and ABA	147
5.5	A language for navigation	151
5.5.1	Syntax	151
5.5.2	Adding program variables to $\mathcal{Nall}TL$	152
5.6	Operational semantics	153
5.6.1	Preliminary terminology, assumptions and results	155
5.6.2	Concrete semantics	157
5.6.3	Canonical form for HABA states	161
5.6.4	Symbolic semantics	167
5.6.5	Symbolic operational rules	169
5.6.6	Relating the concrete and symbolic semantics	172
5.7	Model checking $\mathcal{Nall}TL$	174
5.7.1	Stretching HABA	175
5.7.2	Valuations	178
5.7.3	Tableau-graph for $\mathcal{Nall}TL$	186
5.7.4	Paths	192
5.7.5	Discussion and future work: the HABA emptiness problem.	193
5.8	Related work	195
6	An application: analysis of Mobile Ambients	197
6.1	Introduction	197
6.2	An Overview of Mobile Ambients	198
6.2.1	Syntax	198
6.2.2	Operational semantics	199
6.3	An analysis oriented semantics with HABA	200
6.3.1	Motivating examples	201
6.3.2	HABA modelling approach	202
6.3.3	Process indexing	205
6.3.4	Preliminary notation	206
6.3.5	Pre-initial and initial state: an overview	207
6.3.6	On morphisms and canonical form for mobile ambients	209
6.3.7	Coding processes into HABA configurations	211
6.3.8	Pre-initial and initial state construction	217
6.3.9	Configuration link manipulations	218
6.3.10	A HABA semantics of mobile ambients	221
6.4	Related work	225
7	Conclusions and Future work	229
7.1	Achievements	229
7.2	Future work	231

Bibliography	235
A Proofs of Chapter 4	245
A.1 Proofs of Section 4.2	245
A.2 Proofs of Section 4.3	246
A.3 Proofs of Section 4.4	250
A.4 Proofs of Section 4.5	254
B Proofs of Chapter 5	263
B.1 Proofs of Sections 5.3 and 5.4	263
B.2 Proofs of Section 5.5	267
B.3 Proofs of Section 5.7	282
C Proofs of Chapter 6	295
Notations	301
Index	307
Samenvatting	311

1

Introduction

1.1 The challenges of model checking object-based software

Software verification aims to prove that a given software artifact behaves according to the original intentions of its designer. If we exclude some classical toy examples such as “Hello world” or some very specific and well-studied routines reported in every introductory book to programming, such as binary search or sorting algorithms, it is well-known that every software product has bugs and therefore misbehaves. This statement seems to be valid regardless from the programming paradigm used. It was true at the time when assembly languages were employed, and it is still valid nowadays where the object-oriented paradigm is manifestly dominating the world software development scene. The object-oriented methodology has undoubtedly contributed with remarkable improvements in the software development process, but unfortunately it does not represent the ultimate solution. Also object-oriented software is buggy. This is of course not because of a weakness of the object-oriented methodology itself, it is simply a natural consequence of the fact that humans make mistakes.

The work carried out in this thesis lies in this rather general context: *verification of object-oriented systems*. Among the different methodologies studied for software verification, we follow *model checking* [29], a formal technique which has been shown very successful in other fields such as hardware verification.

The application of model checking technology consists of three major phases: *modelling*, *property specification*, and *verification*. In the modelling phase, one constructs a formal model of the system — either manually or automatically

— containing all the relevant aspects of the software to be analysed. In the property specification phase, one formulates the requirements that the system must fulfil in some formalism. In the verification phase, one uses a tool, called a model checker, to check whether indeed the system meets the desired requirements. The tool may detect an error, in which case a manual analysis of the verification results must be carried out in order to single out what went wrong.

These three tasks, accomplished by the user, represent the front-end of the complete model checking procedure. However, the design of model checking technologies requires also a relevant effort confined to the back-end and hidden from the final user inside the model checker. That phase consists of the design of algorithms that represent the foundations for the implementation of the model checker engine (*underlying algorithms design phase*)¹.

Traditionally, model checking has been applied to mostly hardware and communication protocols. The design and the application of model checking for object-based software is usually more problematic than for hardware systems. This applies to each of the aforementioned phases. Let us try to quickly illustrate the major difficulties involved.

Beyond classical model abstraction and towards model extraction.

Certainly, modelling is not an easy task. Making a formal model of a software program can be as difficult (or even more difficult) as making the program itself. There exists a gap between a software artifact and the reality that it tries to implement, as well as between the software artifact and the verification model used by the model checker. Therefore manual modelling can be as error-prone as writing a program. Validating an erroneous model is both useless and time consuming. Human modelling may, on the one hand, introduce errors in the verification model that are not actually present in the software. Those would cause *false negatives*. Normally they can be discovered, since they will appear as counterexamples, and by a careful analysis it can be checked whether or not they are real errors in the software. On the other hand, manual modelling can hide some errors present in the program. This would produce *false positives* that are rather difficult to debug [62]. Given the ever increasing complexity and the excessive growth of the size of software, it is easy to imagine that manual modelling for software does *not* scale up. Other techniques must be applied.

Like many state-of-the-art software model checkers are investigating nowadays, a reasonable strategy would be to automatically *extract* the verification model from the source code by some process of compilation. This would have the advantage of reducing the gap between the two artifacts (program and verification model) therefore diminishing the effort for the application of model checking and, most importantly, eliminating the errors caused by the inconsistency between the model and the program. Moreover, any change in the source

¹This phase, which releases the final user from intricate technical details, has conferred to model checking the adjective of “push-button” technology, which is most probably the reason for its success even outside of the academic world.

code would be directly reflected in the model without any need of manual updating. Ultimately, this would correspond to specifying the verification model directly by a programming language. This in turn would mean releasing the user even from the modelling phase, which would be confined among the tasks carried out by the designer of the model checker.

The major obstacle to the straightforward implementation of this promising strategy is an extreme degeneration of the usual fundamental problem of model checking known as *state-space explosion*. For hardware systems the manifestation of this phenomenon corresponds to the exponential combinatorial explosion of the number of states w.r.t. the number of modelled components. For software the situation is even more frightening since the problem becomes the *infinite state-space explosion*. This difference comes mainly from the fact that object-based software artifacts are written in programming languages supporting *dynamic constructs*, such as unbounded dynamic *allocation* and *deallocation* (birth and death) of objects and threads, unbounded recursion, etc². Aggressive abstraction must be applied at the (unfortunate) price of having methodologies that may be not necessarily complete, i.e., possibly providing false negatives.

Yet another typical dynamic aspect of object-based software (that also induces infinite-state spaces) comes from the capability proper of objects to refer to other allocated objects by means of references (pointers). From state to state, references between objects are added as well as deleted and modified. This gives to the heap (where objects are allocated) the nature of a topological structure which, along with the time, evolves in an unpredictable and intricate manner. It has been well known for already thirty years that pointers are an error-prone mechanism because of the wide range of difficult to control side effects [60]. The advent of object-based languages has alleviated the problem derived by explicit use of pointers, but only to a limited extent: objects can still refer to each other. Therefore, it is easy to encounter unwanted mysterious behaviours of the program or even the more common run-time safety violations such as dereferencing null or disposed pointers, whose detection is beyond the scope of type systems. These issues must be addressed at run-time where exceptions are possible throughout the execution of the program.

A motivating example. Consider the methods listed in Table 1.1 written in some hypothetic Java-like object-based programming language. They manipulate linked lists. More precisely, `erase(l)` deletes the object list `l` in a reverse order starting from its last node. The primitive `del(x)` deallocates the object `x`. The method `append(l, x)` adds the object node `x` to the end of the list `l`. Finally, `reverse(l)` reverses the order of the elements in `l`. This last method is a rather standard example in the literature (see for example [94, 100]).

²Other (static) forms of infinity come from the use of data structures. This is true in general even for software that is not strictly object-based. Techniques as abstract interpretation [36] have been successfully applied in this direction. We will not be concerned with these issues in this dissertation.

Assume that `RequestQueue` is a FIFO queue implemented by a linked list and used in some web-server that provides services to clients over the Internet. The web-server has a thread, say t_1 , accepting client requests. Once it receives a request, t_1 creates a new *call* for it which is appended to the end of `RequestQueue` to be processed later on. Essentially t_1 executes the following code:

```
while (true) {
    getRequest(client);
    append(RequestQueue, new Call(client))
}
```

The statement `new Call(client)` creates a fresh object of the class `Call`. The size of `RequestQueue` is not fixed since the number of requests is not known in advance and can grow unboundedly, in which case the state space of this simple example becomes *infinite*. Moreover, for the implementation of `append` and `delete` in Table 1.1, there are no a priori bounds on the number of recursive method calls. Assume also, that `RequestQueue` is shared by other threads of the web-server fulfilling other tasks. The thread t_2 is a special thread, invoked in some emergency circumstances, which acts as a kind of collector resetting the list of requests by a call to `erase(RequestQueue)` and freeing the corresponding memory. Moreover, although the web-server follows a FIFO policy, in special cases, there can be a change that causes the processing of the current requests in the queue in reverse order. The thread t_3 implements this specific policy by calling `reverse(RequestQueue)`.

This rather naive implementation of the system suffers from the classical concurrency problems derived by sharing `RequestQueue`. Interference is possible if the executions of the methods are not done in a mutually exclusive way, causing the resulting queue to be what t_1 , t_2 and t_3 are not actually expecting. For example, some of the client requests may not be deallocated after calling `erase(RequestQueue)`, which will eventually create problems of memory leak. Again, memory leak can occur even in the case of some simultaneous invocations of `append` (if for example the web-server has more than one thread collecting requests) causing the loss of some requests. Another possible side-effect can be that the execution of `reverse` will not actually reverse the complete queue. Hence, several properties are relevant on such a system, like: does there exist memory leak? or may some received requests not be enqueued? or after the invocation of `reverse`, is the queue actually properly reversed? etc. \square

Most of the existing model checkers have been designed and applied to hardware systems and communication protocols. They are therefore not tailored to treating the dynamic aspects of object-based software in an adequate manner; these model checkers have instead *static* and *bounded* input languages.

Dealing with the dynamics of object-based software involves both the internal representation of the model exploited by the tool (typically a kind of finite-

<pre> void erase(LinkedList l) { if (l.nxt != null) { erase(l.nxt); del(l); } else return; } void append(LinkedList l, Call x){ if (l.nxt != null) { append(l.nxt,x) } else { l.nxt=x; } return; } </pre>	<pre> void reverse(LinkedList l){ w=null; while (l != nil) { t=w; w=l; l=l.nxt; w.nxt=t; } t=null; return; } </pre>
--	---

Table 1.1: Example methods that manipulate lists.

state automaton) and the algorithms acting with this model³. Consequently, concerning the modelling phase and the design of the underlying algorithms, there is certainly the need for new appropriate techniques with the ability to meet these challenges.

Specification. Like modelling, also the specification of properties can be rather hard and error prone. Unfortunately, there is not too much space left for automation. Maybe some of the relevant properties referring to the dynamics of object-based software can be encoded in some tricky way in standard first-order temporal logics. Nevertheless, we advocate defining some high-level specification languages, with object-based flavour, which may simplify this task by allowing more targeted (and therefore more natural) specifications.

Counter-example analysis. Finally, another difficulty in model checking is the interpretation of the error trace given by the model checker. In the case of software this can be very long, therefore making the counterexample verification a tedious (and again error-prone) job. Furthermore, the existing gap between the source code and the verification model is also present at the moment of the interpretation of the error trace. The correspondence is not straightforward especially if the model results from a non-trivial process of abstraction. Techniques for mapping an error trace back to the source code level have been developed [28, 32].

³Some existing model checkers for object-based systems rely on engines designed for static state-spaces. Therefore, the dynamic aspects introduced above are encoded in order to fit the limitation of the back-end engine. However, this approach seems to scale only up to a limited extent.

1.2 Contributions

This thesis provides a contribution on three sides: property specification, model abstraction and extraction, and verification algorithms.

On the specification side, we define a temporal logic that is aimed at specifying properties of object-based systems. The logic is called BOTL (Object-Based Temporal Logic), and its object-based ingredients are largely inspired by the Object Constraint Language (OCL) [110], a part of the UML [98] which is nowadays considered the standard for modelling object-oriented systems. BOTL is equipped with a well-defined formal semantics. In this thesis, a translation from OCL into BOTL is defined, thus providing a formal semantics of a significant subset of OCL. This sets the foundation for the development of model checking tools that use OCL as specification language. Moreover, this approach addresses a few ambiguities in OCL that have been reported in the literature [53].

We define and study two main subsets of BOTL. The first, called *Allocational Temporal Logic* ($\mathcal{All}TL$), restricts BOTL to a core subset that is amenable to capture — as primitive notions — two fundamental concepts in object-based systems: the *birth* and *death* of objects. The second subset, called $\mathcal{Nall}TL$, possesses primitives addressing dynamic references (pointers) between objects. With some abuse of mathematical notation we can express the relation between BOTL, $\mathcal{All}TL$, and $\mathcal{Nall}TL$ as follows:

$$\mathcal{All}TL \subset \mathcal{Nall}TL \subset \text{BOTL}.$$

Hence, $\mathcal{Nall}TL$ is both a subset of BOTL as well as an extension of $\mathcal{All}TL$.

Concerning modelling and (model) abstraction, this thesis contributes with the definition of two formalisms. The first one is represented by a special kind of automata, so-called *High-level Allocational Büchi Automata* (HABA). They can be used for the interpretation of $\mathcal{All}TL$ as well as for finite-state abstractions of certain kinds of infinite-state systems. Their main feature is their potential to model systems where allocation and deallocation of entities take place in a compact way. The second formalism is an enhancement of HABA aimed at modelling systems concerned with— besides allocation and deallocation — the representation of dynamic pointer structures. In order to provide finite models of such systems, HABA with references exploit dedicated kinds of abstractions.

Going from model abstraction towards model extraction, both HABA with and without references are used to define the operational semantics of small programming languages demonstrating, therefore, how finite models can be automatically extracted from a program. This is exemplified by the definition of a small programming language whose main features are the allocation and deallocation of entities (of which there can be unboundedly many), as well as a simplified form of navigation.

To illustrate the expressivity of our operational model (with references), and its accompanying logic, we show as an application example, their use for

the verification of security properties in the context of Mobile Ambients [19]. The latter is a popular formalism meant to model wide-area and mobile computations. We propose an analysis of mobile ambients exploiting the abstraction capabilities of our operational model to define suitable representations of (ambient) processes. Additionally, we show how $\mathcal{N}allTL$ can be used to express security properties of such processes. The $\mathcal{N}allTL$ model checking algorithm (see below) can then be applied in order to check if the properties are not satisfied by the processes.

The main contribution of the thesis is, however, on the verification side. More precisely on the definition of algorithms for model checking. First, we show that the model-checking problem for $\mathcal{A}llTL$ is decidable on HABA (without references). This is done by the definition of a tableau-based model checking algorithm which, given an $\mathcal{A}llTL$ -formula and a HABA, automatically decides whether or not the formula is satisfied in the model. Secondly, for the more complex case of $\mathcal{N}allTL$, we define an adapted model checking algorithm that verifies whether a $\mathcal{N}allTL$ formula is *not satisfiable* in a given HABA with references. The algorithm is semi-decidable and may return false negatives.

1.3 Outline

This dissertation comprises the following parts.

Chapter 2 provides the theoretical background needed for the formal machinery developed in the rest of the thesis. From the formal side, it gives an introduction to model-checking [29] and temporal logics [26, 91]. A short overview on a formalism called History-Dependent automata [83] concludes the formal background. The rest of the chapter is devoted to the introduction of those notions related to object-based systems that are mostly addressed in the successive chapters.

Chapter 3 presents the temporal logic BOTL and defines the translation from OCL into BOTL that provides a formal semantics to a subset of OCL.

A preliminary version of this chapter has been published in [42].

Chapter 4 studies the notions of birth and death by restricting BOTL to $\mathcal{A}llTL$. The first version of HABA (without references) is introduced. A small programming language for the allocation and deallocation of objects is defined and its semantics in terms of HABA is given. Finally, the model-checking problem for $\mathcal{A}llTL$ is shown to be decidable.

An extended abstract of this chapter has been published in [45].

Chapter 5 studies the extension of $\mathcal{A}llTL$ to dynamic references. $\mathcal{N}allTL$ is defined and HABAs are enhanced with dynamic pointer structures. Another simple programming language with a simplified form of navigation is introduced. Its semantics is given in terms of HABA with references.

Finally, the chapter defines a (semi-decidable) model-checking algorithm that verifies whether a $\mathcal{N}\mathit{all}\mathit{TL}$ formula is *not satisfiable* in a given HABA.

Chapter 6 describes an application of the theories developed in previous chapters to another domain. It illustrates how the developed operational model (with references) can be used to model the dynamic behaviour of Mobile Ambients and how the logic $\mathcal{N}\mathit{all}\mathit{TL}$ can be used to specify security properties.

Chapter 7 contains a summary of the main results and proposes some directions for further research.

Appendixes contain the proofs of the results given in Chapters 4, 5, and 6.

1.4 Related work: the jungle of software model checkers

The challenges discussed in Section 1.1 have been taken up by the scientific community, and lately, the substantial increase of interest in software model-checking has resulted in several different approaches, methodologies and tools. In this section, we try to give an overview of the major approaches (in particular those involving object-based/oriented systems) of which we are aware.

Each individual chapter contains a more detailed discussion of the related work.

Bandera tool set. Bandera [32], developed at Kansas State University, is a model checker for Java source code. The philosophy followed by Bandera is reusing existing verification tools. Hence, Bandera allows for the automatic extraction of a verification model analysable by a model checker, e.g., SPIN [61], dSPIN [41] or SMV [80] directly for a Java program. An interesting feature is that the output of the verification tool is mapped back to the level of the source code (like a debugging tool). This makes it possible to interpret counterexamples directly on the code. The extraction of the model from the source code employs sophisticated techniques for abstraction such as *slicing* (i.e., the removal of variables and data structures not related to the property to be proved) and *data abstraction* (by abstract interpretation). The combination of these techniques results in rather compact state spaces. For the specification of properties, Bandera supports a high-level specification language based on temporal patterns (see Section 3.6.1).

Java PathFinder. Another model checker for Java is Java PathFinder (JPF) [58], developed at NASA Ames research centre. JPF relies on a special Java virtual machine that is invoked by the model checking engine to interpret byte-code generated by the Java compiler. The analysis is done at the byte code level, thus permitting the verification of third party software where the source code is not available. Moreover, any language translated into byte-code can

be analysed. In order to reduce the state space, JPF exploits a combination of techniques (à la Bandera) such as: abstraction, static analysis, slicing, and runtime analysis. JPF uses LTL (or CTL) as a property specification language, and it has been employed in several case studies.

Slam project. The purpose of the Slam project [5] — currently being carried out at Microsoft Research — is to check temporal safety properties of C programs. Properties are encoded in a language called SLIC (Specification Language for Interface Checking) and are mostly oriented to describe the execution behaviour of the program at the level of function calls and returns.

Given a program P and a specification S , a preprocessing phase creates a new program P' that reaches an error state if and only if the original program P does not conform to the specification S . The analysis itself is done on a sound abstraction of P' in terms of so-called *boolean programs* that are automatically generated using predicate abstraction [54]. Boolean programs have control-flow constructs typical of C, but only admit boolean variables. The applied abstraction is sound but not complete. Thus *false* counterexamples may be generated. Therefore, for every detected error state a careful process of counterexample analysis must be carried out. The corresponding SLAM toolkit has been used to validate the behaviour of some Windows XP device drivers.

FeaVer. FeaVer [63], developed at Bell Labs, is a tool that can automatically extract Promela⁴ models from C programs. As for every other software verification tool, for the generation of the model out of the source code, FeaVer makes use of abstraction techniques such as slicing. The resulting models can then be verified by the model checker SPIN. FeaVer has been applied successfully in an industrial case study at Lucent Technologies. The objective of this case study was the design of the call processing software for a new telephone switching system.

3-valued logic A model checker, called 3VMC, for concurrent Java programs aimed at the verification of safety properties was introduced in [112]. The main applications concern: detection of *interference* between threads accessing the same shared object; *deadlock* detection in concurrent programs; verification that shared linked lists preserve some properties under concurrent manipulation. The tool is able to handle an unbounded number of threads and objects, and it is based on the 3-valued logic engine of TVLA [74]. The latter is a system that, given the operational semantics of a program expressed in 3-valued logic, automatically generates an abstract interpretation analysis algorithm. The model checker 3VMC is sound but not complete since it may give spurious counterexamples.

⁴Promela is the input language of the SPIN model checker [61].

BLAST. The tool BLAST (Berkeley Lazy Abstraction Software verification Tool) [59] verifies properties of C programs starting from a coarse predicate abstraction of the source code. The abstraction is automatically refined on-the-fly until a bug in the program is detected or the correctness of the specification is proved. Large C programs have been verified using BLAST, in particular several Windows and Linux device drivers.

2

Preliminaries

This thesis is driven by two rather unrelated worlds: *formal verification* and *object-based systems*. The present chapter is meant to set up the theoretical background framework needed for the formal machinery developed in later chapters. We will start our overview with an introduction on the formal verification technique called *model-checking* [29] that we have chosen to exploit and the related specification languages, i.e., *temporal logics*. Moreover, we give a quick introduction to History-Dependent automata, a formalism that has been the source of inspiration for a number of ideas in this thesis. We will continue our description by introducing some notion related to object-based systems focusing on those particular concepts and problems we will try to tackle later on.

2.1 Model checking and temporal logics

Model checking is a formal technique used for the verification of hardware and software systems represented by finite models (e.g., automata) that in turn are suitable abstractions of real systems. The model checking slogan is not at all sophisticated but, in fact rather simple, and can be stated in two words: *brute force*. The idea is to explore exhaustively *all* the possible states of the system in order to verify whether a property may be satisfied or not. The strength of model checking is that the exhaustive search is done mechanically by a tool called the *model checker*. Nowadays, state of the art model checkers can handle rather large state spaces: up to $10^8 - 10^9$ states. In special cases suitable to optimisations, in the literature have been reported experience of systems with

10^{30} [27] and even 10^{476} [105] states. The latter is a number far beyond the estimated amount of atoms in the universe.

Although model checking has been used in the past especially for the verification of hardware systems and/or designs as well as for protocol verification, the interest of the research community is lately devoted also to model checking techniques for software systems. In this thesis, we restrict ourselves to this last case.

The tasks involved in the application of model checking can be summarised as follows:

- **Modelling.** From the software system and/or design, a *model* expressed in a formalism accepted by a model checker must be defined. The definition can be done either manually or automatically by extracting the model from the software itself, e.g., by some kind of compiler (cf. Section 1.1). It is in any case necessary that the model presents all the relevant aspects to be analysed.
- **Properties specification.** Since model checking allows to formally verify that a system (or a design) fulfils given properties, we need a language to express them. Normally, the language used for the specification of requirements is a *temporal logic*.
- **Verification.** This phase is done by the model checker. The human activity is confined to the analysis of the verification results, e.g., counterexamples. For software verification the counterexample may be very lengthy making therefore the human interpretation a rather hard task (cf. Section 1.1).

Because of the verification phase, model checking has been several time addressed in the literature as a *push-button* technology. However, applying model checking as a whole to non-trivial systems may require considerable human effort and skills especially during the first two phases (modelling and property specification). *Sound abstractions* must be designed in order to have meaningful verification results. General abstractions, suitable for a wide range of problems are difficult to define.

This section is based on [29, 71, 79] and it is organised as follows. Section 2.1.1 introduces two mathematical models used in model checking, namely Kripke structures and Büchi automata. Sections 2.1.2 and 2.1.3 present two very well studied temporal logics, i.e., LTL and CTL. Section 2.1.4 gives the formal definition of model-checking, satisfiability and other related notions. Then Sections 2.1.5 and 2.1.6 introduce two different approaches to model checking. Finally Section 2.1.7 compares the two methods.

2.1.1 Kripke structures and Büchi automata

Kripke structure [72] and Büchi automata [16] are mathematical models used extensively for the definition of the semantics of temporal logics.

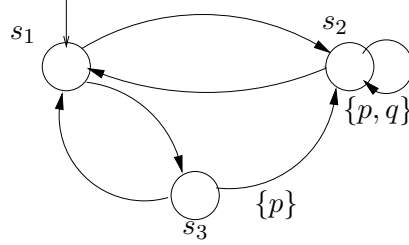


Figure 2.1: Example of Kripke structure.

Throughout this section, let AP be a finite set of *atomic propositions* ranged over by p, p', p_1, r , etc. As for propositional logic, atomic propositions are the most basic statements expressible in temporal logic.

Definition 2.1.1 (Kripke structure). A *Kripke structure* is a tuple $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$ where:

- S is a countable set of states;
- $I \subseteq S$ is a set of initial states;
- $\rightarrow \subseteq (S \times S)$ is a transition relation satisfying $\forall s \in S. (\exists s' \in S. (s, s') \in \rightarrow)$
- $L : S \rightarrow 2^{AP}$ is an *interpretation function* on the set of state S , i.e., a function that assigns a set of atomic propositions to each state.

In the following, we write $s \rightarrow s'$ as a shorthand for $(s, s') \in \rightarrow$. The transition relation is total because of the condition imposed in the definition.

Definition 2.1.2 (Path). Let $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$ be a Kripke structure. A *path* in \mathcal{K} is an infinite sequence of states $s_0 s_1 s_2 \dots$ such that $s_i \rightarrow s_{i+1}$ for all $i \geq 0$.

As a convention for a path $\eta = s_0 s_1 s_2 \dots$ we write $\eta[i]$ to denote the $(i + 1)$ -th element and η^i for the suffix of η starting at state $i + 1$, that is $\eta^i = s_i s_{i+1} s_{i+2} \dots$. The set of paths starting at state s is defined by

$$Path_{\mathcal{K}}(s) = \{\eta \in S^\omega \mid \eta[0] = s, \forall i \geq 0: \eta[i] \rightarrow \eta[i + 1]\} .$$

As a consequence of the condition imposed upon the transition relation of a Kripke structure \mathcal{K} , we have $Path_{\mathcal{K}}(s) \neq \emptyset$ for any state s .

Example 2.1.3. An example of Kripke structure is depicted in Figure 2.1. We have:

$$\begin{aligned}
 S &= \{s_1, s_2, s_3\} \\
 I &= \{s_1\} \\
 \rightarrow &= \{(s_1, s_2), (s_2, s_1), (s_1, s_3), (s_2, s_2), (s_3, s_2), (s_3, s_1)\} \\
 L &= \{(s_1, \emptyset), (s_2, \{p, q\}), (s_3, \{p\})\}.
 \end{aligned}$$

□

Definition 2.1.4 (Labelled Finite State Automaton). A *labelled finite-state automaton* (LFSA), over finite words, is a tuple $\mathcal{A} = \langle \Sigma, S, \rightarrow, I, F, L \rangle$ where

- Σ is a finite *alphabet*;
- S is a finite set of states;
- $\rightarrow \subseteq S \times S$ is a transition relation;
- $I \subseteq S$ is a set of initial states;
- $F \subseteq S$ is a set of *accept* states;
- $L : S \rightarrow \Sigma$ is a function labelling the states.

An equivalent and more widespread definition usually given, e.g., in compiler theory, labels transitions instead of states. Following [51, 71], here we prefer this definition since it suits better with the theory developed later on.

As usual, for a finite alphabet Σ , let Σ^* represent the set of all finite words and Σ^ω the set of all infinite words, both over Σ .

Definition 2.1.5. For a LFSA $\mathcal{A} = \langle \Sigma, S, \rightarrow, I, F, L \rangle$,

- a *run* ρ is a finite sequence of states $\rho = s_0 s_1 \cdots s_n$ such that $s_0 \in I$ and $s_i \rightarrow s_{i+1}$ for $0 \leq i < n$. ρ is called *accepting* if $s_n \in F$.
- $w = a_0 a_1 \cdots a_n \in \Sigma^*$ is an *accepted* word if there exists an accepting run $\rho = s_0 s_1 \cdots s_n$ such that $L(s_i) = a_i$ for all $0 \leq i \leq n$.
- the language accepted is the set $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid w \text{ is accepted by } \mathcal{A}\}$.

LFSA's are often used to model the behaviour of terminating programs. Since, concurrent systems not always are supposed to terminate, their behaviour is modelled by finite automata accepting *infinite* words, called Büchi automata [16]. Büchi automata are defined as LFSA except for the definition of runs, which must be infinite.

Definition 2.1.6. For a Büchi automaton $\mathcal{B} = \langle \Sigma, S, \rightarrow, I, F, L \rangle$,

- a *run* ρ is an infinite sequence of states $\rho = s_0 s_1 s_2 \cdots$ such that $s_0 \in I$ and $s_i \rightarrow s_{i+1}$ for $i \geq 0$. Let $\text{Inf}(\rho)$ be the set of states occurring infinitely often in ρ . Then ρ is called *accepting* if $\text{Inf}(\rho) \cap F \neq \emptyset$.
- $w = a_0 a_1 a_2 \cdots \in \Sigma^\omega$ is an *accepted* word if there exists an accepting run $\rho = s_0 s_1 s_2 \cdots$ such that $L(s_i) = a_i$ for all $i \geq 0$.
- the language accepted is the set $\mathcal{L}(\mathcal{B}) = \{w \in \Sigma^\omega \mid w \text{ is accepted by } \mathcal{B}\}$.

From this definition it follows that a run ρ is accepting if some accepting state occurs infinitely often.

There exist several kinds of Büchi automata that are obtained by changing the notion of acceptance. An important one that we will use in the following, is the notion of *generalised* Büchi automaton. In particular, a generalised Büchi automaton has a set of sets of accept states of the form $F \subseteq 2^S$ (where S is the set of states). If $F = \{F_0, \dots, F_n\}$, a run of a generalised Büchi automaton is accepting if

$$\text{Inf}(\rho) \cap F_i \neq \emptyset \quad \forall 0 \leq i \leq n \quad (2.1)$$

that is, for each set of accept states F_i , there is some state occurring infinitely often in ρ . It is interesting to note that the generalised acceptance condition does not extend the set of languages accepted by Büchi automata.

Kripke structures versus Büchi automata. The two formal notion of Kripke structure and Büchi automaton are closely related. In particular, a Kripke structure directly corresponds to a Büchi automaton where all states are accepting and the alphabet is the power set of the atomic proposition. That is: let $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$, then the corresponding Büchi automaton is $\mathcal{B} = \langle 2^{AP}, S, \rightarrow, I, S, L \rangle$. Vice-versa, a Büchi automaton, can be seen as a Kripke structure with a fairness condition determined by the set of accept states. In particular, a generalised Büchi automaton is sometimes called in the literature *fair Kripke structure* [29]. Hence, every notion introduced in this chapter for Kripke structures can be restated in terms of Büchi automata¹

More on Büchi automata. We now introduce some results on Büchi automata that will be useful in Section 2.1.5. Given a Büchi automaton \mathcal{B} , there exists an automaton $\overline{\mathcal{B}}$ such that

$$\mathcal{L}(\overline{\mathcal{B}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B}) \quad (2.2)$$

i.e., Büchi automata are closed under negation. For the rather involved construction of $\overline{\mathcal{B}}$, the reader is referred to [103]. Büchi automata are also closed under intersection, that is given \mathcal{B}_1 and \mathcal{B}_2 there exists the automaton $\mathcal{B}_1 \cap \mathcal{B}_2$ such that

$$\mathcal{L}(\mathcal{B}_1 \cap \mathcal{B}_2) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2). \quad (2.3)$$

The construction of the intersection (or product) automaton is a modification of the definition of synchronous product of LFSA. The difficulty is given by the Büchi acceptance condition that must be satisfied. $\mathcal{B}_1 \cap \mathcal{B}_2$ can be constructed as follows. Let $\mathcal{B}_1 = \langle \Sigma, S_1, \rightarrow_1, I_1, F_1, L_1 \rangle$ and $\mathcal{B}_2 = \langle \Sigma, S_2, \rightarrow_2, I_2, F_2, L_2 \rangle$ then $\mathcal{B}_1 \cap \mathcal{B}_2 = \langle \Sigma, S, \rightarrow, I, F, L \rangle$ where:

- $S = \{(s, s') \in S_1 \times S_2 \mid L_1(s_1) = L_2(s_2)\} \times \{1, 2\}$
- $I = (I_1 \times I_2) \times \{1\} \cap S$

¹Being this chapter an overview on well established theories, when there will be the choice between Kripke structures and Büchi automata we will try to follow what in the literature is more widespread.

- if $s_1 \rightarrow_1 s'_1$ and $s_2 \rightarrow_2 s'_2$ then
 - (i) if $s_1 \in F_1$ then $(s_1, s_2, 1) \rightarrow (s'_1, s'_2, 2)$
 - (ii) if $s_2 \in F_2$ then $(s_1, s_2, 2) \rightarrow (s'_1, s'_2, 1)$
 - (iii) otherwise $(s_1, s_2, i) \rightarrow (s'_1, s'_2, i)$ for $i = 1, 2$
- $F = (F_1 \times S_2) \times \{1\} \cap S$
- $L(s, s', i) = L_1(s)$.

This construction was proposed in [23]. The automaton $\mathcal{B}_1 \cap \mathcal{B}_2$ makes transitions when \mathcal{B}_1 and \mathcal{B}_2 perform steps on the same label. $\mathcal{B}_1 \cap \mathcal{B}_2$ can be thought as having two tapes: states with third component 1 form the first tape whereas those with third component 2 form the second tape. The transition relation ensures that in order to visit infinitely many times the accept states in F , the accept states of both \mathcal{B}_1 and \mathcal{B}_2 must be visited infinitely many times as well. For the definition of F , it is enough to consider states in the tape 1 since by the transition relation the $\mathcal{B}_1 \cap \mathcal{B}_2$ switches tape when visiting a state in F . Therefore, in order to visit a state in F again, the automaton must switch from tape 2 to tape 1. But, because of the definition of the transition relation this can happen only if the control of the automaton passes through an accept state of \mathcal{B}_2 . Thus, a run of $\mathcal{B}_1 \cap \mathcal{B}_2$ traverses infinitely many times F precisely when both \mathcal{B}_1 and \mathcal{B}_2 traverse infinitely many times their set of accept states F_1 and F_2 . Note that setting $F = F_1 \times F_2$ would not work since it requires not only that \mathcal{B}_1 and \mathcal{B}_2 visit accept states both infinitely often, but also that they do it *at the same time* ruling out, hence, some good runs.

2.1.2 Linear Temporal Logic

In general there exists two main branches of temporal logics and their classification is based on the way time is modelled: *linear time* or *branching time*. For the interpretation of a formula, in the linear case every state may only have one successor state. On the contrary, in the branching case every state may have more than one.

The main representative of the first kind is Linear Temporal Logic (LTL) introduced by Pnueli in 1977 [91].

Syntax of LTL. Let $p \in AP$ be an atomic proposition. The syntax of Linear Temporal Logic is given by:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi.$$

Intuitively, the meaning of the formulae is as follows: in a state $\neg\phi$ holds if ϕ does not hold; $\phi_1 \vee \phi_2$ holds if either ϕ_1 holds or ϕ_2 holds; $X\phi$ holds in a state if ϕ holds in the successive state. X is called *next* operator and represents the first temporal operator we encounter. Finally, $\phi_1 U \phi_2$ holds if ϕ_2 eventually will

hold for some state and — until that state — ϕ_1 continuously holds. \mathbf{U} is called *until* operator. From this basic syntactic definition of LTL, it is customary to define other connectives as syntactic sugar:

$$\begin{array}{lll}
\mathbf{tt} & \equiv & p \vee \neg p \\
\mathbf{ff} & \equiv & \neg \mathbf{tt} \\
\phi \wedge \psi & \equiv & \neg(\neg\phi \vee \neg\psi) \\
\phi \Rightarrow \psi & \equiv & \neg\phi \vee \psi \\
\phi \Leftrightarrow \psi & \equiv & (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \\
\mathbf{F}\phi & \equiv & \mathbf{tt} \mathbf{U} \phi \quad [\text{eventually } \phi] \\
\mathbf{G}\phi & \equiv & \neg\mathbf{F}(\neg\phi) \quad [\text{globally } \phi]
\end{array}$$

Most of these abbreviations are standard in propositional logic, the last two are typical of LTL. Intuitively, $\mathbf{F}\phi$ means that either ϕ is true now or it will become true in some state in the future. $\mathbf{G}\phi$ means that ϕ holds in the current state and it holds in every state in the future.

Semantics of LTL. The term *linear* in LTL derives from the fact that formulae are interpreted over sequences of states representing computation of a system modelled by a Kripke structure $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$. These sequences are paths of \mathcal{K} .

Let $p \in AP$ be an atomic proposition, η a path and ϕ, ψ be LTL formulae. The satisfaction relation \models is given by:

$$\begin{array}{ll}
\eta \models p & \text{iff } p \in L(\eta[0]) \\
\eta \models \neg\phi & \text{iff not } (\eta \models \phi) \\
\eta \models \phi \vee \psi & \text{iff either } (\eta \models \phi) \text{ or } (\eta \models \psi) \\
\eta \models \mathbf{X}\phi & \text{iff } \eta^1 \models \phi \\
\eta \models \phi \mathbf{U} \psi & \text{iff } \exists j \geq 0: \eta^j \models \psi \wedge (\forall 0 \leq k < j: \eta^k \models \phi)
\end{array}$$

As for propositional logic, an atomic proposition p is true in a state $\eta[0]$ if it is among the interpretations given by $L(\eta[0])$. A negated formula $\neg\phi$ is valid in a state if ϕ is not valid in the same state. The disjunction $\phi \vee \psi$ is valid when at least one of ϕ , or ψ is valid in the same state. The next operator is more interesting since it is proper of temporal logics. $\mathbf{X}\phi$ is satisfied by a path η if the formula ϕ is valid in the path starting from the successor state of $\eta[0]$. Finally, the until formula $\phi \mathbf{U} \psi$ is satisfied if ψ become valid on a path starting at $\eta[0]$ or starting from a state following $\eta[0]$, say $\eta[j]$ with $j \geq 0$. Furthermore, it is required that in every path starting with a state between $\eta[0]$ and the predecessor of $\eta[j]$, the formula ϕ is continuously valid.

2.1.3 Computation Tree Logic

For the sake of completeness, we give some notions of branching temporal logic. Although in the remaining of the thesis we will focus on linear temporal logics,

the present overview may help the reader to realise that several of the definition introduced in the following chapters in a linear setting can be restated, and therefore applied, in a branching temporal logic.

Computation Tree Logic (CTL), introduced by Clarke and Emerson in 1981 [26], is a temporal logic that has a *branching* notion of time (in contrast with LTL), i.e, for every state there can be more than one possible successor state. The underlying notion of semantics of a branching temporal logic is a *tree* of states rather than a sequence. Like a sequence, a tree is obtained from a Kripke structure by a process of “unfolding”. The outgoing transitions starting from a state represent the possible next states, and CTL allows to reason about *all* or *some* of the paths by a special quantifier over paths.

Syntax of CTL. As usual let AP be a set of atomic propositions ranged over by p, q, r . The syntax of CTL is defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid EX\phi \mid E[\phi U \phi] \mid A[\phi U \phi].$$

The temporal operators have the following intuitive meaning:

- $EX\phi$ expresses that there is a next state in which the formula ϕ holds.
- $E[\phi U \psi]$ expresses that there exists a path starting from the current state along which ψ holds at a given state, and ϕ holds in every state before.
- $A[\phi U \psi]$ expresses that along *every* path starting from the current state, ψ holds at a given state and ϕ holds in every state before.

As usual, by combining these basic operators we can obtain other auxiliary operators. The first two correspond to the case where the first formula is tt . We have:

$$\begin{aligned} EF\phi &\equiv E[\text{tt} U \phi] \quad [\text{potentially } \phi] \\ AF\phi &\equiv A[\text{tt} U \phi] \quad [\phi \text{ is inevitable}]. \end{aligned}$$

Finally, by negation we can define the (very useful) duals of the previous operators:

$$\begin{aligned} AX\phi &\equiv \neg EX\neg\phi \quad [\text{for all path next } \phi] \\ AG\phi &\equiv \neg EF\neg\phi \quad [\text{invariantly } \phi] \\ EG\phi &\equiv \neg AF\neg\phi \quad [\text{potentially always } \phi]. \end{aligned}$$

Given a state s the previous connectives have the following intuitive meaning. $EF\phi$ is valid in s if and only if eventually ϕ holds along at least one of the paths starting at s , whereas $AF\phi$ is valid in s if and only if ϕ holds in all paths starting at s . The formula $EG\phi$ holds in s if there exists some path starting at s where ϕ holds along every state of this path. Symmetrically, $AG\phi$ holds in s if and only if ϕ holds in every state of every path starting at s .

Semantics of CTL. Also the semantics of CTL is based on the notion of Kripke structure. Given a Kripke structure $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$ and state $s \in S$, there exists a infinite computation tree rooted at s such that (s', s'') is an arc of the tree if and only if $s' \rightarrow s''$ and s' is reachable from s .

The semantics of CTL is defined by satisfaction relation \models between the Kripke structure \mathcal{K} , a state s and a formula ϕ . We write $\mathcal{K}, s \models \phi$ if and only if ϕ holds in the state s of \mathcal{K} . The structure \mathcal{K} is usually omitted if it is clear from the context.

Let $p \in AP$ be an atomic proposition, $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$ be a Kripke structure, $s \in S$, and ϕ a CTL formula. The satisfaction relation \models is defined by:

$$\begin{aligned}
s \models p & \quad \text{iff } p \in L(s) \\
s \models \neg\phi & \quad \text{iff not } (s \models \phi) \\
s \models \phi \vee \psi & \quad \text{iff either } (s \models \phi) \text{ or } (s \models \psi) \\
s \models \text{EX}\phi & \quad \text{iff } \exists \eta \in \text{Path}_{\mathcal{K}}(s) : \eta[1] \models \phi \\
s \models \text{E}[\phi \text{U} \psi] & \quad \text{iff } \exists \eta \in \text{Path}_{\mathcal{K}}(s) : (\exists j \geq 0 : \eta[j] \models \psi \wedge (\forall 0 \leq k < j : \eta[k] \models \phi)) \\
s \models \text{A}[\phi \text{U} \psi] & \quad \text{iff } \forall \eta \in \text{Path}_{\mathcal{K}}(s) : (\exists j \geq 0 : \eta[j] \models \psi \wedge (\forall 0 \leq k < j : \eta[k] \models \phi)).
\end{aligned}$$

The interpretation of atomic propositions, negation, disjunction is the same as for LTL. $\text{EX}\phi$ is true if there exists a path among those starting at s that satisfies ϕ in the next state. The interpretation of $\text{A}[\phi \text{U} \psi]$ and $\text{E}[\phi \text{U} \psi]$ is the same as $\phi \text{U} \psi$ in LTL, except that there is the existential and the universal quantification on the paths starting at s . Namely, we require that the until formula holds for at least one path in $\text{E}[\phi \text{U} \psi]$ and for every path in $\text{A}[\phi \text{U} \psi]$.

2.1.4 A few formal notions relating formulae and models

Given a Kripke structure \mathcal{K} modelling the behaviour of a system and a formula ϕ expressed in some temporal logic, e.g., LTL or CTL, representing a specification of the system, we have several notions relating \mathcal{K} and ϕ . We give the definition for the case of LTL.

Definition 2.1.7. Let $\mathcal{K} = \langle S, \rightarrow, I, L \rangle$ be a Kripke structure and ϕ a LTL formula:

$$\mathcal{K} \models \phi \quad \text{iff } \forall s \in I : (\forall \eta \in \text{Path}_{\mathcal{K}}(s) : \eta \models \phi)$$

When $\mathcal{K} \models \phi$ we say that \mathcal{K} *satisfies* ϕ .

The notion of satisfiability given by Definition 2.1.7 corresponds to the informal idea that an implementation satisfies a specification, that is: if no possible computation of the system does violates the specification given by ϕ . Definition 2.1.7 takes origin in logic from the well-known *satisfiability problem* (SAT) defined in Table 2.1. For propositional LTL, (SAT) is decidable, but for other logics it is not (e.g., first-order logic). Indeed this is a rather hard problem since only ϕ is given whereas \mathcal{K} must be somehow determined and/or constructed.

Dual to the satisfiability problem is the *validity problem* (VAL) again defined in Table 2.1. To decide (VAL) for ϕ , one can solve the the satisfiability problem for $\neg\phi$ since if a formula is valid then its negation is unsatisfiable.

The third problem defined in Table 2.1 is the *model checking problem* (MC). In that case, the Kripke structure \mathcal{K} is given and we must “only” verify that it satisfies ϕ . The model checking problem conforms more to the idea of software verification, since it essentially tries to verify whether a model of an implementation (i.e., \mathcal{K}) satisfies a specification (i.e., ϕ). In contrast, an algorithm showing (SAT) can be used as a feasibility test for a specification ϕ : in fact, if ϕ does not pass the test it cannot be implemented at all, therefore another specification must be provided.

In this thesis, we deal with issues related to (MC). Hence, every time, we consider the case where \mathcal{K} is given. We can see \mathcal{K} in two different ways [99]:

- as a *post-design* or *pre-implementation* model; in this case \mathcal{K} is used to verify the design of a system and it can help to define a candidate implementation that is supposed to satisfy the specification ϕ . This point of view corresponds to the “classical” model checking approach.
- as *post-implementation* model; in this case \mathcal{K} is used as a verification model of an existing implementation and the aim is the debugging of the implementation. This point of view corresponds to the “modern” model checking approach (cf. Section 1.1 for related issues).

Along the lines of the two previous points of view, we refine the notion of satisfiability and validity given above in order to be more related to the idea of computations performed by \mathcal{K} . In fact, because of the two universal quantifications, Definition 2.1.7 has more the flavour of validity in terms of the computations performed by \mathcal{K} . The last two problems defined in Table 2.1 are parametric (dependent) on \mathcal{K} . The *\mathcal{K} -validity problem* (\mathcal{K} -VAL) corresponds to (MC), although the quantification (and therefore the emphasis) is on the computation rather than on the model \mathcal{K} . The dual problem of (\mathcal{K} -VAL), is the *\mathcal{K} -satisfiability problem* (\mathcal{K} -SAT). (\mathcal{K} -SAT) asks if there exists (at least) *one* computation in \mathcal{K} satisfying ϕ whereas (\mathcal{K} -VAL) asks if *every* computation of \mathcal{K} satisfies ϕ . Again the difference between (SAT), (VAL), (\mathcal{K} -SAT) and (\mathcal{K} -VAL) is that the quantification in the former two is on the Kripke structure \mathcal{K} whereas in the latter two the quantification is over the computations of a given \mathcal{K} .

Note that solving (\mathcal{K} -VAL) corresponds to solving (MC) and by solving (\mathcal{K} -SAT) for $\neg\phi$ we automatically determine an answer for (\mathcal{K} -VAL).

Moreover, the corresponding notions of (\mathcal{K} -VAL) and (\mathcal{K} -SAT) for Büchi automata will be introduced in Chapters 4 and 5 for the special case of HABAs which are the automata we are going to define and use there.

2.1.5 The automata theoretic approach to model checking

Automata, and in particular Büchi automata, can be used not only to describe the behaviour of concurrent systems but also to formalise the specifications of

(SAT)	given a formula ϕ , does there exist a Kripke structure \mathcal{K} such that $\mathcal{K} \models \phi$?
(VAL)	given a formula ϕ , is it $\mathcal{K} \models \phi$ for <i>every</i> Kripke structure \mathcal{K} ?
(MC)	given a formula ϕ and a Kripke structure \mathcal{K} , is it $\mathcal{K} \models \phi$?
(\mathcal{K} -VAL)	given a formula ϕ , is it $\forall s \in I : (\forall \eta \in Path_{\mathcal{K}}(s) : \eta \models \phi)$?
(\mathcal{K} -SAT)	given a formula ϕ , does there exist a $s \in I$ such that there exists $\eta \in Path_{\mathcal{K}}(s)$ such that $\eta \models \phi$?

Table 2.1: Definition of satisfiability, validity and model checking problem.

the systems. The advantage of using specifications by automata is that both the system and the specification are described with the same formalism.

Let \mathcal{B}_{spec} be a Büchi automaton describing the specification of a system. Then its language $\mathcal{L}(\mathcal{B}_{spec})$ is the set of allowed behaviours for the system. Let \mathcal{B}_{sys} be the Büchi automaton modelling the system. The system satisfies its specification \mathcal{B}_{spec} if

$$\mathcal{L}(\mathcal{B}_{sys}) \subseteq \mathcal{L}(\mathcal{B}_{spec}). \quad (2.4)$$

Stated in words, this set inclusion says that the set of behaviours of the system are among those behaviours allowed by the specification. We can restate (2.4) using the automaton $\overline{\mathcal{B}_{spec}}$ accepting the complement of the language $\mathcal{L}(\mathcal{B}_{spec})$ (see Section 2.1.1) in the following way:

$$\mathcal{L}(\mathcal{B}_{sys}) \cap \mathcal{L}(\overline{\mathcal{B}_{spec}}) = \emptyset. \quad (2.5)$$

If this intersection would not be empty, any infinite word contained would represent a counterexample. From (2.5) we can derive in a straightforward manner the model checking Algorithm 1 (reported in pseudo-code).

Algorithm 1 Automata theoretic procedure for model checking

```

1: procedure ModelChecking( $\mathcal{B}_{spec}, \mathcal{B}_{sys}$ ) do
2:   Construct the automaton  $\overline{\mathcal{B}_{spec}}$ ;
3:   Construct the automaton  $\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}}$ ;
4:   if  $\mathcal{L}(\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}}) = \emptyset$  then
5:     Output: “The system satisfies the specification”;
6:   else
7:     return as a counterexample a word in  $\mathcal{L}(\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}})$ ;
8:   end if
9: end procedure ModelChecking

```

In many cases, the specification is not directly expressed in terms of automata but by a specification language such as LTL. There exists a method

that automatically translates an LTL formula ϕ into an automaton, say \mathcal{B}_ϕ , that accepts precisely the runs satisfying ϕ [51]. In this method, \mathcal{B}_{spec} (of Algorithm 1) is represented by \mathcal{B}_ϕ . It is interesting to note that when \mathcal{B}_{spec} is obtained by the translation of a LTL formula ϕ , the construction of the complemented automaton (cf. step 1) can be skipped by constructing directly the automaton describing $\neg\phi$, i.e., $\mathcal{B}_{\neg\phi}$.

We have seen in Section 2.1.1 how the construction of the automaton $\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}}$ can be done. The condition $\mathcal{L}(\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}}) = \emptyset$ is not straightforward and therefore deserves some further explanation.

Checking for emptiness. Since an accepting run ρ of a Büchi automaton $\mathcal{B} = \langle \Sigma, S, \rightarrow, I, F, L \rangle$ contains infinitely many occurrences of accept states and S is finite then there must exist ρ' and ρ'' such that $\rho = \rho'\rho''$ and all the states in ρ'' appear infinitely many times. Hence, states in ρ'' must be contained in a strongly connected component (SCC), say $S_{\rho''}$. Furthermore, since ρ is a run, it follows that:

- (a) $S_{\rho''}$ is reachable from an initial state;
- (b) $S_{\rho''} \cap F \neq \emptyset$.

Conversely, to any SCC for which (a) and (b) holds, a corresponding accepting run of \mathcal{B} does exist. This suggests that in order to verify that the language is not empty, we can check for the existence of a SCC reachable from an initial state and containing at least an accepting state. Finding such a SCC boils down to finding a cycle containing some accepting state. In fact, if there exists a SCC then this must be contained in a cycle, conversely, if there exists a cycle, it must correspond to a SCC. The implication of this observation is that it is possible to represent a run of a Büchi automaton in a finite way. This is particularly interesting for runs of $\mathcal{B}_{sys} \cap \overline{\mathcal{B}_{spec}}$, since they correspond to computations in which the system does not satisfy the specification, i.e., *counterexamples*. Summarising, two steps must be done:

1. find the set F_{reach} of all accept states reachable from an initial state;
2. find a cycle containing some state $s \in F_{reach}$.

The construction of F_{reach} can be done in linear time by a depth-first search. Step 2, which amounts to detecting cycles in a finite graph, can also be done in linear time using Tarjan's algorithm [107] for constructing SCCs in a graph. The complete procedure for step 1 and 2 will have a time complexity of $O(|S| + |\rightarrow|)$. A slightly better solution is the Algorithm 2 known as *nested depth-first search* and proposed in [35]. It detects reachable accept states and cycles containing them at the same time. The advantage of this last solution is that the search can be done on-the-fly. In particular, errors are detected while traversing the state space and the procedure can be stopped as soon as the first error is caught. The procedure NestedDFS can be explained as follows.

Algorithm 2 Nested depth-first search

```

1: procedure NestedDFS( $\mathcal{B}$ ) do
2:   for all  $s \in I_{\mathcal{B}}$  do
3:     DFS1( $s$ );
4:   end for
5:   return [];
6: end procedure NestedDFS
7:
8: procedure DFS1( $s$ ) do
9:   Insert  $s$  in  $STK_1$ ;
10:  for all  $s'$  such that  $s \rightarrow s'$  do
11:    if  $s' \notin STK_1$  then
12:      DFS1( $s'$ );
13:    end if
14:  end for
15:  if  $s \in F_{\mathcal{B}}$  then
16:    DFS2( $s$ );
17:  end if
18: end procedure DFS1
19:
20: procedure DFS2( $s$ ) do
21:   Insert  $s$  in  $STK_2$ ;
22:   for all  $s'$  such that  $s \rightarrow s'$  do
23:     if  $s' \in STK_1$  then
24:       return counterexample;
25:     else if  $s' \notin STK_2$  then
26:       DFS2( $s'$ );
27:     end if
28:   end for
29: end procedure DFS2

```

It calls for every initial state the procedure DFS1 that, in turn, traverses in depth-first manner the graph, putting every visited state in the stack STK_1 till it has to backtrack on an accept state. Then, DFS1 calls DFS2 which is in charge of finding out if the current accept state s is contained in a cycle. Again, DFS2 explores the state space in depth-first manner till it finds a successor of the accept state, say s' , that has been already visited by DFS1, i.e., $s' \in STK_1$. If this is the case, a cycle has been found and the procedure terminates successfully. The counterexample is given by the sequence of states in the stack STK_1 up to s that constitutes the prefix of the SCC. Finally, the SCC itself is constructed by taking the states in STK_2 that form a path from s to s' and closing the cycle with the states in STK_1 that form a path from s' to s .

2.1.6 The tableau approach to model checking

An alternative algorithm for model checking LTL is based on the construction of a *tableau graph*. This method was introduced by Lichtenstein and Pnueli in [77]. In general, the construction of the tableau graph depends on the formula ϕ we want to model check and on the Kripke structure \mathcal{K} . The main property of the tableau graph is the following:

a model for the formula ϕ can be extracted from it if and only if the formula is satisfiable in \mathcal{K} .

Moreover, another important property of the tableau is that it is a finite structure which embeds all possible models of the formula. We briefly summarise the construction of the tableau graph for the formula ϕ and the Kripke structure \mathcal{K} .

The *closure* of ϕ , denoted by $CL(\phi)$, is the set of formulae whose truth value may influence the truth value of ϕ itself.

Definition 2.1.8. Let ϕ be an formula. The *closure* of ϕ , $CL(\phi)$, is the smallest set of formulae (identifying $\neg\neg\psi$ with ψ) such that:

- $\phi, \text{tt}, \text{ff} \in CL(\phi)$;
- $\neg\psi \in CL(\phi)$ iff $\psi \in CL(\phi)$;
- if $\psi_1 \vee \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2 \in CL(\phi)$;
- if $X\psi \in CL(\phi)$ then $\psi \in CL(\phi)$;
- if $\neg X\psi \in CL(\phi)$ then $X\neg\psi \in CL(\phi)$;
- if $\psi_1 \cup \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2, X(\psi_1 \cup \psi_2) \in CL(\phi)$.

The tableau graph for \mathcal{K} and ϕ denoted by $G_{\mathcal{K}}(\phi)$ is given by a set of vertexes $A_{\mathcal{K}}(\phi)$ and a set of arcs $\rightarrow \subseteq A_{\mathcal{K}}(\phi) \times A_{\mathcal{K}}(\phi)$. The vertexes of the tableau, called *atoms*, are composed by states of \mathcal{K} decorated with a maximal set of consistent formulae compatible with the labelling of \mathcal{K} , i.e. the interpretation function L .

Definition 2.1.9. Given a Kripke structure $\mathcal{K} = \langle S, I, \rightarrow, L \rangle$, and a formula ϕ , an *atom* is a pair $A = (s, D)$ with $s \in S$ and $D \subseteq CL(\phi) \cup AP$ such that:

- if $p \in AP$, then $p \in D$ iff $p \in L(s)$;
- if $\psi \in CL(\phi)$, then $\psi \in D$ iff $\neg\psi \notin D$;
- if $\psi_1 \vee \psi_2 \in CL(\phi)$ then $\psi_1 \vee \psi_2 \in D$ iff $\psi_i \in D$ for $i = 1$ or $i = 2$;
- if $\neg X\psi \in CL(\phi)$, then $\neg X\psi \in D$ iff $X\neg\psi \in D$;
- if $\psi_1 \cup \psi_2 \in CL(\phi)$, then $\psi_1 \cup \psi_2 \in D$ iff either $\psi_2 \in D$, or both $\psi_1 \in D$ and $X(\psi_1 \cup \psi_2) \in D$.

We denote the first and the second components of an atom $A = (s, D)$ by s_A and D_A , respectively.

To complete the construction of the graph $G_{\mathcal{K}}(\phi)$, we must define arcs between vertexes. Those take into account the semantics of the formulae contained in the atoms and the transitions of the Kripke structure.

Definition 2.1.10. The *tableau graph* $G_{\mathcal{K}}(\phi)$ for \mathcal{K} and ϕ consists of vertexes $A_{\mathcal{K}}(\phi)$ and arcs $\rightarrow \subseteq A_{\mathcal{K}}(\phi) \times A_{\mathcal{K}}(\phi)$ determined by:

$$(s, D) \rightarrow (s', D') \Leftrightarrow \begin{cases} s \rightarrow s' \text{ and} \\ \forall X\psi \in CL(\phi), (X\psi \in D \Leftrightarrow \psi \in D') \end{cases}$$

Hence the resulting graph will mimic the behaviour of \mathcal{K} as long as the formulae involving temporal operators are satisfied. For the definition of the arcs, it is sufficient to require the consistency of the next operator X .

A *path* π of $G_{\mathcal{K}}(\phi)$ is an infinite sequence of atoms $\pi = A_0 A_1 A_2 \cdots$ in G such that

- $s_{A_i} \rightarrow s_{A_{i+1}}$ for all $i \geq 0$;
- if $\psi_1 \cup \psi_2 \in D_{A_i}$ for some atom $i \geq 0$, then there exists an atom A_j with $j \geq i$ such that $\psi_2 \in D_{A_j}$.

Thus a path² corresponds to a run of \mathcal{K} and while ensuring the satisfaction of until formulae. Let $\sigma = s_{A_0} s_{A_1} s_{A_2} \cdots$, the path π *fulfils* ϕ if

$$\sigma \models \phi.$$

The presence of a formula in an atom A_i belonging to a fulfilling path π is related to the satisfaction of that formula in the model underling the path from state i on, i.e., π^i . More precisely we have the following fact whose proof is given in [77].

Proposition 2.1.11 (Lichtenstein and Pnueli [77]).

- $\pi = A_0 A_1 A_2 \cdots$ fulfils ϕ if and only if $\phi \in D_{A_0}$.
- ϕ is satisfiable if and only if there exists a path π fulfilling it.

By the previous proposition we can conclude that in order to check the satisfiability of a formula ϕ in \mathcal{K} it is sufficient to look for a fulfilling path in the tableau $G_{\mathcal{K}}(\phi)$. However, in a graph there may be infinitely many different paths, therefore Proposition 2.1.11 still does not provide us with a computable method.

A strongly connected subgraph (SCS) $G' \subseteq G_{\mathcal{K}}(\phi)$ is called *self-fulfilling* if for every $A \in G'$ such that $\psi_1 \cup \psi_2 \in A$ there exists $B \in G'$ such that $\psi_2 \in D_B$. Let $Inf(\pi)$ denote the set of atoms that appear infinitely often in the path π .

²By some unfortunate coincidence, the term “path” is used for two different mathematical objects, i.e., a sequence of states in a Kripke structure and a sequence of atoms in the tableau graph. In the following we will specify which object we are referring to, unless it is clear from the context.

Proposition 2.1.12 (Lichtenstein and Pnueli [77]).

- If π is fulfilling then $Inf(\pi)$ is a self-fulfilling SCS of $G_{\mathcal{K}}(\phi)$.
- for every self-fulfilling SCS $G' \subseteq G_{\mathcal{K}}(\phi)$ there exists a fulfilling path π such that $Inf(\pi) = G'$.

An immediate consequence of this proposition is that ϕ is satisfiable if and only if there exists an initial atom A in $G_{\mathcal{K}}(\phi)$ such that $\phi \in D_A$ and a self-fulfilling SCS is reachable from A . Indeed this proposition can be used to define a mechanical way to verify the satisfaction of the formula ϕ since there are only a finite number of SCS in $G_{\mathcal{K}}(\phi)$.

The algorithm presented in this section has a complexity linear in the size of \mathcal{K} and exponential in the size of ϕ (cf. [77]).

2.1.7 Automata theoretic versus tableau approach

After having summarised the automata theoretic approach in Section 2.1.5 and the tableau method in Section 2.1.6, it is natural to ask which method is more suitable for our study. The argument in favour of the automata-theoretic approach is essentially that it can be performed *on-the-fly* [51] meaning that it is not necessary to construct the complete state space of the Büchi automaton \mathcal{B}_{sys} modelling the system. Instead, states of \mathcal{B}_{sys} are generated only when it is needed during the construction of the product automaton. This means that it is not always necessary to construct the complete \mathcal{B}_{sys} since we can find a counterexample before, and thus we can stop the procedure at that point. As a direct consequence, the automata theoretic method may help to alleviate the problem of *state-space explosion*. This occurs when in order to model the system accurately, the automaton \mathcal{B}_{sys} needs a number of states that is too large (normally exponential in the number of components of the system) and exceeds the computer memory. Currently this is the main problem of model checking. In contrast, in order to apply the tableau method described in Section 2.1.6 we need the complete Kripke structure modelling the system which (again) can be rather large.

Of course, the automata-theoretic approach (for LTL) relies on the feasibility of the construction of the Büchi automaton corresponding to the negation of the LTL formula we want to verify. However, if we change focus, and from LTL we move to some other specification language, the situation may be different. And in fact, this is what happens, for example, with the logic $\mathcal{A}\ell\text{LTL}$ that we study in Chapter 4. For this logic, the construction of the automaton for a given formula is most likely undecidable³ due to the very dynamic nature of its models. Being able to construct the automaton for every formula of the logic corresponds to solve the satisfiability problem (SAT), although we are mainly interested in the model checking problem (MC) (cf. Section 2.1.7).

³The study we have carried out in the past strongly suggest the undecidability of the problem, although we do not have any proof of this conjecture.

This consideration has led us to consider the tableau method. In that approach, the only Kripke structure involved is the one corresponding to the behaviour of system (that is given). Furthermore, this given Kripke structure guides us in the construction of the tableau graph. Hence, in this sense, the method appears to be more promising, since somehow, it overcomes the construction of the automaton corresponding to the formula. In the subsequent chapters we will use the tableau method and see that it can be effectively exploited to develop model-checking algorithms for logics with a high degree of dynamism such as \mathcal{ALLTL} .

2.2 History-Dependent Automata

History-dependent formalisms are those whose actions performed in a particular state may depend on actions performed in previous states. A typical history-dependent formalism is the π -calculus [82] where, for example, channel names are created in some state and can be referred to in some other successive state. Other history-dependent formalisms are CCS with localities [11] and Petri Nets. In general, the semantics of history-dependent formalisms is given in terms of labelled transition systems. Ordinary automata, such as LFSA given by Definition 2.1.4, present some limitations in dealing with such kinds of calculi. Even very simple actions may produce infinite state spaces. This is clearly problematic for the application of formal techniques such as model checking which, as we have seen, mostly require the inspection of a finite-state model. Montanari and Pistore introduced an extension of ordinary automata, called History-dependent (HD) automata [83, 84, 90] that are meant to represent adequate models for history-dependent formalisms, overcoming therefore the limitations of LFSA. Notice that object-based systems have a clear history-dependent behaviour since, for example, objects are created and can be referenced to in the future. Therefore, the features of HD-automata can be exploited to model the birth and death behaviour of objects.

We quickly introduce a (specialisation of the) definition of HD-automata that differs slightly from the original definition of HD-automaton given in [84] but easier fits towards our needs⁴. Let \mathfrak{N} be a countable set of global names.

Definition 2.2.1. A HD-automaton \mathcal{A} is a tuple $\langle Q, N, \rightarrow, q_0 \rangle$ where:

- Q is a (possibly infinite) set of states;
- $N : Q \rightarrow 2^{\mathfrak{N}}$ is a function that associates to each state $q \in Q$ a finite set N_q of names;
- $\rightarrow \subseteq Q \times (\mathfrak{N} \rightarrow \mathfrak{N}) \times Q$, such that for $q \rightarrow_{\lambda} q'$, we have $\lambda : N_q \rightarrow N_{q'}$ injective;

⁴This is only done for the sake of simplicity in the presentation since the definitions of our formalisms (cf. Chapters 4 and 5) resemble the one given here.

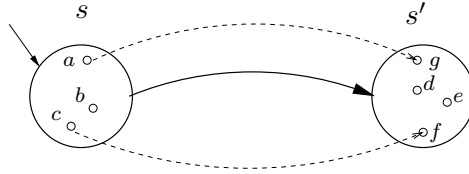


Figure 2.2: A simple HD-automaton.

- $q_0 \in Q$ is an initial state;

States of an HD-automaton are equipped with a set of *local* names that can be created dynamically. The transitions are equipped by a partial injective mapping λ from names of the source state to names of the target state. Since names do not have a global identity, a single state of an HD-automaton can be seen as an abstraction of the set of all states that differ only by bijective re-namings of names. The correspondence between the local names of two successive states is ensured by the mapping λ attached to the transitions.

The difference between Definition 2.2.1 and the original definition of HD-automaton given in [84] is essentially that in the latter, not only states are equipped with names but also transitions and labels (in the transitions). The correspondence between names of the source state with the those of the target state is split in two mappings (instead of one like in Definition 2.2.1): the first mapping goes from the names of the source state to the names of the transitions and second mapping from the name of the target state to the names of the transition. We will not further consider the difference between Definition 2.2.1 and that given in [84].

Names appearing in a state (of an HD-automaton) are considered to be currently in use, i.e., *alive*. The mapping attached to the transition is partial, giving the possibility to distinguish three categories of names:

- names that are mapped from the source to the target state are preserved in the transition;
- names that appear in the source state but not in the target state are deallocated;
- names that appear only in the target state are created.

Example 2.2.2. A simple HD-automaton is shown in Figure 2.2. We have $Q = \{s, s'\}$; the initial state (q_0 in the definition) is s . Names are depicted by small circles, therefore we have: $N(s) = \{a, b, c\}$, $N(s') = \{d, e, f, g\}$. The mapping λ is represented by dashed lines. By of λ , b is deallocated when the transition is fired, whereas the name a in s is renamed into g in s' and c is renamed into f . It is important to stress the local nature of names, in fact if we assume that a represents a name of an object, then the same object after the transition is called g . Names d and e are new in s' . \square

The local nature of names in HD-automata is helpful because the states can be identified up to renaming, i.e., the actual identity of names does not matter. This is beneficial when it comes to give the semantics to history-dependent formalisms such as the π -calculus. Without entering in the details (for which we refer the interested reader to [84]), it is enough to mention that the semantics of π is usually given in terms of standard labelled-transition systems. Furthermore, in the π -calculus, an agent in every state uses a certain set of names. Also, there exists an *input* action in which the agent receives a name via a channel that is bound to a variable, say x . However, an input action generates a *infinite* number of transitions because x can be bounded with any possible name in \mathfrak{N} not used by the agent (and there are infinitely many of such names). Thus, even a really simple agent (with only an input action) may generate an infinite transition system. The important point there is that it does not matter which name is bound to x , every choice is equivalent, but this fact is not exploited in a standard transition systems semantics. With local names instead, since we can identify states up to renaming of names, the infinite branching transitions can be collapsed into a single one by, for example, choosing a representative name to bind to x . Another advantage is that it is possible to obtain a minimal automaton up to bisimulation. Finally, it is interesting to note that a large class of *finitary* π -calculus agents⁵ correspond to finite-state HD-automata that in turn can be translated into finite ordinary automata, in such a way that bisimilar HD-automata correspond to bisimilar ordinary automata. Dealing with the latter is convenient since standard algorithms may be applied.

In Section 2.3.2, we will see that object-based systems are highly dynamic also because of the allocation and deallocation of objects. Here we have learnt that HD-automata are a formalism suited to model (in a compact way) the allocation and deallocation of names. In studying models oriented towards the verification of dynamic properties of object-based systems it therefore becomes natural to look at HD-automata as a starting point. Indeed, in Chapter 4 and 5 we will define two classes of Büchi automata — very much inspired by HD-automata — that also exploit the advantage of having local names and remapping in order to obtain more compact models.

2.3 Object-based systems

In this thesis, we confine ourselves to *object-based* systems, i.e., object-oriented systems in which inheritance and sub-typing are not considered⁶. Object-based systems are composed by *objects* that run concurrently and *communicate* with

⁵For the reader familiar with the π -calculus, finitary agents are those where parallel composition does not occur in the bodies of their recursive definitions.

⁶In other classifications (e.g. [1]), object-based languages do not have classes in contrast to class-based languages. We will not use this classification.

each other. Objects are dynamic, can be *created* in arbitrary number and can be *deallocated* during the computation.

Objects. At the conceptual level, an object can be seen as an *entity* that has an *identity*, a *state* and a *behaviour*. This definition has originally been given by Booch [8].

The *state* of the object contains internal data that can only be accessed from the outside by invoking one of the object's methods. The invocation is done by sending a message to the object. Data are usually called *attributes* and they have a value of a certain type. Depending on the definition of the object, some attributes may be modified while others may not.

The *behaviour* of the object is the reaction that the object has after having been stimulated by a message from the outside world. The way an object behaves is in general dependent from its current internal state. The outside world is constituted by other objects running concurrently.

The *identity* of the object is distinct from its state: the value of its attributes may change during the computation although the object's identity remains fixed. In general an object is created at a certain point in time, and it may die successively. However, between the moment in which the object is born until it dies, the object is continuously *alive* and is identified by its identity.

In later chapters in this thesis we will deeply investigate the nature of object identity. In our models, we will clearly distinguish between the identity of an object and its state. We will also give identity to a method invocation. A justification will become clear later.

Classes. On the static level, the corresponding notion of object is that of a class. A *class* is a template for the creation of its instances, i.e., its objects. A class specifies the state of its objects by describing the attribute types, and the objects behaviour by the definition of a set of methods. A *method* is a fragment of code that implements an operation. The implementation is hidden, the only thing the outside world knows is that the objects belonging to the class provide that operation. Methods of a class are executed by instances of the same class. The set of class operations that can be invoked from the outside is known as the *interface* of the class. Interfaces are useful to increase the abstraction of the classes since methods and attributes not included in any interface are hidden to the outside⁷. We refer to the object executing a method as the *owner* of the method. A special keyword that may be used in the code of the method is `self`. At run-time, it refers to the identity of the owner⁸. By `self` a method can access the attributes (i.e., the state) of its owner.

⁷Here we use the term “abstraction” in a different way than the rest of the thesis, where in general abstraction will describe some technique to reduce the state space of a model.

⁸Some languages such as Java or C++ prefer the notation `this` instead of `self`.

2.3.1 The Unified Modelling Language

The Unified Modelling Language (UML) [9, 98] is the standard modelling languages accepted by the Object Management Group (OMG) in 1997. It is the result of a merging process of different notations previously existing.

The UML provides the designer with several types of diagrams that concentrate on a special part of the system, amongst others:

- the *use case diagram* concentrates on the interaction between the user of the system (called actors) and the system itself;
- the *class diagram* focuses on the relation between classes of the system.
- the *sequence diagram* describes the behaviour of the system by showing the interaction of objects through the messages they interchange.
- the *object diagram* models a single state (of a computation) of the system (sometimes called *snapshot*). It contains the values of the object attributes as well as the references between objects; the latter take the form of links among them.
- the *state chart diagram* describes the life cycle of an object by a sequence of states of a state machine associated with the object.

Since in this thesis we will be concerned only with class diagrams, we introduce some of their basic concepts.

Class diagrams. In UML, the static structure of classes in the system and their associations are described by class diagrams that illustrate for each class

- the attributes with their type, and
- the methods with their formal parameters.

A simple example class diagram (adopted from [111]) is depicted in Figure 2.3. Boxes represent classes. Three parts, delineated by a line, can be distinguished in a box. The class name is indicated in the upper part, thus in our system we have three classes: Hotel, Room, and Guest. The attributes of the class together with their types are reported in the middle part of the box. For example, the class Hotel has only two attributes *numOfFloors* and *numOfRooms*, whereas both Room and Guest have three attributes. The attribute name is followed by the attribute type (separated by a colon sign ':'). In the example, both *numOfFloors* and *numOfRooms* have type *Integer*. Finally, the bottom part of a box lists the operations (or methods) of the class. The signature of an operation describes its name, its formal parameters and the return type (if any). The formal parameters are depicted between parentheses indicating also their type. Thus for example, the class Hotel has two operations *checkIn* and *checkOut*, respectively, with only one parameter of type Guest and without return type. Classes Room and Guest do not have any operation.

Boxes are interconnected by lines denoting *associations*. They represent the relationship between classes. Each direction of an association has an optional name (called *role name*) and multiplicity giving information on the number of elements related by the association. For instance, the multiplicity 1..* states that a Hotel has at least one room, whereas on the association relating Hotel and Guest, * says that the Hotel can have a number of guests greater or equal to zero (i.e., the hotel can be empty). If the multiplicity is not specified it is intended to be one.

Note that class diagrams only address the static aspects of the system, not its dynamic (i.e., process) aspects. The latter aspects are described by other diagrams such as UML state-charts.

Navigation. An important factor is that associations between classes in UML diagrams can be traversed. Thus, they constitute a path within the model that allows to reach a class starting from another class. This phenomenon is referred to as *navigation* (or navigability) and allows to refer to attributes and methods of a (collection of) object(s) in the system. As we will see in Section 3.4, in the notation used by the Object Constraint Language [110], navigations are expressed by indicating the role name attached to the association we want to traverse. For instance, for an object of class Hotel denoted by the variable *h*, the expression *h.guests* refers to the path between class Hotel to the class Guest using the association with role name *guests*. Navigations are parsed from left to right. The navigation expression *h.guests.guestCode* refers to the collection of guests' code of *h*.

2.3.2 Dynamic allocation and deallocation

Concurrent object-based systems are characterised by a *highly dynamic* behaviour. A first interesting source for this dynamic behaviour comes from the lives of objects. During the computation, objects are *created* by some creation mechanism provided by the programming language that must be invoked explicitly. For example, for the Hotel example in Java we would write

```
new Hotel()
```

to create an instance of the class Hotel. Here, `Hotel()` represents a *constructor*, that is, a procedure specifying in detail how to build the object and how to initialise its attributes. Several constructors can be defined for the same class according to different possibilities and/or needs to create objects.

Moreover, during the computation some object may *die* either because of an explicit command given by the programmer (such as `delete` in C++) or because of some system routine (e.g. garbage collection). In any case, an object-based computation does not have a static pre-allocated number of objects that will remain fixed till the end. This model of computation is therefore different from the more traditional programming languages (e.g. Pascal or C)

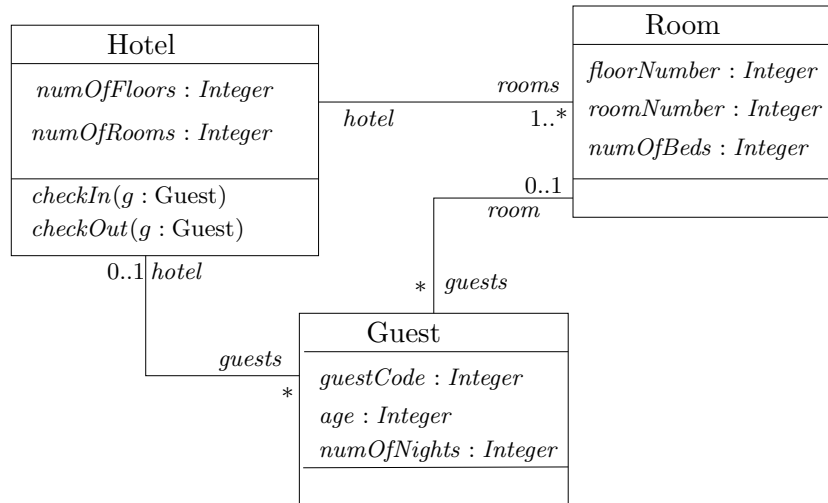


Figure 2.3: The Hotel class diagram.

where most of the resources are allocated statically according to the declarations contained in the program. Figure 2.4 depicts a sequence of three run-time states of an object-based system. The internal structure — that sometimes will be referred to as (*object-*) *heap* — shows the alive objects (and their references). In the state s_1 , six objects are alive: o_1 through o_6 . In the successor state s_2 , the object o_5 has died, whereas o_7, o_8 have been created. Finally, in state s_3 , object o_1 dies. We can paraphrase the previous description moving the focus towards the lives of the single heap-allocated objects⁹ as follows:

In state s_1 , the objects $o_1, o_2, o_3, o_4, o_5, o_6$ are *alive*. In state s_2 object o_5 is *dead* and $o_1, o_2, o_3, o_4, o_6, o_7, o_8$ are *alive*, from which o_7 and o_8 are *new* (i.e., fresh) whereas o_1, o_2, o_3, o_4, o_6 are *old*. In state s_3 the object o_1 is *dead* and $o_2, o_3, o_4, o_6, o_7, o_8$ are *alive* and at the same time *old*.

This second description gives us an important hint: the substantial properties that allow us to capture the dynamism of the system w.r.t. the lives of objects are given by the following four adjectives: new, old, alive, dead. This will be used later on. Furthermore, it is clear that “old” is synonym of *alive but non-new* as well as “dead” is synonym of *non-alive*, therefore in principle we can focus on only two of this object status¹⁰.

⁹This is essentially the dynamic evolution of the system we want to capture

¹⁰Provided, of course, that we can use an operator expressing negation.

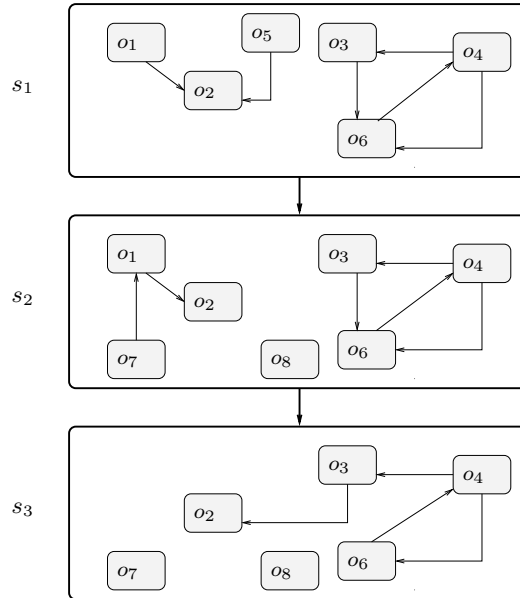


Figure 2.4: Sequence of states in a concurrent object-based system.

The models that we aim to develop must be as general as possible. In principle, there is no reason to admit in our models only objects that are created during the computation and that will die at some point in the future. In the history of the humanity many models have been developed where entities were supposed to have been always alive, or that would not die. It is enough to think about the several religions or ancient mythology. Thus — beside (standard) objects that are born in some state of the computation and die in some other state — we can very well have *immortal* objects that do not die at all during the computation (cf. Java real-time); or *eternal* objects that are alive but they are neither born nor die in any state of the computation. Modelling the states of an object-based computation as described in Figure 2.4 is quite expressive and general. Unfortunately, the model becomes somehow naively idyllic once we try to apply it for model-checking technology. There are two problems related to the object model presented above. First of all, as we have previously remarked, the model-checking motto is brute force: the state-space must be completely covered in order to fully apply this technique. Secondly, computer memory is finite. As a consequence, we cannot just let the number of object grow in a naive way. Their number may become *unbounded* and therefore the state space *infinite*. Bounding the number of objects that may be created seems to be a rather strong requirement on the model (and on the program). Moreover, techniques such as garbage collection are in general not enough to maintain the number of objects bounded. Different strategies

are needed.

On object-heap abstraction. Abstraction can be employed. Unfortunately, there do not exist general recipes for the definition of “good” object-heap abstractions, therefore the task is not trivial. Many factors may play a role: the kind of property we are interested in; the kind of classes that compose our system; and so on. Sometimes an abstraction may be too coarse, and therefore not informative enough; on the contrary it might be too fine, keeping far too many details making it intractable; some other abstractions can be unsound — therefore providing us with wrong information. Theories such as abstract interpretation [36, 37] help in proving the soundness of an abstraction; however, to define the abstraction itself is almost a complete manual exercise. In this thesis we will also investigate some possible abstractions that may render a finite state space — even if we do not force any bound on the number of objects (or threads) that can be created — but still providing us with the possibility to deal with some interesting properties about the birth and death of objects (and threads).

2.3.3 Dynamic references

Associations between classes in the UML class diagram correspond at the run-time level to the concept of *references* between objects. References are attributes whose type is not a basic one (e.g., integer, boolean etc.) but instead another class. In Figure 2.4, references are represented by direct arrows connecting objects. For instance, in state s_1 , the object o_1 has a reference to o_2 . A reference can be either null, i.e., it is not pointing to any object or it can be “attached” to another object. The reference mechanism gives the possibility to combine objects and therefore to create rather complicated structures. Furthermore, programming languages provide the programmer with the possibility to manipulated references at run-time. Simple statements like

$$x := y \tag{2.6}$$

where x, y have type reference, allow to detach existing references and to attach new ones, therefore changing the heap-object topology. From state to state, the evolution of this heap topology produces the second form of interesting dynamic behaviour in the object model which we will try to investigate in Chapter 5.

For example, the variable x loses the reference to the object it is pointing to before the assignment (2.6) and the reference is substituted by y ’s reference. After (2.6) x and y have a pointer to the same run-time object. This well-known phenomenon is called *dynamic aliasing*. The consequences of this kind of dynamic evolution of the heap topology are rather interesting. From state to state non-trivial interdependencies even among apparently non-related objects may be introduced. The unpleasant effect is that, in general, the execution of

an operation may influence objects not even mentioned in the expression. This has been called by some authors the *complexity of pointer swing* [67, 94]. The problem seems very dangerous since it easily occurs and at the same time it is difficult to debug.

Another very common problem due to the dynamic nature of references is the possibility to dereference, at any moment, a null reference and, as a consequence, an *exception* is thrown by the run-time environment and the computation stops (as in Java). The ideal solution would be that the compiler could statically detect any attempt to dereference a null pointer. In reality, this cannot be done since such problems lie beyond the range of capabilities of conventional type systems. Therefore, so far, the programmer is the only one responsible for ensuring that his program “does not go wrong” in this sense.

In object models where references are taken into account, an object is called *reachable* if there exists a path of references starting from a program element (such as a program variable or a formal parameter of a method) and ending at the object. Usually, only reachable objects are seen as useful since they can be manipulated whereas the unreachable ones are considered to be “garbage” because they cannot be used anymore. An issue somehow related to the life of objects, and at the same time, to the manipulation of references is what to do in case an object becomes unreachable. In general, three different approaches can be taken [81]:

- the *casual* approach follows the philosophy that unreachable objects are left in memory for ever;
- the *manual deletion* approach gives the programmer the possibility to delete explicitly the objects that are considered garbage through either some algorithm or by explicit programming constructs;
- the *automatic garbage collection* approach burdens the run-time environment with the responsibility to dispose of the garbage, freeing therefore the corresponding memory.

Here, we do not argue which approach is the best. In general, the third one is very well accepted, although in some cases, the other two are also used. For example, C++ and Objective-C provide the programmer with the possibility to dispose objects explicitly. In some real-time applications, it is preferred to allocate all the objects in the beginning of the computation and let them (or part of them) live for ever even if they become garbage (cf. immortal objects in Java real-time [49]). This avoids the degree of uncertainty in the response of the system induced by the execution of the garbage collector routines.

Since in Chapter 4 we are interested in studying properties of the system in relation to the allocation and deallocation of objects we will try to have a model as general as possible. Therefore, we will not make a real choice. This allows us to use the model also to reason about garbage collector routines. Instead in Chapter 5 where we will define models to reason about references we will use automatic garbage collection in order to simplify the mathematical machinery.

3

A logic for object-based systems

3.1 Introduction

This chapter presents a temporal logic, referred to as BOTL (Object-Based Temporal Logic), that is aimed at specifying *static* and *dynamic properties* of object-based systems. The dynamic properties are related to the behaviour of the system when time evolves, while the static properties refer to the relations between syntactical entities such as classes. The logic is an object-based extension of the linear temporal logic LTL [91], a formalism for which efficient model checking algorithms and tools do exist. The object-based ingredients in our logic are largely inspired by the Object Constraint Language (OCL) [88, 110, 111], that is part of the Unified Modelling Language (UML) [9, 98]. OCL allows expressing static properties over class diagrams in a textual way. The precise relationship with OCL is studied by defining a mapping of a large fragment of OCL onto BOTL.

The semantics of BOTL is defined in terms of a *general* operational model that is aimed to be applicable to a rather wide range of object-oriented programming languages. The operational model is a (generalised) Büchi automaton, in which states are equipped with information concerning the status of objects and method invocations. The interpretation of BOTL in terms of these automata is defined in a formal, rigorous way. Such a formal approach is in our opinion indispensable for the construction of reliable software tools such as

model checkers. Besides, the semantics of BOTL together with the aforementioned translation of OCL provides a *formal semantics* of OCL. This approach resolves several ambiguities and unclarities in OCL that have been reported in the literature, e.g., [31, 53, 55]. A significant fragment of OCL is covered including, amongst others, invariants, pre- and postconditions, navigations and iterations.

This chapter is organised as follows: Section 3.2 defines the BOTL syntax and data types. Section 3.3 defines the operational model in which the logic is interpreted, as well as the formal semantics of BOTL. Section 3.4 gives a short introduction to OCL as a primer for Section 3.5 where the mapping from OCL to BOTL is defined. Finally, in Section 3.6 we discuss some related work, focusing in particular on the Bandera Specification Language.

3.2 Syntax of BOTL

Before introducing the syntax of BOTL we present the data types over which logical variables may range. In the following we assume given:

- a (countable) set $VNAME$ of *variable names*;
- a (countable) set $MNAME$ of *method names*, ranged over by M ;
- a (countable) set of *class names* $CNAME$, ranged over by C .

3.2.1 Data types and values

BOTL expressions rely on a language $TYPE$ of data types, defined by the following grammar:

$$\tau (\in TYPE) ::= \text{void} \mid \text{nat} \mid \text{bool} \mid \tau \text{ list} \mid C \text{ ref} \mid C.M \text{ ref}$$

where $C \in CNAME$ and $M \in MNAME$ are arbitrary. The types have the following intuitions:

- **void** is the unit type; it only has the trivial value $()$.
- **nat** is the type of natural numbers.
- **bool** is the type of boolean values **tt** (true) and **ff** (false).
- τ **list** denotes the type of lists of τ , with elements $[]$ (the empty list) and $h :: w$ (for the list with head element h and tail w). For the sake of readability, we will often write lists as comma-separated sequences enclosed by square brackets; e.g., $1::2 :: []$ is written $[1, 2]$, whereas $[[1], [2]]$ denotes $(1 :: []) :: (2 :: []) :: []$.
- C **ref** denotes the type of objects of class C .

- $C.M\text{ref}$ denotes the type of method occurrences (discussed in more detail below) of the method M of class C .

Let us specify the data values of these types more precisely. Among others we will use (references to) *objects* and *events* as data values; the latter correspond to *method occurrences*, i.e., invocations of a given method of a given object. For this purpose, we introduce the following sets (for all $C \in \text{CNAME}$ and $M \in \text{MNAME}$):

$$\begin{aligned} \text{OID}^C &= \{C\} \times \mathbb{N} \\ \text{EVT}^{C,M} &= \text{OID}^C \times \{M\} \times \mathbb{N} . \end{aligned}$$

Thus, object identities $o \in \text{OID}^C$ correspond simply to numbered instances of the class C , whereas events $(o, M, j) \in \text{EVT}^{C,M}$ are numbered instances of the method name M , together with an explicit association to the object $o \in \text{OID}^C$ executing the method. We also use

$$\begin{aligned} \text{OID} &= \bigcup_C \text{OID}^C \\ \text{EVT} &= \bigcup_C \bigcup_M \text{EVT}^{C,M} \end{aligned}$$

ranged over by o and ζ respectively. OID and EVT are the universe of object ids and events, respectively.

Example 3.2.1. Consider the Hotel class diagram of Figure 2.3. The following are instances of the class Hotel:

$$(\text{Hotel}, 1) \quad (\text{Hotel}, 2) \quad (\text{Hotel}, 31) \quad (\text{Hotel}, 127) \quad \dots$$

The following are events related to the method *checkIn*:

$$\begin{aligned} &((\text{Hotel}, 1), \text{checkIn}, 1) \quad ((\text{Hotel}, 1), \text{checkIn}, 2) \\ &((\text{Hotel}, 31), \text{checkIn}, 1) \quad ((\text{Hotel}, 127), \text{checkIn}, 3) \quad \dots \end{aligned}$$

Note that the first two events represent different executions of method *checkIn* performed by the same object. \square

The combined universe of values will be denoted by VAL ; the set of values of a given type $\tau \in \text{TYPE}$ is denoted by VAL^τ . We define:

$$\begin{aligned} \text{VAL}^{\text{void}} &= \{()\} \\ \text{VAL}^{\text{nat}} &= \mathbb{N} \\ \text{VAL}^{\text{bool}} &= \{\text{ff}, \text{tt}\} \\ \text{VAL}^{\tau \text{ list}} &= \{[]\} \cup \{h :: w \mid h \in \text{VAL}^\tau, w \in \text{VAL}^{\tau \text{ list}}\} \\ \text{VAL}^{C \text{ ref}} &= \{\text{null}\} \cup \text{OID}^C \\ \text{VAL}^{C.M \text{ ref}} &= \text{EVT}^{C,M} . \end{aligned}$$

There exists a large number of standard boolean, arithmetic and list operations over these values, which we will use when convenient, without introducing them formally. As an example consider:

- $+$: $\text{VAL}^{\text{nat}} \times \text{VAL}^{\text{nat}} \rightarrow \text{VAL}^{\text{nat}}$ is the standard sum on natural numbers.
- sort : $\text{VAL}^{\tau \text{ list}} \rightarrow \text{VAL}^{\tau \text{ list}}$ orders a given list of values of type τ .
- flat : $\text{VAL}^{\tau \text{ list list}} \rightarrow \text{VAL}^{\tau \text{ list}}$ flattens nested lists.

Finally, there is a special element $\perp \notin \text{VAL}$ that is used to model the “undefined” value: we write $\text{VAL}_{\perp} = \text{VAL} \cup \{\perp\}$. All operations are extended to \perp by requiring them to be *strict* (meaning that if any operand equals \perp , the entire expression equals \perp). For instance, for lists we have $\perp :: w = \perp$ and $h :: \perp = \perp$.

3.2.2 Syntax of BOTL

The syntax of BOTL is built up from two kinds of terms: *static expressions* S_{BOTL} (for a large part inspired by OCL, see Section 3.4) and *temporal formulae* T_{BOTL} (largely taken from LTL). We assume a set of *logical variables* LVAR . BOTL terms are defined by the grammar:

$$\begin{aligned} \xi (\in S_{\text{BOTL}}) & ::= x \mid \xi.a \mid \xi.\text{owner} \mid \xi.\text{return} \mid \xi \text{ new} \mid \xi \text{ alive} \mid \omega(\xi, \dots, \xi) \\ & \quad \mid \text{with } x_1 \in \xi \text{ from } x_2 := \xi \text{ do } x_2 := \xi \\ \phi (\in T_{\text{BOTL}}) & ::= \xi \mid \neg\phi \mid \phi \vee \phi \mid \exists x \in \tau : \phi \mid \text{X}\phi \mid \phi \text{U}\phi \end{aligned}$$

where $\tau \in \text{TYPE}$, $a \in \text{VNAME}$ and $x \in \text{LVAR}$. Apart from this context-free grammar, we implicitly rely on a context-sensitive *type system*, with type judgements of the form $\xi \in \tau$, to ensure type correctness of the expressions; its definition is outside the scope of this thesis.

Before defining the formal semantics in the next section, we give an informal explanation of the BOTL constructs.

Static expressions.

- $x \in \text{LVAR}$ is a variable bound to a value elsewhere in the expression or formula;
- $\xi.a$ stands for *attribute/parameter navigation*. The sub-expression ξ provides either a reference to an object with an attribute named a or to a method occurrence with a formal parameter named a ; the navigation expression¹ denotes the value of that attribute/parameter. Navigation is extended naturally to the case where ξ is a *list* of references; the result of $\xi.a$ is then the list of $_a$ - navigations from the elements of ξ .
- $\xi.\text{owner}$ denotes the object executing the method ξ .

¹See Section 2.3.3 for an introduction to the concept of navigation.

- $\xi.\text{return}$ denotes the return value of the method denoted by ξ (in case the method has indeed returned a value, otherwise the result of the expression is undefined; see below).
- $\xi \text{ new}$ expresses that the object or the method occurrence denoted by ξ is fresh in the current state, i.e., it did not exist in any of the previous states. Typically, an object is new, just after it has been created as result of the execution of some mechanism for object creation provided by the programming language at hand; A method occurrence is new at the moment of the invocation of the method.
- $\xi \text{ alive}$ expresses that ξ denotes an object or method occurrence that is currently *alive*. An object becomes alive when it is created and remains alive until it is deallocated (e.g. by garbage collection), whereas a method occurrence becomes alive when it is invoked and it is deallocated after it has returned a value. This is made more precise in the semantics model; see Section 3.3.1.
- $\omega(\xi_1, \dots, \xi_n)$ ($n \geq 0$) denotes an application of the n -ary operator ω . Thus, ω is a syntactic counterpart to the actual boolean, arithmetic and list operations defined over our value domain. Possible values for ω include at least a conditional expression (“if-then-else”) as well as an (overloaded) equality test $=^\tau$ for all $\tau \in \text{TYPE}$ (where the index τ is usually omitted). We will use $\llbracket \omega \rrbracket$ to indicate the underlying operation of which ω is the syntactic representation.
- The with-from-do expression is inspired by the *iterate* feature of OCL —which in turn resembles the *fold* operation of functional programming. The expression binds logical variables and can therefore not be seen as an ordinary operator. Informally, $\text{with } x_1 \in \xi_1 \text{ from } x_2 := \xi_2 \text{ do } x_2 := \xi_3$ has the following semantics: first, x_2 is initialised to ξ_2 ; then ξ_3 is computed repeatedly and its result is assigned to x_2 while x_1 successively takes as its value an element of the sequence ξ_1 . For instance, the expression

$$\text{with } x_1 \in [1, 2, 3] \text{ from } x_2 := 0 \text{ do } x_2 := x_1 + x_2$$

computes the sum of the elements of the list $[1, 2, 3]$ ($= 6$).

Temporal expressions. A temporal expression ϕ is built by the application of classical propositional logic operators (\neg , \vee etc.) and LTL temporal operators (X , U). As we have seen in Section 2.1.2, from the latter it is possible to derive other useful temporal operators. The basic predicates are given by boolean expressions in S_{BOTL} . We briefly summarise the intuition of the temporal operators:

- $X\phi$ expresses that in the next state the formula ϕ holds.

- $\phi U \psi$ expresses that along the path starting from the current state there exists a state in which ψ holds and ϕ holds in every state before. The special case where ϕ equals tt (true) thus stands for the property that there exists a reachable state where ψ holds; this is sometimes denoted $F\psi$ (“eventually ψ ”). The dual of that is denoted $G\psi$ (“globally ψ ”) and it holds if ψ is true in every state along the path starting from the current state.

Quantification. Special attention must be given to the temporal expression involving existential (and, by duality, universal) quantification. The formula $\exists x \in \tau: \phi$ expresses that ϕ must hold for at least one *alive* instance x of the type τ . By convention, for the universes of values given by VAL^{void} , VAL^{nat} , VAL^{bool} , we assume their elements to be *all* alive in every state of the computation. This is justified since they describe static sets of values. Note that VAL^τ is infinite for some $\tau \in \text{TYPE}$.

The situation is different for the special domains $\text{VAL}^{C \text{ref}}$ and $\text{VAL}^{C.M \text{ref}}$. Although by definition they are infinite, in a particular state of the computation there is only a *finite subset* of their elements that have been created, i.e., that are alive. Because of object creation and method invocation, from state to state new instances may be created and destroyed, making therefore the subsets of alive instances of $\text{VAL}^{C \text{ref}}$ and $\text{VAL}^{C.M \text{ref}}$ dynamic. Thus, if we consider the overall computation, the set of alive objects and method invocations may become *unbounded*. This implies that model checking of universally quantified formulae is impossible if we rely on standard techniques. For example, consider the formula:

$$G[\forall x \in \tau: \phi].$$

If the alive elements in VAL^τ grow infinitely often, then, since indeed the state space becomes infinite in order to use exhaustive state space exploration, it is necessary to design sound *abstractions* able to keep the state space finite. In Chapters 4 and 5, — along the lines of HD-automata² — we will see how the use of local identities for objects and method occurrences helps to alleviate the problem. Furthermore, we will introduce two abstractions and study the consequences from the model checking point of view³.

Abbreviations. In examples, we often omit the type τ when it is clear from the context. Moreover, for ξ denoting an element of $C \text{ref}$ or $C.M \text{ref}$ we use the abbreviations in Table 3.1. The predicates $\xi \text{ dead}$ and $\xi \text{ old}$ are very useful and they will be used extensively throughout the dissertation. The last abbreviation is convenient for quantify over alive method occurrences of a given alive object.

²See Section 2.2.

³For the sake of completeness, abstractions must be applied also to VAL^{nat} . However, for this domain, it is customary to restrict quantification to bounded cases; for instance, all integers *smaller than* a given upper bound.

$\phi \Rightarrow \psi$	for	$\neg\phi \vee \psi$
$\phi \wedge \psi$	for	$\neg(\neg\phi \vee \neg\psi)$
$\phi \Leftrightarrow \psi$	for	$(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$
ξ dead	for	$\neg(\xi$ alive)
ξ old	for	$(\xi$ alive) \wedge $\neg(\xi$ new)
$\forall x \in \tau : \phi$	for	$\neg\exists x \in \tau : \neg\phi$
$\exists x \neq \xi : \phi$	for	$\exists x : (x \neq \xi) \Rightarrow \phi$
$\exists x \in \xi.M$ ref: ϕ	for	$\exists x \in C.M$ ref: $(x.owner = \xi) \Rightarrow \phi$

Table 3.1: BOTL abbreviations.

Along the computation of the hotel h , eventually at least one guest will check in: $F[\exists m \in h.checkIn \text{ ref} : m \text{ alive}]$
Guests of hotel h are not hosted forever: $G[\forall x \in \text{Guest} : \text{includes}(h.guests, x) \Rightarrow F[\neg\text{includes}(h.guests, x)]]$
The hotel is never completely empty: $G[\text{size}(h.guests) > 0]$
A guest cannot be hosted in more than one room: $G[\forall x \in \text{Guest} : \exists y \in \text{Room ref} : \exists z \in \text{Room ref} : \text{includes}(y.guests, x) \wedge \text{includes}(z.guests, x) \Rightarrow y = z].$
A guest cannot be checked in and out at the same time: $G[\forall x \in \text{Guest} : \exists m_1 \in h.checkIn \text{ ref} : m_1.g = x \Rightarrow \neg\exists m_2 \in h.checkOut \text{ ref} : m_2.g = x]$
A guest cannot be checked in more than once unless he has been checked out: $G[\forall x \in \text{Guest} : \exists m_1 \in h.checkIn \text{ ref} : m_1.g = x \Rightarrow (\neg\exists m_2 \in h.checkIn \text{ ref} : m_1 \neq m_2 \wedge m_2.g = x) \cup (\exists m_3 \in h.checkOut \text{ ref} : m_3.g = x)]$
The check-out procedure of a guest cannot start until the check-in procedure has been completed: $G[\forall x \in \text{Guest} : \exists m_1 \in h.checkIn \text{ ref} : m_1.g = x \Rightarrow (\neg\exists m_2 \in h.checkOut \text{ ref} : m_2.g = x) \cup m_1 \text{ dead}].$

Table 3.2: Some BOTL example properties.

Example 3.2.2. Table 3.2 reports some example properties expressible in BOTL. The boolean list operator $\text{includes}(l, e)$ is true if and only if the element e belongs to the list l ; the operator $\text{size}(l)$ returns the cardinality of l . \square

CTL or LTL? As we have seen, for the specification of the temporal aspects, BOTL is based on the linear temporal logic LTL. A fair question would be why we have chosen LTL instead of another temporal logic as for example CTL. As a matter of fact, for the definition of BOTL there is no special reason to prefer one of these logics over the other. And in fact, we have designed BOTL with the intention to make the object-based ingredients as orthogonal as possible to the temporal aspects. This means that BOTL can be adapted, in a straightforward manner, not only to CTL — as we have done in the original definition of BOTL [42] — but even to other (more expressive) temporal logics like CTL* or μ -calculus. An approach for the latter case as been investigated by [13].

3.3 Semantics of BOTL

3.3.1 BOTL operational models

In the design of BOTL we have concentrated on the *essential* features of an object-based system. By this we mean that the logic can only address features, such as object attributes, that are likely to be available in any reasonable behavioural model of an object system. Accordingly, we will define the semantics of BOTL using an operational model that is as “poor” as possible, i.e., includes those features addressable by the logic but no more than those. In this chapter, we do not go into the question how such a model is to be generated. For instance, the degree of parallelism or the way of method invocation is part of the translation of an object-oriented language to the model. Any richer kind of model can be abstracted to a BOTL model; thus, hopefully, the logic can be used to express properties of behaviour models generated by a wide range of formalisms. Later, in Section 4.4 and Section 5.6, we will study how models can be created from two simple imperative languages that include some essential features of object-based languages such as object creation and a simplified version of navigation.

We first need to give the notions of classes, methods and variables more substance. Consider the following partial functions:

$$\begin{aligned} \text{VDECL} &= \text{VNAME} \rightarrow \text{TYPE} \\ \text{MDECL} &= \text{MNAME} \rightarrow \text{VDECL} \times \text{TYPE} \\ \text{CDECL} &= \text{CNAME} \rightarrow \text{VDECL} \times \text{MDECL} \end{aligned}$$

A variable declaration in VDECL is a partial function mapping variable names to the corresponding (image) types. MDECL does the same for method names, taking into account that these are actually functions with formal parameters and a return value. Finally, each $D \in \text{CDECL}$ is a class declaration mapping class names to the corresponding attribute and method declarations.

Let us assume the class declaration $D \in \text{CDECL}$ to be given. For any class $C \in \text{dom}(D)$, we denote $C.attrs$ ($\in \text{VDECL}$) for its attribute declara-

$\text{Hotel.attrs}(v) = \begin{cases} \text{nat} & \text{if } v \in \{\text{numOfFloors}, \\ & \text{numOfRooms}\} \\ \text{Room list} & \text{if } v = \text{rooms} \\ \text{Guest list} & \text{if } v = \text{guests} \\ \perp & \text{otherwise} \end{cases}$
$\text{Room.attrs}(v) = \begin{cases} \text{nat} & \text{if } v \in \{\text{floorNumber}, \\ & \text{roomNumber}, \text{numOfBeds}\} \\ \text{Hotel} & \text{if } v = \text{hotel} \\ \text{Guest list} & \text{if } v = \text{guests} \\ \perp & \text{otherwise} \end{cases}$
$\text{Guest.attrs}(v) = \begin{cases} \text{nat} & \text{if } v \in \{\text{guestCode}, \text{age}, \\ & \text{numOfNights}\} \\ \text{Hotel} & \text{if } v = \text{hotel} \\ \text{Room} & \text{if } v = \text{room} \\ \perp & \text{otherwise} \end{cases}$
$\text{checkIn.fpars}(v) = \begin{cases} \text{Guest} & \text{if } v = g \\ \perp & \text{otherwise} \end{cases}$
$\text{checkOut.fpars}(v) = \text{checkIn.fpars}(v)$
$\text{Hotel.meths}(M) = \begin{cases} (\text{checkIn.fpars}, \text{void}) & \text{if } M = \text{checkIn} \\ (\text{checkOut.fpars}, \text{void}) & \text{if } M = \text{checkOut} \\ \perp & \text{otherwise} \end{cases}$
$\text{Room.meths}(M) = \perp$
$\text{Guest.meths}(M) = \perp$

Table 3.3: Class definition $D \in \text{CDECL}$ for the Hotel example.

tion function, and $C.meths$ ($\in \text{MDECL}$) for its method declaration function; thus, $D(C) = (C.attrs, C.meths)$. Furthermore, if the class C of a method M is clear from the context then we use $M.fpars$ ($\in \text{VDECL}$) to denote the formal parameters of M and $M.retty$ ($\in \text{TYPE}$) for the return type; hence $C.meths(M) = (M.fpars, M.retty)$.

Example 3.3.1. For the Hotel model of Figure 2.3, the class definition $D \in \text{CDECL}$ is contained in Table 3.3 where $v \in \text{VNAME}$, $M \in \text{MNAME}$ and $C \in \text{CNAME}$. \square

BOTL operational models are generalised Büchi automata⁴, i.e., tuples $\mathcal{M}_D = \langle \text{Conf}, \rightarrow, I, \mathcal{F} \rangle$ where Conf is the set of configurations (or states)

⁴In the original definition of BOTL given in [42], we used Kripke structures. This mod-

over which $\rightarrow \subseteq Conf \times Conf$ defines a transition relation, $I \subseteq Conf$ is the (non-empty) set of initial states and $\mathcal{F} \subseteq 2^{Conf}$ is the set of accept states⁵. The components Σ and L representing the alphabet and the interpretation of the atomic propositions are not explicitly represented in the model because they are not relevant for the interpretation of the logic. In fact, in BOTL the atomic propositions would correspond to S_{BOTL} and their interpretation in a state will be given by the definition of their semantics in Section 3.3.2. $D \in CDECL$ is the global class declaration, whereas the elements of $Conf$ are tuples of the form $(O, E, \varsigma, \gamma)$ where:

- $O \subseteq \text{OID}$;
- $E \subseteq \text{EVT}$;
- $\varsigma : O \rightarrow \text{VNAME} \rightarrow \text{VAL}$;
- $\gamma : E \rightarrow (\text{VNAME} \rightarrow \text{VAL}) \times \text{VAL}_{\perp}$.

We discuss these elements briefly:

- O describes the objects currently alive in the state.
- E describes the method occurrences currently alive (active).
- For each alive object $o \in O$, $\varsigma(o)$ denotes the local state of o , i.e., it records the values of the attributes of o . ς has to be consistent with D in the sense that $\varsigma(o) = \ell$ with $o \in \text{OID}^C$ implies $\text{dom}(\ell) = \text{dom}(C.attrs)$ and $\ell(a) \in \text{VAL}^{C.attrs(a)}$ for all $a \in \text{dom}(\ell)$.
 ς is extended point-wise to *lists* of objects; thus $\varsigma(\square)(a) = \square$ and $\varsigma(h :: w)(a) = \varsigma(h)(a) :: \varsigma(w)(a)$.
- The images of γ consist of a (partial) mapping of variable names to values, representing the valuation of the formal parameters of the method invocation, as well as the value returned by the method. The latter is defined when the method has terminated; otherwise the value can be \perp . γ has to be consistent with D : if $\gamma(\zeta) = (\ell, v)$ for a given method occurrence $\zeta \in \text{EVT}^{C,M}$ then $\text{dom}(\ell) = \text{dom}(M.fpars)$ and $\ell(p) \in \text{VAL}^{M.fpars(p)}$ for all $p \in \text{dom}(\ell)$, and $v \in \text{VAL}_{\perp}^{M.retty}$.

Assumptions. The transition relation \rightarrow satisfies the following property regarding the termination of method invocations: if an active method occurrence ζ becomes inactive then it has a well-defined return value (i.e., not \perp). Formally: if $(O, E, \varsigma, \gamma) \rightarrow (O', E', \varsigma', \gamma')$ then:

$$\zeta \in E \setminus E' \Rightarrow \exists v \in \text{VAL} : \gamma(\zeta) = (\ell, v).$$

ification is not at all critical since—as we have seen in Section 2.1.1—there exists a tight correspondence between the two kind of models. Here the choice of Büchi automata is only driven by consistency reasons with subsequent chapters of the dissertation.

⁵See Section 2.1.1 for an introduction to Büchi automata.

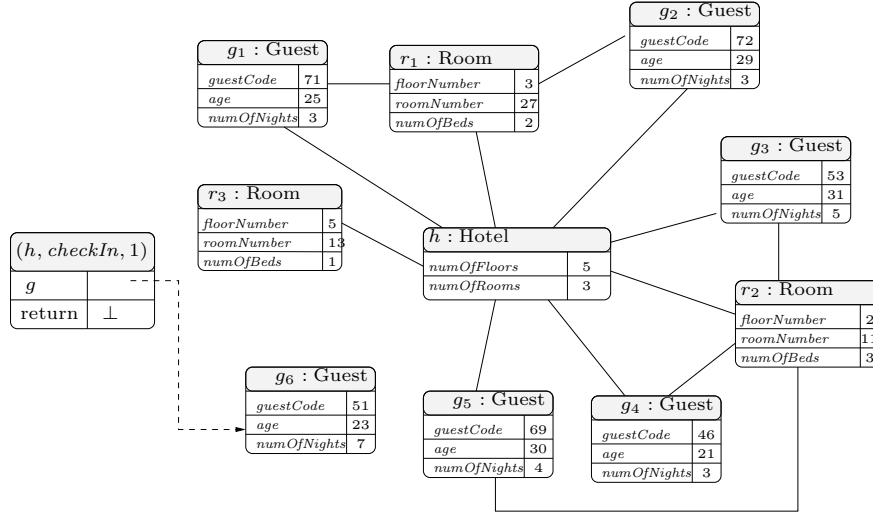


Figure 3.1: A possible configuration of the Hotel model.

Intuitively, we say that a method *has terminated* in the current state if it is deallocated in the next state. However, we assume that at the moment of termination the return value is defined. This might be because of the execution of a return statement (as in Java for example). Furthermore, we assume that *Conf* contains no terminated or deadlocked configurations; i.e., there is at least one outgoing transition from every element of *Conf*. (This property is imposed only for the sake of simplifying the definitions later on; it can be satisfied easily by adding a self-loop to every deadlocking configuration.)

Example 3.3.2. Figure 3.1 depicts a possible configuration of the Hotel model. In particular we have:

$$\begin{aligned} O &= \{h, r_1, r_2, r_3, g_1, g_2, g_3, g_4, g_5, g_6\} \\ E &= \{(h, \text{checkIn}, 1)\} \end{aligned}$$

where we have adopted the following abbreviation: $h = (\text{Hotel}, 1)$, $g_i = (\text{Guest}, i)$ and $r_i = (\text{Room}, i)$ for $i \in \mathbb{N}$. The objects show the values of the components ς and γ . For example, for object g_6 we have:

$$\begin{aligned} \varsigma(g_6)(\text{guestCode}) &= 51 \\ \varsigma(g_6)(\text{age}) &= 23 \\ \varsigma(g_6)(\text{numOfNights}) &= 7 \end{aligned}$$

For the other objects, ς can be obtained in a similar way. The γ component for the only active method is $\gamma(h, \text{checkIn}, 1) = (g \mapsto g_6, \perp)$. \square

3.3.2 Semantics of BOTL static expressions

We are now in a position to define the semantics of our logic. We assume the class declaration D to be fixed and given. Let $\Theta = \text{LVAR} \rightarrow \text{VAL}$, ranged over by θ , be the set of maps that assign values to (some of) the logical variables.

The semantics of expressions is given by the function

$$\llbracket - \rrbracket : \mathcal{S}_{\text{BOTL}} \rightarrow (\text{Conf} \times 2^{\text{Obj} \cup \text{Evt}} \times \Theta) \rightarrow \text{VAL}_{\perp}.$$

Let $q = (O_q, E_q, \varsigma_q, \gamma_q)$ be a configuration of \mathcal{M}_D and $N \subseteq (O_q \cup E_q)$ representing the set of *new* object and method occurrences in the configuration. Then:

$$\begin{aligned} \llbracket x \rrbracket_{q,N,\theta} &= \theta(x) \\ \llbracket \xi.a \rrbracket_{q,N,\theta} &= \ell(a) \quad \text{where } \llbracket \xi \rrbracket_{q,N,\theta} \in C \text{ ref and } \varsigma_q(\llbracket \xi \rrbracket_{q,N,\theta}) = \ell \\ &= \vec{\ell}(a) \quad \text{where } \llbracket \xi \rrbracket_{q,N,\theta} \in C.M \text{ ref and } \gamma_q(\llbracket \xi \rrbracket_{q,N,\theta}) = (\ell, v) \\ &= \vec{\ell}(a) \quad \text{where } \llbracket \xi \rrbracket_{q,N,\theta} \in C \text{ ref list and } \varsigma_q(\llbracket \xi \rrbracket_{q,N,\theta}) = \vec{\ell} \\ &= \vec{\ell}(a) \quad \text{where } \llbracket \xi \rrbracket_{q,N,\theta} \in C.M \text{ ref list and } \gamma_q(\llbracket \xi \rrbracket_{q,N,\theta}) = (\vec{\ell}, \vec{v}) \\ \llbracket \xi.\text{owner} \rrbracket_{q,N,\theta} &= o \quad \text{where } \llbracket \xi \rrbracket_{q,N,\theta} = (o, M, j) \\ \llbracket \xi.\text{return} \rrbracket_{q,N,\theta} &= v \quad \text{where } \gamma_q(\llbracket \xi \rrbracket_{q,N,\theta}) = (\ell, v) \\ \llbracket \xi \text{ new} \rrbracket_{q,N,\theta} &= (\llbracket \xi \rrbracket_{q,N,\theta} \in N) \\ \llbracket \xi \text{ alive} \rrbracket_{q,N,\theta} &= (\llbracket \xi \rrbracket_{q,N,\theta} \in O_q \cup E_q) \\ \llbracket \omega(\xi_1, \dots, \xi_n) \rrbracket_{q,N,\theta} &= \llbracket \omega \rrbracket(\llbracket \xi_1 \rrbracket_{q,N,\theta}, \dots, \llbracket \xi_n \rrbracket_{q,N,\theta}) \\ \llbracket \text{with } x_1 \in \xi_1 \text{ from } x_2 := \xi_2 \text{ do } x_2 := \xi_3 \rrbracket_{q,N,\theta} &= \llbracket \text{for } x_1 \in \llbracket \xi_1 \rrbracket_{q,N,\theta} \text{ do } x_2 := \xi_3 \rrbracket_{q,N,\theta\{\llbracket \xi_2 \rrbracket_{q,N,\theta}/x_2\}} \\ \text{where } \llbracket \text{for } x_1 \in \square \text{ do } x_2 := \xi \rrbracket_{q,N,\theta} &= \llbracket x_2 \rrbracket_{q,N,\theta} \\ \llbracket \text{for } x_1 \in h :: w \text{ do } x_2 := \xi \rrbracket_{q,N,\theta} &= \llbracket \text{for } x_1 \in w \text{ do } x_2 := \xi \rrbracket_{q,N,\theta\{\llbracket \xi \rrbracket_{q,N,\theta\{h/x_1\}}/x_2\}} \end{aligned}$$

The semantics of a variable x corresponds to its interpretation θ . The semantics of a navigation expression $\xi.a$ is the value of the function ℓ (cf. page 46) in the attribute a of the object denoted by ξ . If ξ is a method, then ℓ corresponds to the value of the formal parameter a at the moment of the method call. In case ξ denotes a list of references to objects or method calls, the definition is extended point-wise. The semantics of the formula $\xi.\text{owner}$ (where ξ represents a method occurrence) is simply the object executing the method. The semantics of $\xi.\text{return}$ is the return value stored in the component γ of the method instance denoted by ξ . $\xi \text{ new}$ is true if the object or method call denoted by ξ belongs to the set N of new objects and method calls in the current configuration. $\xi \text{ alive}$ is true if and only if the object or method invocation denoted by ξ is indeed

among the object and method instances present in the current state. The semantics of an operation $\omega(\xi_1, \dots, \xi_n)$ is given by the operation $\llbracket \omega \rrbracket$ applied to the interpretation of its parameter ξ_1, \dots, ξ_n in the current configuration. Finally, the “with-from-do”-expression is evaluated by means of the “for-do” meta-expression, which successively re-computes the “do”-expression for every value of x_1 out of the “for”-list.⁶

Example 3.3.3. Consider the object diagram in Figure 3.1, and suppose we want to evaluate $x.rooms.guests$ in the configuration $q = (O, E, \zeta, \gamma)$ with variable interpretation $\theta: x \mapsto h$ and $N = \emptyset$. Skipping some details, we obtain:

$$\begin{aligned} \llbracket x.rooms.guests \rrbracket_{q,N,\theta} &= \zeta(\llbracket x.rooms \rrbracket_{q,N,\theta})(guests) \\ &= \zeta(\zeta(\llbracket x \rrbracket_{q,N,\theta})(rooms))(guests) \\ &= \zeta(\zeta(h)(rooms))(guests) \\ &= \zeta([r_1, r_2, r_3])(guests) \\ &= [[g_1, g_2], [g_3, g_4, g_5], []] . \end{aligned}$$

As expected, the result is a list of lists. □

3.3.3 Semantics of BOTL temporal formulae

The semantics of BOTL formulae is now rather straightforward. It is defined by a satisfaction relation between an accepting run of the model \mathcal{M}_D defined by the transition system, a set of new objects and method occurrences N , a valuation θ and a formula ϕ . To define it, we recall some notation and in particular Definition 2.1.2 of path through a transition model and Definition 2.1.6 of accepting run for Büchi automaton. An accepting run is simply a path satisfying a fairness condition on the accept states. Let $\mathcal{M}_D = \langle Conf, \rightarrow, I, \mathcal{F} \rangle$. For a path of configurations $\eta \in Conf^\omega$ we denote by $\eta[i]$ the $(i+1)$ -th state and we write η^i for the suffix of η starting at state $\eta[i]$, i.e., $\eta^i = \eta[i]\eta[i+1]\eta[i+2] \dots$. Moreover, we denote $Inf(\eta)$ the set of configurations in η that occur infinitely often and we indicate by $(O_i^\eta, E_i^\eta, \zeta_i^\eta, \gamma_i^\eta)$ the single components of configuration $\eta[i]$. Formally, a run of \mathcal{M}_D is an infinite path of configurations $\eta \in Conf^\omega$ such that $\eta[0] \in I$ and $Inf(\eta) \cap F \neq \emptyset$ for all $F \in \mathcal{F}$.

The set N_i^η of new objects and method occurrences at state $i \geq 0$ of η is defined as

$$\begin{aligned} N_0^\eta &= N \subseteq (O_0^\eta \cup E_0^\eta) \\ N_{i+1}^\eta &= (O_{i+1}^\eta \setminus O_i^\eta) \cup (E_{i+1}^\eta \setminus E_i^\eta). \end{aligned}$$

where N is the set of initial new objects and new method occurrences assumed

⁶For the reader familiar with functional programming: with $x_1 \in \xi_1$ from $x_2 := \xi_2$ do $x_2 := \xi_3$ may alternatively be translated to $foldl \llbracket \xi_1 \rrbracket_{q,N,\theta} \llbracket \xi_2 \rrbracket_{q,N,\theta} \lambda v h. \llbracket \xi_3 \rrbracket_{q,N,\theta} \{h/x_1, v/x_2\}$.

to be given⁷. Similarly, given $\theta \in \Theta$, let $\theta_i^\eta \in \Theta$ denote the valuation of the logical variables in configuration i . This valuation must be undefined for the variables that θ maps outside the alive objects and method occurrences in configuration i .

$$\theta_i^\eta(x) = \begin{cases} \theta(x) & \text{if } \forall k \leq i : \theta(x) \in O_k^\eta \cup E_k^\eta \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We need the following auxiliary notation. Let $\text{VAL}^\tau \upharpoonright (O, E)$ denote the subset of VAL^τ alive w.r.t. the sets O and E of alive objects and method occurrences. It is defined as:

$$\text{VAL}^\tau \upharpoonright (O, E) = \begin{cases} \text{VAL}^\tau \cap O & \text{if } \tau = C \text{ ref} \\ \text{VAL}^\tau \cap E & \text{if } \tau = C.M \text{ ref} \\ \text{VAL}^\tau & \text{otherwise.} \end{cases}$$

Intuitively if $\tau \neq C \text{ ref}$ or $\tau \neq C.M \text{ ref}$ then, by definition, the alive values are precisely all the elements of VAL^τ . Otherwise, in case of object references we have to restrict only to the subset of O of type τ . Similarly for method occurrences.

The semantics of temporal formulae is now given by a relation $\models \subseteq (\text{Conf}^\omega \times (\text{Oid} \cup \text{EVT}) \times \Theta) \times \mathbb{T}_{\text{BOTL}}$. Let η be an accepting run of \mathcal{M}_D , $N \subseteq O_0^\eta \cup E_0^\eta$, and $\theta \in \Theta$ then

$$\begin{aligned} \eta, N, \theta \models \xi & \quad \text{iff } \llbracket \xi \rrbracket_{\eta[0], N, \theta} = \text{tt} \\ \eta, N, \theta \models \neg\phi & \quad \text{iff } \eta, N, \theta \not\models \phi \\ \eta, N, \theta \models \phi \vee \psi & \quad \text{iff } \text{either } (\eta, N, \theta \models \phi) \text{ or } (\eta, N, \theta \models \psi) \\ \eta, N, \theta \models \exists x \in \tau : \phi & \quad \text{iff } \exists v \in \text{VAL}^\tau \upharpoonright (O_0^\eta, E_0^\eta) : \eta, N, \theta\{v/x\} \models \phi \\ \eta, N, \theta \models \mathbf{X}\phi & \quad \text{iff } \eta^1, N_1^\eta, \theta_1^\eta \models \phi \\ \eta, N, \theta \models \phi \mathbf{U} \psi & \quad \text{iff } \exists j \geq 0 : \\ & \quad (\eta^j, N_j^\eta, \theta_j^\eta \models \psi \text{ and } \forall 0 \leq k < j : \eta^k, N_k^\eta, \theta_k^\eta \models \phi) \end{aligned}$$

Most of the temporal operators have standard semantics (cf. the definition of semantics of LTL in Section 2.1.2). Note that in the existential quantification, the variable x is bound only to alive values in the current state.

3.4 Object Constraint Language

The notation given by UML is mainly based on diagrams. Although this visual nature has provided the UML with a widespread popularity among practitioners, sometimes diagrams do not give the right level of expressiveness needed. In

⁷ N will usually depend on the particular specialisation of the BOTL operational model. For example, in some cases it could be reasonable to consider every object and method occurrence present in the initial state to be new. For other models, it could be reasonable to consider only a proper subset of $O_0^\eta \cup E_0^\eta$.

many situations, using a textual language can help to formulate concise specifications. The Object Constraint Language (OCL) [88, 110, 111] is a part of the UML that provides a framework to add textual annotation to UML diagrams. More specifically, the important class of annotations covered by OCL is given by *constraints*, i.e., statements that impose additional restrictions on the UML model. There are two kinds of constraints in OCL:

- *invariants*, i.e., statements that should be valid at any point in the computation;
- *pre- and postconditions*, i.e., statements about the start and end of a method execution.

In the following section, we describe some of the basic concepts proper of OCL, later in Section 3.4.2 we give a formal syntax of (a subset of) OCL.

3.4.1 An informal and concise summary of OCL basic concepts

The content of this subsection is mostly based on [95, 110].

Types. OCL is a typed language. Its types can be grouped in

- *predefined types* and
- *model types*.

The first group is composed of basic types such as *Integer*, *Real*, *String*, *Boolean* and collection types, that is *Collection(T)*, *Set(T)*, *Bag(T)* and *Sequence(T)*. The parameter *T* ranges over types, thus, for example, we have *Bag(Integer)* for a multi-set of integers. Each of the predefined types has a set of standard operations defined. For example:

- $+$, $-$, $*$, $/$ for *Integer* and *Real*;
- **and**, **or**, **not** for *Boolean*;
- *union*, *intersection*, *includes* (i.e., membership test) for *Set* and *Bag*;
- *concatenation* for *String* and so on.

Among the predefined types there is also the special types *OclAny*.

Model types are all those defined in the UML model. For example, in the Hotel class diagram in Figure 2.3 these are: types Hotel, Guest and Room.

The set of OCL types is organised in a hierarchy depicted in Figure 3.2 where special types for the meta-model are not included. For example, *Collection* is a super type of *Set*, *Bag* and *Sequence*. Apart from *Collection*, all other types are subtypes of *OclAny*. We will not be concerned with issues related to sub-typing in this thesis.

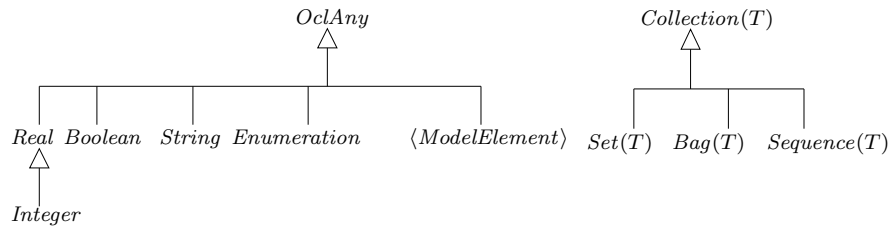


Figure 3.2: OCL types hierarchy

Expressions and constraints. Expressions are built out of literals, identifiers and variables. Each OCL expression has a type and a result. The type of the result is the type of the expression. For example, the expression $3 + 4$ has result 7 and type *Integer*. An expression yields an *undefined* value if one of its operands is undefined. OCL expressions may use navigation expressions in similar way as BOTL. Constraints in OCL are built on the basis of OCL expressions and have a boolean type. Moreover, constraints have a context that changes depending on their kind. For invariants, the context is given by a class. The following OCL invariant:

```

context Room invariant
  guests→size ≤ numOfBeds

```

states that the number of guests in a room cannot exceed the number of beds in the room.

Pre- and postconditions have a method and a class as context. They define a contract that an implementation of the method must fulfil [81]. Thus, pre- and postconditions specify necessary requirements that must be satisfied to consider the implementation of a method to some extent correct. For instance, in

```

context Hotel :: checkIn(g : Guest)
pre not guests→includes(g)
post guests→size = (guests@pre→size) + 1 and guests→includes(g)

```

the precondition states that the person to be checked in is not a current guest of the hotel, while the postcondition states that after checking him/her in, the number of guests has increased by one and the new guest is one of the current guests. The `@pre`-operator refers to the number of guests at the beginning of the check-in. The standard OCL operation *size* determines the number of elements of a collection. Note that invariants as well as pre- and postconditions are safety properties, whereas BOTL also allows to express liveness properties such as the first one in Example 3.2.2.

3.4.2 OCL syntax

The set of OCL constraints and OCL expressions is given by the following grammar:

$$\begin{aligned}
 (\chi \in) \mathbf{C}_{\text{OCL}} &::= \text{context } C \text{ invariant } \xi \mid \text{context } C :: M(\vec{p}) \text{ pre } \xi \text{ post } \xi \\
 (\xi \in) \mathbf{S}_{\text{OCL}} &::= \text{self} \mid z \mid \text{result} \mid \xi @ \text{pre} \mid \xi.a \mid \omega(\xi, \dots, \xi) \\
 &\quad \mid \xi.\omega(\xi, \dots, \xi) \mid \xi \rightarrow \omega(\xi, \dots, \xi) \mid \xi \rightarrow \text{iterate}(x_1; x_2 = \xi \mid \xi)
 \end{aligned}$$

As for BOTL we assume that OCL terms are type correct (with, however some differences in the possible types; see below). At the top level, a *constraint* χ can either be an invariant or a pre- and postcondition (see Section 3.4.1). As discussed before, the context of a constraint is a class C in case of an invariant or a class and a method $M \in \text{dom}(C.\text{meths})$ in case of pre- and postconditions. The context can be referred to by the expression in the constraint. For instance, in an OCL navigation expression $\text{self}.a$, we describe a route starting from an object of the context class C .

Many of the expressions $\xi \in \mathbf{S}_{\text{OCL}}$ have their direct counterpart in BOTL.

- **self** refers to the context object of the class C .
- z represents either an attribute of the context object, or a formal parameter of the context method, or a logical variable.
- **result** refers to the value returned by the context method.
- **@pre** is a suffix that refers to the value of its operand at the time of the method invocation. Both **result** and **@pre** may be used in postconditions only (see below).
- $\xi.a$ and $\omega(\xi_1, \dots, \xi_n)$ are the same as for BOTL, i.e., they express navigations and ω operations on types.
- $\xi.\omega(\xi_1, \dots, \xi_n)$ represents an operator ω on basic types that is applied to ξ, ξ_1, \dots, ξ_n . If the expression ξ is a collection (i.e., a set, bag or list), we have the special case $\xi \rightarrow \omega(\xi_1, \dots, \xi_n)$.
- $\xi_1 \rightarrow \text{iterate}(x_1; x_2 = \xi_2 \mid \xi_3)$ has the same meaning as with $x_1 \in \xi_1$ **from** $x_2 := \xi_2$ **do** $x_2 := \xi_3$. The difference is only in the type that can be returned, namely sets and bags (see Section 3.5.1). A large group of OCL queries (e.g., **exists**, **forAll**, **select**, **reject**, **collect**) can be reduced to *iterate* expressions (and therefore to with-from-do expressions) [24, 97, 110].

Particular OCL features not included in the previous syntax are expressions of the kind $M(\xi, \dots, \xi)$ and $\xi.M(\xi, \dots, \xi)$ where M is a so-called *query method*; i.e., M is a method which returns a value without side effects. Nevertheless, also constraints where query methods appear can be translated in terms of

another OCL expression that does not contain them but that describes the function implemented by the query method⁸. Thus, as in other related works [56, 96], we do not treat query methods explicitly.

Example 3.4.1. The OCL invariant:

```
context Guest invariant
age ≥ 18
```

says that every guest of the hotel must be at least 18 years old. In BOTL, this has a straightforward translation: $G[\forall x \in \text{Guest ref} : x.age \geq 18]$. \square

Example 3.4.2. As a more involved example, consider the following OCL invariant:

```
context Hotel invariant
rooms.guests = guests
```

states that the collection of guests in the rooms of the hotel should be consistent with the collection of guests maintained at the hotel. Clearly, this statement is not valid in every state of the system as, for instance, its validity cannot be guaranteed while executing a method that changes the number of guests (like checking a guest in or out). In BOTL, the same property would be expressed by

$$G[\forall x \in \text{Hotel ref} : (\neg \exists m \in x.checkIn \text{ ref} : \text{tt} \wedge \neg \exists m' \in x.checkOut \text{ ref} : \text{tt}) \\ \Rightarrow \text{sort}(\text{flat}(x.rooms.guests)) = \text{sort}(x.guests)].$$

The function *flat* (defined in Section 3.2.1) flattens nested lists; we need it because *x.rooms.guests* is a list of lists, whereas *x.guests* is a simple list. The function *sort* orders lists. Note that requiring the absence of occurrences *m, m'* of the methods *checkIn* and *checkOut* is essential: during the execution of a *checkIn* and *checkOut*, it is not possible to guarantee the validity of the invariant since these methods change the number of guests. The same observation would apply to any other method that could modify the state. \square

3.4.3 Some OCL restrictions

OCL should allow to specify constraints on UML models in a *formal way*. However, as pointed out by several authors in the literature [31, 53, 55], in its early stages, OCL contained a number of ambiguities as well as inconsistencies, mostly due to the lack of a mathematical foundation. These restrictions hardly made OCL a truly formal language. Lately more attention has been given on its mathematical foundation⁹. It has become clear that, by a rigorous formalisation, OCL would gain the necessary precision needed to correctly formulate

⁸Provided the function is not defined recursively.

⁹As testified also by the Revised Submission 1.5 of OCL 2.0 suggested for the “UML 2.0 Request for Proposals for OCL” of the OMG that contains a section on the formal semantics of OCL taken mostly from [95].

constraints on UML models. As a direct consequence, it would become possible to design software tools supporting this constraint language. Following different approaches (see Section 3.6.2), several attempts that aim to provide OCL with a formal foundation have been carried out. In Section 3.5, we present our proposal that consists of a translation from a large subset of OCL into BOTL.

Following [95], we give a few example concepts known in the literature that are not defined satisfactorily in OCL.

Flattening. The definition of OCL does not allow nested collection types as, for example:

$$Set(Set(Integer)).$$

The OCL specification prescribes that in case of nested collections the result type is “automatically flattened”. However, how the flattening should be done is defined only by means of an naive example (see [89], Chapter 7, pp.7-20):

$$Set\{Set\{1, 2\}, Set\{3, 4\}, Set\{5, 6\}\} = Set\{1, 2, 3, 4, 5, 6\}.$$

Indeed, this instance works perfectly. Nevertheless, it is not difficult to construct other examples where in fact this is not the case anymore. Let g_1 , g_2 , g_3 be of type Guest and consider the expression:

$$Sequence\{Set\{g_1, g_2, g_3\}\}$$

that may be obtained by navigating from Hotel, via Room to the class Guest¹⁰. In this case, the flattening does not work anymore since the result depends on the order of the elements in $Set\{g_1, g_2, g_3\}$. By definition, the elements of a set are not ordered, therefore, any of the following results is acceptable:

$$\begin{aligned} &Sequence\{g_1, g_2, g_3\} \quad \text{or} \quad Sequence\{g_1, g_3, g_2\} \quad \text{or} \\ &Sequence\{g_2, g_1, g_3\} \quad \text{or} \quad Sequence\{g_2, g_3, g_1\} \quad \text{or} \\ &Sequence\{g_3, g_2, g_1\} \quad \text{or} \quad Sequence\{g_3, g_1, g_2\}. \end{aligned}$$

In order to make the flattening deterministic it seems unavoidable to impose an order on the elements of sets. In Section 3.5, we will propose a possible solution to this problem.

Iterate expression. The OCL expression `iterate` provides a rather powerful iteration mechanism on collections since many of the expressions on collections can be stated in terms of it. However, the evaluation of the iterate expression may be problematic. In fact, the result is not always well defined, meaning that there can be different result for the same input, or in other words, it may behave nondeterministically. Once again this happens when the whole iteration depends on the order in which the elements of the collection are chosen [96, 97]. Hence, when the iteration is done on collections like *Set* or *Bag*.

¹⁰Assuming that in the association *rooms* of the class Hotel there would be the keyword `ordered` that yields a sequence instead of a set.

As an example, assume the class `Guest` of our running example would have an attribute name of type `String`. Then, consider the following OCL operation that returns the concatenation of the guest names:

$$h.guests \rightarrow \text{iterate}(x_1; x_2 = ' ' \mid x_2.\text{concat}(x_1.\text{name})).$$

Clearly, if `h.guests` is a set or a bag then there will be more than one result for this expression according to the order in which elements of `h.guests` are bound to x_1 . Similarly, consider the following OCL expression:

$$\{1, 2, 4\} \rightarrow \text{iterate}(x_1; x_2 = 0 \mid -x_2 + x_1).$$

Again, the result will be either -1, or 3, or 5 according to the order in which element of $\{1, 2, 4\}$ are bound to x_1 .

For a discussion on other OCL restrictions (e.g. concerning type issues) the reader is referred to [25] where a list of resolved issues is also provided.

3.5 Translating OCL into BOTL

In this section, we will give a translation of OCL into BOTL and investigate differences as well as relations between these two languages. First note that BOTL is not primarily intended to be the exact formal counterpart of OCL. In defining BOTL we were concerned with some issues derived mostly from our aim to do model checking of object-based programs. On the other hand, our logic can be seen as one of the many approaches on how to give a sound foundation to OCL. At the same time, the translation provides us with a good insight in the expressiveness of BOTL.

3.5.1 Translation issues

Before proceeding with the formal translation of OCL into BOTL, let us give the intuition, in a rather informal way, of the solutions to the issues involved.

Data types. One of the differences between BOTL and OCL is their type system: rather than arbitrary lists, OCL allows sets, bags and lists of primitive data values; i.e., *nested* lists are not included¹¹. There are two reasons why BOTL considers only arbitrary lists. On the one side, lists have sufficient expressive power to represent sets and bags; on the other side, by using only lists we avoid the problem of *nondeterministic* behaviour in the BOTL expression with-do-from. As we have seen in Section 3.4.3 this problem is present in OCL. Note that by only using lists the problem described in the examples of that section would not occur.

¹¹However, at this moment, it seems that in the Revised Submission version 1.5 of OCL 2.0 the contributors advocate for the use of nested collection types.

In order to have a more rigorous comparison, let us define OCL types. Then we will show how to encode them using BOTL types. For simplicity, we omit strings, reals and enumerations which are absent in BOTL but could be added without essential problems. OCL types are defined by:

$$\begin{aligned}\rho & ::= \text{nat} \mid \text{bool} \mid C \text{ ref} \\ \tau (\in \text{TYPE}_{\text{OCL}}) & ::= \rho \mid \rho \text{ list} \mid \rho \text{ set} \mid \rho \text{ bag}\end{aligned}$$

$\rho \text{ set}$ are sets of elements of type ρ , while $\rho \text{ bag}$ are multi-sets whose elements have type ρ . The semantics of the sorts included in TYPE is unchanged, while for the new types we have the following value domains:

$$\begin{aligned}\text{VAL}^{\rho \text{ set}} & = 2^{\text{VAL}^{\rho}} \\ \text{VAL}^{\rho \text{ bag}} & = \text{VAL}^{\rho} \rightarrow \mathbb{N}.\end{aligned}$$

The set of values in OCL is:

$$\text{VAL}_{\text{OCL}} = \bigcup_{\tau \in \text{TYPE}_{\text{OCL}}} \text{VAL}^{\tau}.$$

Now let us discuss how we will translate OCL operations on sets and bags, say $\xi_1 \rightarrow \omega(\xi_2, \dots, \xi_n)$. For OCL types $\rho \text{ set}$ and $\rho \text{ bag}$, we define functions α_{set} and α_{bag} on BOTL values. These functions abstract from the order of the elements in a list and return a set or a bag, respectively. Formally $\alpha_{\text{set}} : \text{VAL} \rightarrow \text{VAL}_{\text{OCL}}$ is given by:

$$\alpha_{\text{set}}(v) = \begin{cases} \emptyset & \text{if } v = [] \\ \{h\} \cup \alpha_{\text{set}}(w) & \text{if } v = h :: w \\ v & \text{otherwise.} \end{cases}$$

Using $\{\cdot\}$ as notation for bags and \uplus for their union, $\alpha_{\text{bag}} : \text{VAL} \rightarrow \text{VAL}_{\text{OCL}}$ is given by

$$\alpha_{\text{bag}}(v) = \begin{cases} \{\cdot\} & \text{if } v = [] \\ \{h\} \uplus \alpha_{\text{bag}}(w) & \text{if } v = h :: w \\ v & \text{otherwise.} \end{cases}$$

For each operation $\xi_1 \rightarrow \omega(\xi_2, \dots, \xi_n)$ on sets or bags, there exists a corresponding operation in BOTL, say $\bar{\omega}(\xi_1, \xi_2, \dots, \xi_n)$, such that the diagram in Figure 3.3 commutes.

Example 3.5.1. Consider the OCL expression $\xi_1 \rightarrow \text{union}(\xi_2)$. The intended semantics $\llbracket \text{union} \rrbracket$ is the mathematical union on sets. In BOTL, there will be an appropriate operator with semantics $\llbracket \overline{\text{union}} \rrbracket : \text{VAL}^{\tau \text{ list}} \times \text{VAL}^{\tau \text{ list}} \rightarrow \text{VAL}^{\tau \text{ list}}$. According to the commutative diagram, we have that

$$\alpha_{\text{set}}(\llbracket \overline{\text{union}} \rrbracket(v_1, v_2)) = \llbracket \text{union} \rrbracket(\alpha_{\text{set}}(v_1), \alpha_{\text{set}}(v_2)).$$

That is, the result on lists is equal, up to abstraction from sets, to the corresponding union on sets. The operator $\overline{\text{union}}$ can be defined for instance as $\overline{\text{union}}(w_1, w_2) \triangleq \text{concat}(w_1, w_2)$ where w_1 and w_2 are lists. \square

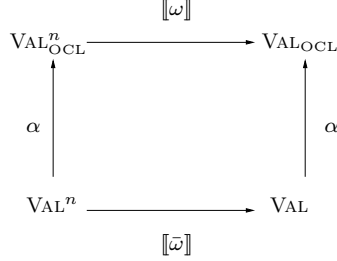


Figure 3.3: Commutative diagram.

Example 3.5.2. Consider now equality on sets in OCL: $\xi_1 = \xi_2$ where ξ_1 and ξ_2 have type set. The corresponding BOTL expression will have semantics $\llbracket \equiv^{\text{set}} \rrbracket : \text{VAL}^{\tau \text{ list}} \times \text{VAL}^{\tau \text{ list}} \rightarrow \text{VAL}^{\text{bool}}$. The operator \equiv^{set} is defined as follow:

$$\equiv^{\text{set}}(w_1, w_2) \triangleq \text{EqList}(\text{sort}(\text{del_duplicates}(w_1)), \text{sort}(\text{del_duplicates}(w_2))).$$

The function *EqList* takes two sorted lists and returns true if they have the same elements, and false otherwise. The function *del_duplicates* deletes the duplicate elements in a list. Apart from *del_duplicates*, the same argument applies to bags.

Invariants. The key issue for the translation of context *C* invariant ξ , concerns the identification of the states in which the invariant expression ξ has to hold. In particular we have to ensure that none of the methods in $\text{dom}(C.\text{meths})$ is active. In fact, during the execution of methods, there can be some intermediate configurations in which ξ does not hold (see Example 3.5.4).

Pre- and postconditions. The translation of pre- and postconditions is more involved. In particular, the OCL operator @pre has to be handled in a special way as it forces us to consider two different moments in time, viz. the start and end of a method invocation. We use the following strategy. Consider the constraint: $\text{context } C :: M(\vec{p}) \text{ pre } \xi_{\text{pre}} \text{ post } \xi_{\text{post}}$. By definition, $\xi_{\text{@pre}}$ subexpressions occur a finite number of times, say $n \geq 0$, only in ξ_{post} . We first enumerate all the occurrences of $\xi_{\text{@pre}}$ subexpressions in ξ_{post} . We write $\xi_{\text{@}_i \text{pre}}$ for $1 \leq i \leq n$. Then when we translate ξ_{post} , by means of the function δ that we will define in the next subsection, we substitute terms $\xi_{\text{@}_i \text{pre}}$ with new fresh logical variables $u_i \in \tau_i$ for $1 \leq i \leq n$. The value of the variable u_i is bound to the appropriate value in the translation of ξ_{pre} . We “add” to the translated precondition $\delta(\xi_{\text{pre}})$ a binding term $u_i = \delta(\xi)$ for all u_i and $\xi_{\text{@}_i \text{pre}}$. Thus, the variables u_i are associated to the value of ξ in $\xi_{\text{@}_i \text{pre}}$ at the beginning of the method execution, and therefore can be used instead of $\xi_{\text{@}_i \text{pre}}$ in the postcondition. Note that the judgement $u_i \in \tau_i$ can be inferred by the type of ξ in $\xi_{\text{@}_i \text{pre}}$.

3.5.2 Translating OCL expressions into BOTL

We will now define a syntactic mapping of OCL into BOTL. First we will give a partial function δ that maps OCL expressions onto BOTL static expressions. Then by means of δ we will address the issues involved in the translation of OCL constraints. The function δ takes three parameters: o , m , \vec{p} . Given $\chi \in \mathbf{C}_{\text{OCL}}$, the first parameter o represents a variable bound to an object of the context class C . In case of pre- and postconditions, the value of parameter m is a method occurrence of the context method M and \vec{p} is the list of its formal parameters. In case of invariants, m has an arbitrary value whereas \vec{p} is the empty list. The translation function $\delta : \mathbf{S}_{\text{OCL}} \rightarrow (\text{LVAR} \times \text{LVAR} \times \text{VNAME}^*) \rightarrow \mathbf{S}_{\text{BOTL}}$ is given by:

$$\delta_{o,m,\vec{p}}(\mathbf{self}) = o$$

$$\delta_{o,m,\vec{p}}(x) = \begin{cases} o.x & \text{if } o \in C \text{ ref and } x \in \text{dom}(C.\text{attrs}) \\ m.x & \text{if } x \in \vec{p} \\ x & \text{otherwise} \end{cases}$$

$$\delta_{o,m,\vec{p}}(\mathbf{result}) = m.\mathbf{return}$$

$$\delta_{o,m,\vec{p}}(\xi@_i\mathbf{pre}) = u_i$$

$$\delta_{o,m,\vec{p}}(\xi.a) = \begin{cases} \mathit{flat}(\delta_{o,m,\vec{p}}(\xi).a) & \text{if } \xi \in C \text{ ref list and } C.\text{attrs}(a) = \tau \text{ list} \\ \delta_{o,m,\vec{p}}(\xi).a & \text{otherwise} \end{cases}$$

$$\delta_{o,m,\vec{p}}(\omega(\xi_1, \dots, \xi_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(\xi_1), \dots, \delta_{o,m,\vec{p}}(\xi_n))$$

$$\delta_{o,m,\vec{p}}(\xi.\omega(\xi_1, \dots, \xi_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(\xi), \delta_{o,m,\vec{p}}(\xi_1), \dots, \delta_{o,m,\vec{p}}(\xi_n))$$

$$\delta_{o,m,\vec{p}}(\xi \rightarrow \omega(\xi_1, \dots, \xi_n)) = \bar{\omega}(\delta_{o,m,\vec{p}}(\xi), \delta_{o,m,\vec{p}}(\xi_1), \dots, \delta_{o,m,\vec{p}}(\xi_n))$$

$$\delta_{o,m,\vec{p}}(\xi_1 \rightarrow \mathbf{iterate}(x_1; x_2 = \xi_2 \mid \xi_3)) =$$

$$\text{with } x_1 \in \delta_{o,m,\vec{p}}(\xi_1) \text{ from } x_2 := \delta_{o,m,\vec{p}}(\xi_2) \text{ do } x_2 := \delta_{o,m,\vec{p}}(\xi_3) .$$

The translation of \mathbf{S}_{OCL} is straightforward for almost every operator.

- A variable x is prefixed by the context object if it is one of its attributes; it is prefixed by m if it is among m 's formal parameters.
- As discussed in the previous section, in translating $\xi@_i\mathbf{pre}$, we assume an enumeration of their occurrences, say $\xi@_i\mathbf{pre}$ for $1 \leq i \leq n$. Each numbered expression is then replaced by a fresh variable u_i .
- In case of attributes or navigations $\xi.a$ we apply the definition recursively on the prefix. If both ξ and a are lists then the resulting BOTL expression has to be flattened since the result would produce a nested list that is not admitted by OCL. This is done explicitly by the operation flat .
- The expressions $\xi \rightarrow \omega(\xi_1, \dots, \xi_n)$ and $\xi.\omega(\xi_1, \dots, \xi_n)$ are translated using the corresponding BOTL $(n + 1)$ -ary operation $\bar{\omega}$.

3.5.3 Translating OCL constraints into BOTL

In this section, we complete the translation of OCL into BOTL by defining a map $\Delta : C_{\text{OCL}} \rightarrow T_{\text{BOTL}}$.

Invariants. In case of an invariant, the translation has the typical prefix G . The invariant must hold for all alive objects of the class C when none of their methods is active. Let $y \in \text{LVAR}$ and $\text{dom}(C.\text{meths}) = \{M_1, \dots, M_k\}$. We define:

$$\begin{aligned} \Delta(\text{context } C \text{ invariant } \xi) = \\ G[\forall x \in C \text{ ref} : (\neg \exists m_1 \in x.M_1 \text{ ref:tt} \wedge \dots \wedge \neg \exists m_k \in x.M_k \text{ ref:tt}) \Rightarrow \delta_{x,y,\llbracket \xi \rrbracket}]. \end{aligned}$$

We give an example of the application of this invariant schema.

Example 3.5.3. Consider again the OCL invariant in Example 3.4.2 assuming that the collection *guests* is a bag.

```
context Hotel invariant
rooms.guests = guests
```

Recalling the considerations of Example 3.5.2 on set equality, we have:

$$\begin{aligned} \delta(\text{rooms.guests} = \text{guests}) &= \text{EqList}(\text{sort}(\delta(\text{rooms.guests})), \text{sort}(\delta(\text{guests}))) \\ &= \text{EqList}(\text{sort}(\text{flat}(x.\text{rooms.guests})), \text{sort}(x.\text{guests})). \end{aligned}$$

We can embed the resulting BOTL expression in the invariant template taking into account that the class *Hotel* has two methods, i.e., *checkIn* and *checkOut*:

$$\begin{aligned} G[\forall x \in \text{Hotel ref} : (\neg \exists m \in x.\text{checkIn ref} : \text{tt} \wedge \neg \exists m' \in x.\text{checkOut ref} : \text{tt}) \\ \Rightarrow \text{EqList}(\text{sort}(\text{flat}(x.\text{rooms.guests})), \text{sort}(x.\text{guests}))]. \end{aligned}$$

Note that apart from the use of the more natural equal sign instead of *EqList*, the resulting invariant coincides with the invariant in Example 3.4.2. \square

Pre- and postconditions. Although the translation of OCL invariants into BOTL is quite straightforward, the translation of pre- and postconditions is more involved and requires an auxiliary definition. In particular, the OCL operator **@pre** has to be handled in a special way as it forces us to consider two different moments in time, i.e., the *start* and *end* of a method invocation. As discussed above, we augment the precondition with some extra information that is used to evaluate the postcondition.

Consider the OCL constraint $\text{context } C :: M(\vec{p}) \text{ pre } \xi_{\text{pre}} \text{ post } \xi_{\text{post}}$. The *extended* translated precondition w.r.t. the object o and the method occurrence m , $\xi_{\text{pre}}^{o,m,\vec{p}}$, is given by

$$\xi_{\text{pre}}^{o,m,\vec{p}} \triangleq \delta_{o,m,\vec{p}}(\xi_{\text{pre}}) \wedge \bigwedge_{\xi @_i \text{pre} \in \xi_{\text{post}}} (u_i = \delta_{o,m,\vec{p}}(\xi))$$

where u_i for $1 \leq i \leq n$ are fresh logical variables.

Here, the symbol \in means “occurs syntactically in”. Thus, for precondition ξ_{pre} we construct an extended precondition $\xi_{\text{pre}}^{o,m,\vec{p}}$ using a new variable u_i for each subexpression $\xi@_i\text{pre}$ occurring in the postcondition. This new variable “freezes” the value of ξ while evaluating the precondition and can be used in the postcondition. Now we are ready to map OCL pre- and postconditions to BOTL:

$$\begin{aligned} \Delta(\text{context } C :: M(\vec{p}) \text{ pre } \xi_{\text{pre}} \text{ post } \xi_{\text{post}}) = \\ \forall u_1 \in \tau_1, \dots, u_n \in \tau_n : \forall z \in C \text{ ref} : \forall m \in z.M \text{ ref} : \\ \mathbf{G}[m \text{ new} \wedge \xi_{\text{pre}}^{z,m,\vec{p}} \Rightarrow m \text{ alive} \cup (\text{term}(m) \wedge \delta_{z,m,\vec{p}}(\xi_{\text{post}}))] \end{aligned}$$

where $\text{term}(m) \equiv m \text{ alive} \wedge \mathbf{X}(m \text{ dead})$.

The expressions $\xi_{\text{pre}}^{z,m,\vec{p}}$ and ξ_{post} are embedded in a kind of “template” scheme. Intuitively, a pre- and postcondition holds if and only if for all invocations m of M executed by an object of the class C we have that: if the (extended) precondition holds at the moment of the method call (i.e., the method is new), then the postcondition holds when the method execution terminates. This must be true for all active objects of C and all possible executions of the method M . In other words, a pre- and postcondition is actually an invariant on method calls. Note that because of the assumption on the operational model (cf. page 46) the expression ξ_{post} is defined at the moment of the method termination.

The encoding of pre- and postconditions presented here reflects a *total correctness* criterion since it requires the termination of the method. In the operational model the use of paths satisfying the fairness constraint on the accept states can avoid situations where a method is unable to terminate because it never gets its turn¹². Alternatively, we could have chosen to demand only partial correctness avoiding to enforce termination. Replacing one approach with the other is not an involved exercise [79]. Although the OCL specification does not provide us with any guidelines on the partial or total interpretation of pre- and postcondition, also other OCL formalisations choose total correctness (for example cf. [97]).

Example 3.5.4. Suppose we want to translate the pre- and postcondition given in Section 3.4.1:

```
context Hotel :: CheckIn(g:Guest)
pre not guests→includes(g)
post guests→size = guests@pre→size + 1 and guests→includes(g).
```

Again, let us call the precondition ξ_{pre} and the postcondition ξ_{post} . Consider two logical variables z and m . The former will be instantiated with an object of

¹²This can be done when extracting the model out of the source code. Sections 4.4.2 and 5.6.2 show, for example, how to use accept states to enforce only fair path among a set of processes.

class `Hotel` and the latter with an occurrence of the method `checkIn`. Applying δ to ξ_{pre} yields:

$$\begin{aligned} \delta_{z,m,g}(\text{not } \overline{\text{guests} \rightarrow \text{includes}}(g)) &= \neg \overline{\delta_{z,m,g}(\text{guests} \rightarrow \text{includes}(g))} \\ &= \neg \overline{\text{includes}(z.\text{guests}, m.g)} \end{aligned}$$

where $\overline{\text{includes}}$ is a BOTL operation that, given a list w and an element l , returns `tt` if and only if the element l belongs to w . The extended precondition becomes:

$$\begin{aligned} \xi_{\text{pre}}^{z,m,g} &\equiv \overline{\neg \text{includes}(z.\text{guests}, m.g)} \wedge u_1 = \delta_{z,m,g}(\text{guests}) \\ &\equiv \overline{\neg \text{includes}(z.\text{guests}, m.g)} \wedge u_1 = z.\text{guests} \end{aligned}$$

After some calculations, the translation of the postcondition yields:

$$\begin{aligned} \delta_{z,m,g}(\xi_{\text{post}}) &= \delta_{z,m,g}(\text{guests} \rightarrow \text{size}) = \delta_{z,m,g}(\text{guests} @ \text{pre} \rightarrow \text{size}) + 1 \\ &\quad \wedge \delta_{z,m,g}(\text{guests} \rightarrow \overline{\text{includes}}(g)) \\ &= (\text{size}(z.\text{guests}) = \text{size}(u_1) + 1 \wedge \overline{\text{includes}(z.\text{guests}, m.g)}). \end{aligned}$$

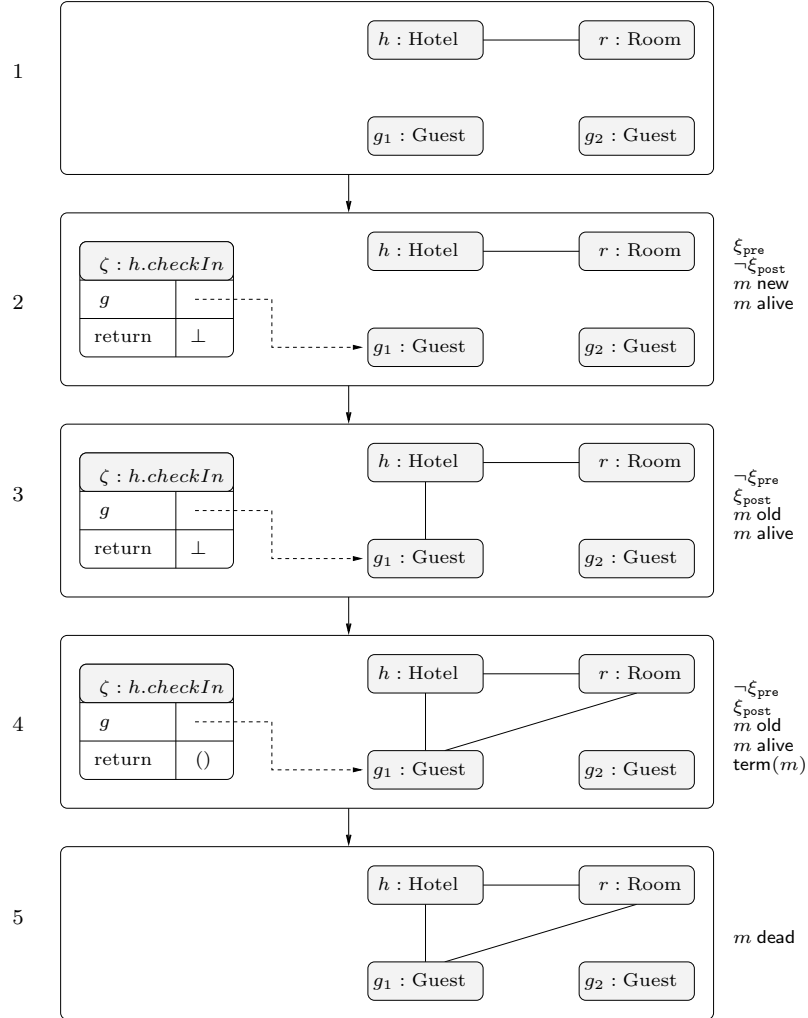
The translation of the pre- and postcondition now yields:

$$\begin{aligned} \forall u_1 : \forall z \in \text{Hotel} \text{ ref} : \forall m \in z.\text{checkIn} \text{ ref} : G[m \text{ new} \wedge \xi_{\text{pre}}^{z,m,g} \\ \Rightarrow m \text{ alive} \cup (\text{term}(m) \wedge \delta_{z,m,g}(\xi_{\text{post}}))]. \end{aligned}$$

Figure 3.4 describes the configurations of the transition system during the execution of the method `checkIn` and indicates how the validity of the pre- and postcondition changes. Inside the states it is possible to observe how the components O , E , ζ and γ evolve w.r.t the configuration. In the figure we have abstract from the value of the attributes since they are not relevant for this example. In configuration 1, object g_1 does not belong to the guests of h and the method has not been invoked yet, in fact the set of method calls E is empty. In this state it is not possible to talk about the validity of pre- and postcondition of the method call m . In configuration 2, the method ζ is newly born as a result of the method invocation. Therefore, in this state $m \text{ new}$ and the precondition ξ_{pre} are valid since g_1 is still not a guest of the hotel. In configuration 3, as a first step of the method execution, g_1 is inserted among z guests. Thus, ξ_{pre} does not hold anymore. However, from this state ξ_{post} becomes valid. m is not new anymore. In configuration 4, g_1 is assigned to room r and the method execution ends. The postcondition ξ_{post} still holds, and this is precisely the state in which it should hold. Finally in configuration 5, `checkIn` is deallocated and it does not exist anymore. Notice how in this example it becomes clear why the invariant in Example 3.4.2 does not hold during the execution of `checkIn`. \square

3.5.4 How to employ BOTL for OCL tools

Probably, a more pragmatic reason that may increase the significance of the translation defined in the previous section has to be sought in the still rather

Figure 3.4: Configurations during the execution of $checkIn(g_1)$.

limited tool support for OCL. Indeed, a number of developed CASE tools support OCL (the reader is referred to [65, 97] for a list of them). However, they are mostly limited to check the syntax or the types of the constraints. So far, the most advanced tools can validate constraints against single configurations of the system (also called snapshots) given manually by the user. This is obviously not enough, in fact, the validation of a constraint against snapshots may only give some hints on the adequateness of a constraint. However, this does not allow the user to conclude anything with respect to the validity of the

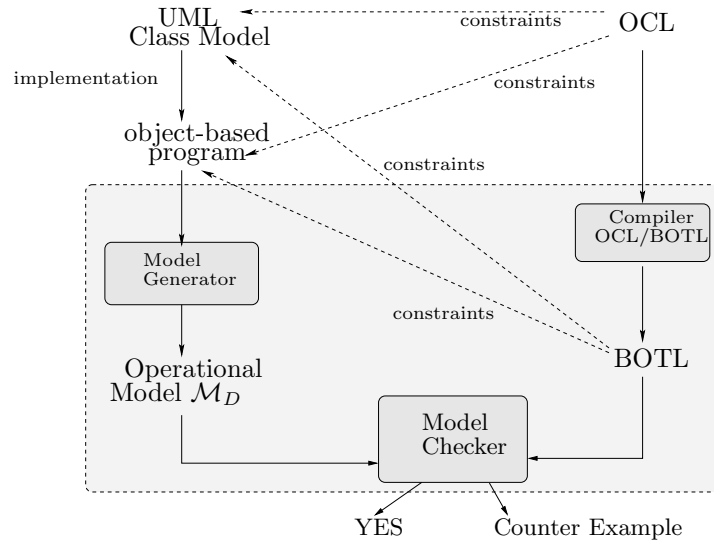


Figure 3.5: Architecture of a possible model checking tool for OCL

constraint itself. Experience has shown that due to the ever increasing complexity of systems, attempts to assess their correctness by engineering “rules of thumb” easily lead to wrong conclusions and may cause costly redesigns. By their very nature, invariants are required to be valid at *any* configuration of the system (except, as we have seen, those involving side effects). Similarly, pre- and postconditions must be valid at *any* invocation of the method they refer to. The current presence of the above mentioned tools attests a need to accomplish the development of more ambitious ones able to support automatic verification of systems with respect to OCL constraints by means of techniques such as model checking. This is certainly a challenge. The translation of (a subset of) OCL into BOTL that we have defined in this chapter, and the model checking algorithms for subsets of BOTL that we will define in the next chapters may be seen as a first step towards a future achievement of such tools. Figure 3.5 depicts how BOTL may be exploited to accomplish the construction of a model checker for object-based systems that uses OCL as a specification language. Given an object-based program, constraint can be written in OCL either at the UML class diagram level or directly at the source code level. The light gray dashed box can be seen as a back-end of a tool composed by a compiler OCL/BOTL, a model generator and a model checker. From a very pragmatic point of view, the software developer that knows OCL does not have to be aware of the parts inside the back-end. If implemented, the translating map provided in this chapter would represent the compiler OCL/BOTL. If implemented for the complete BOTL, the algorithms that we define in Chapter 4

and Chapter 5 would represent the model checker itself¹³. Finally, a model generator can be constructed from the definition of an operational semantics for the particular programming language considered (e.g. Java). This definition can be an extension of the operational semantics that we define in Section 4.4 and Section 5.6 for two simple imperative languages with some object-based feature.

3.6 Related work

In this section we describe some work related to BOTL. We start with a language that is very close to BOTL and therefore deserves some deeper comparison, afterwards we give a summary on the others works.

3.6.1 The Bandera Specification Language

The Bandera Specification Language (BSL) [33] is a formalism defined within the Bandera tool-set [32] developed at Kansas State University. BSL allows the specification of properties of multi-threaded Java programs. These properties are then model-checked by the Bandera system. It is interesting to see that although BSL and BOTL have been developed independently and simultaneously, they have several common aspects. In this section, we give a quick overview on BSL trying to compare it, whenever possible, with BOTL.

BSL is divided in two parts: the *assertion sub-language* and the *temporal sub-language*.

Assertion sub-language. Assertions provide an easy way to write constraints on data at particular points of the program. Assertions contain side-effect free Java expressions (e.g. tests on program variables). Three kinds of assertions can be defined in BSL:

- *preconditions* that, as usual, must hold in the first executable statement of a given method;
- *postconditions* that must hold immediately after the execution of a return statement in the method, or after the execution of the last statement of the method if no return statement exists;
- *location* assertions that must hold when the control is at a particular Java statement.

BOTL does not provide assertion facilities, at least in the sense of BSL location assertions. This is mostly due to the fact that BOTL is meant to be independent from the programming language, therefore, its properties cannot directly apply

¹³Using bounds on the number of objects and method calls it could be possible to use some existing LTL model checker.

to special points of the source code. However, as we have seen in Section 3.5, in BOTL it is possible to define pre- and postconditions.

Temporal sub-language. The temporal sub-language allows to specify properties along the execution of the program. The atomic propositions are given by *predicates* on program features like instance variables and static variables, control points such as method entry and exit. Predicates are embedded into temporal specifications that are not based on a particular temporal logic, but on a collection of *temporal specification patterns* that are automatically translated in the particular specification language accepted by the checkers used by Bandera (e.g. LTL for SPIN [61] or CTL for SMV [80]). Specification patterns are an important consequence of the fact that, in the design of BSL, special care was given by the authors to the challenge of making this formalism easily accessible to software developers normally not acquainted with temporal logic. The idea of the specification patterns is the following: the authors have observed that most of the software requirements, normally specified, follow just a few different patterns such as: *universal, absence, existence, response, precedence*. For example, the universal pattern requires its argument to be valid throughout the complete execution, therefore it essentially corresponds to an invariant. Whereas, the precedence pattern requires that a particular state/event is followed by another designated state/event. Thus, this pattern can be considered as the specification of a liveness property. For software developers the advantage to use pattern specification instead of the complete temporal logic is clear: it is easy to learn since most of the complexity of the temporal specification is predefined in a rather high-level manner. Furthermore, libraries of new patterns may be defined by the user extending thus those already existing.

The embedding of BOTL (static) expressions into BOTL temporal formulae follows clearly a different point of view. The user is free to define its own specification using the complete power of the temporal logic¹⁴. Of course, this may be difficult on the one hand, but very powerful on the other hand. Nevertheless, it is easy to imagine the definition of the BSL patterns for BOTL as it has been done for invariants and for pre- and postcondition. In this respect the encoding of OCL constraints given in Section 3.5 can be seen as a definition of temporal patterns for BOTL.

Class instance quantification. Unavoidably, BSL faces the problem related with the dynamic nature of objects and instances of method call. The language permits a quantification over allocated instances of a class. Without this mechanism it would be impossible to reason about allocated objects along program executions since program variables containing a reference to an object can change during the evolution of the system. The resulting quantification is similar to the one used in BOTL, although there exists a difference in the

¹⁴Note that, for expert users, in BSL there is still the possibility to write specifications in temporal logic.

way the quantification is dealt with. Namely, at the semantical level, BSL enforces an upper bound on the number of instances that can be allocated by a class during the program execution. This limitation is essentially imposed by the back-end model checkers used by Bandera. In fact, the semantics of BSL is defined in terms of Java state transition systems that, in turn, are a modification of state transition systems [78] used for the semantics of LTL and used by SPIN as well. Java transition systems are the cross product of a finite number of components that capture essential information about the execution of Java programs. Therefore, these kind of models are static, and in order to be finite, every component of the cross product must range over a finite domain. Thus, the number of allocations allowed for every class in the program must be bounded. Java transition systems can then be mapped back onto standard transition systems accepted by the model checkers used as a back-end of Bandera.

In BOTL we do not impose any restriction on the creation mechanism. In Chapter 4, we will study how to define appropriate models that overcome the problems related to the unboundedness of the state space obtained by object creation.

3.6.2 Others

On the formalisation of OCL. As mentioned before, the problem of formalising OCL has received more attention. Beside our translation into BOTL, a massive amount of work has been done in Richters PhD thesis [95], almost completely devoted to the formalisation of OCL, and in turn is based on previous work of the same author and Gogolla [52, 53, 96, 97]. Their formalisation is based on pure set theory.

Other formalisations that map OCL into some formal language are given by the following approaches: [55, 56] provide a mapping of OCL into the Larch specification language. In [10], OCL constraints are translated into expressions over graph rules. In [7], a mapping to algebraic specifications is defined.

OCL and temporal logics. Extensions of OCL with temporal operators have been proposed. The paper [93] extends OCL with operators of linear temporal logic but does not provide a formal semantics to its extension. A similar proposal is given in [30], but again without a formal foundation. As the original definition of BOTL [42], a CTL extension of OCL is given in [48] where the system behaviour is modelled by state-chart diagrams.

Strongly based on BOTL is the observational μ -calculus extension of OCL presented in [13]. Here, BOTL operational model are used. The resulting formalism, called $\mathcal{O}\mu(\text{OCL})$, is a two-level logic where OCL represents the lower level part and, as in BOTL, expresses the static properties of the system.

Finally, another OCL temporal extension with future and past operators of a linear temporal logic has been recently given in [114].

Object-based logics. Logics for reasoning about object-oriented systems have been mainly based on Hoare-style logics that concentrate on verifying pre- and postconditions and/or invariants [3, 39, 64, 92].

Temporal logics for object-oriented systems have been previously defined. Amongst others: [70] presents the specification language TROLL for the conceptual modelling of information systems. The formal semantics of TROLL is given in terms of a translation into a temporal logic. [102] proposes, in an axiomatic style, a temporal logic for reasoning about object classes and their instances. The logic supports two levels of reasoning: local reasoning related to a single object and global reasoning related to a community of objects. A modal logic for an object calculus is presented in [4].

A language for specifying LTL properties of Java programs somehow similar to the BSL was given in [66] and then used in a tool-set having as a back-end the model checker dSpin [41]. The language allows the definition of predicates at the Java source code level: in particular it addresses program variables as well as in critical control points such as method invocation and method exit. Moreover, this language allows a mechanism of quantification over integers whereas quantification over class instances is not provided.

Finally, Alloy [68] is a declarative object-modelling language that comes with a formal semantics. Alloy's underlying data are sets and relations, in this sense it is a relational language. Alcoa [69] is an analysing tool for Alloy specifications. However, since Alloy is not a decidable language, Alcoa is neither sound nor complete. It only conducts a search within a finite scope chosen by the user. The idea behind is that, according to the authors, if a counterexample does exist it will most likely be found within a small scope. In practise, the user cannot conclude anything if no counterexample is found.

4

Dynamic Allocation and Deallocation

4.1 Introduction

As we have seen in Section 2.3.2, allocation and deallocation are fundamental concepts in object-based systems. Arbitrarily many dynamic objects (*entities*) can be created (*allocated*) during the computation. Moreover, objects can be destroyed (*deallocated*) e.g., by a garbage collector. Similarly, instances of method calls (used by objects for interaction) are *entities* that are *allocated* (created) at the moment of the invocation and that are *deallocated* (destroyed) when the body of the method is completed.

In this chapter we aim to restrict BOTL defined in Chapter 3 to a core subset able to capture the birth and death of abstract entities as primitive concepts. Our attempt to formulate such a core logic, resulted in what we call *Allocational Temporal Logic* (*AllTL*) [45]. It has the following features:

- Entity variables x, y , interpreted by a mapping to the entities existing (i.e., *alive*) in a given state. The interpretation is *partial*: a variable not mapped onto an existing entity stands for an entity that has *died*.
- Entity equations $x = y$ (where x, y are entity variables), asserting that x and y refer to the same entity. This cannot hold if either x or y has died; hence the entity equations express a *partial equivalence* of entity variables (symmetric and transitive, but not reflexive).

- Entity quantification $\exists x.\phi$, which holds in a given state if ϕ holds for some interpretation of x , provided that x is alive.
- A predicate x *new* to express that the entity referred to by x is *fresh*, i.e., born.

In addition, $\mathcal{A}\ell\text{TL}$ has the standard LTL temporal operators (although, as for BOTL, a branching-time version with CTL temporal operators can be defined in a rather similar way, see page 44).

The logic is interpreted over so-called *High-level Allocational Büchi Automata* (HABA), which extend HD-automata [83, 84, 90] with a predicate for the *unboundedness* of (the number of entities in) a state, and with a (generalised) Büchi acceptance condition. As for HD-automata, a crucial point is that entity identity is *local* to a state. Correspondence between the identity of the same entity in two different states is ensured by a mechanism of re-mapping.

HABA can be used as finite-state abstraction of certain kinds of infinite-state systems. As an example, we define a small language whose main features are the allocation and deallocation of entities. Although the number of entities allocated by a program in this language can be unbounded, the operational semantics yields a finite HABA. This is a significant condition for the application of model checking.

Together with the logic $\mathcal{A}\ell\text{TL}$, the main contribution of this chapter is that the model-checking problem for $\mathcal{A}\ell\text{TL}$ is shown to be decidable on HABA. In particular, we present a tableau-based model-checking algorithm that decides whether or not a given $\mathcal{A}\ell\text{TL}$ -formula holds for a given HABA. Our algorithm extends the tableau-based algorithm for LTL [77]. To the best of our knowledge, this yields the first approach to effectively model-check birth and death behaviour of models with an unbounded number of entities¹. This is of particular interest — for what observed above — towards the verification of object-oriented systems in which the number of objects is typically not known in advance and may be even unbounded. Currently, in tools for model checking object-oriented systems (such as [32, 58]), dynamic creation of objects is only supported to a limited extent (the number of created objects must be bounded). Finally, reasoning about allocation and deallocation of fresh entities is relevant not only for object-based systems, but also in relation to *privacy* and *locality* as discussed in, e.g., [2, 17, 21, 82].

This chapter is organised as follows. First of all we define the logic (Section 4.2) and its automata (Section 4.3). The simple imperative language, featuring statements for the allocation and deallocation of entities, with an operational semantics in terms of HABA is given in Section 4.4. The definition of the model checking algorithm for $\mathcal{A}\ell\text{TL}$ formulae against HABAs is given

¹Recently other attempts have been proposed in the literature [113]. However, these techniques are only sound but not complete. The reader is referred to Section 4.6 for a broader overview of the literature.

in Section 4.5. The chapter is concluded with a discussion about related work in Section 4.6.

4.2 Allocational temporal logic

4.2.1 Syntax

For the definition of the logic, in the following we assume the existence of:

- the countable universe of logical variables $LVAR$, ranged over by x, y, z, \dots
- a countable universe of entities Ent , ranged over by e, e', e_1, \dots

Allocational Temporal Logic ($\mathcal{All}TL$) is an extension of propositional LTL [91] that allows existential quantification over logical variables that can denote entities, or may be undefined. It is a subset of BOTL defined in Chapter 3 that focuses on the birth and death of entities. For $x \in LVAR$, the syntax of $\mathcal{All}TL$ is defined by the following grammar:

$$\phi ::= x \text{ new} \mid x = x \mid x \text{ alive} \mid \exists x. \phi \mid \neg \phi \mid \phi \vee \phi \mid X\phi \mid \phi \text{ U } \phi.$$

Let us quickly recapitulate the intuitive meaning of the operators². Formula $x \text{ new}$ holds if the entity denoted by x is fresh in the current state, i.e., the entity denoted by x did not exist in the previous state. Formula $x = y$ holds if variables x and y denote the same entity in the current state; $x = x$ is violated if x is undefined, i.e., if x does not denote any current entity. $x \text{ alive}$ is true if x denotes an entity currently alive in the state. $\exists x. \phi$ is valid in the current state if there exists an entity for which ϕ holds if assigned to x . In contrast to BOTL that distinguishes among different types, in $\mathcal{All}TL$ the only universe of values is Ent . Therefore in the syntax of the existential quantification, we drop the specification of the type. In the context of BOTL, Ent corresponds to the references of objects and method calls. For the purpose of dynamic allocation and deallocation there is no reason to distinguish between them, therefore it is convenient to consider Ent as single abstract domain³. Negation and disjunction are the standard operators of classical propositional logic, while X (next) and U (until) are the standard LTL operators.

Abbreviations. We use the standard abbreviations from propositional and temporal logic, as well as those introduced for BOTL (cf. Table 3.1). Others more typical of $\mathcal{All}TL$ are listed in Table 4.1.

Example 4.2.1. Some example properties concerning dynamic allocation and deallocation expressible in $\mathcal{All}TL$ are reported in Table 4.2. \square

²For a more comprehensive illustration of the informal interpretation the reader is referred to the discussion related to BOTL given in Chapter 3

³It is obvious that element of Ent may be instantiated not only to objects and method calls, but to any other concept for which we want to reason about allocation and deallocation, e.g., resources, memory, channels, etc.

• $x \neq y$	for	$\neg(x = y)$
• x dead	for	$\neg(x \text{ alive})$
• x old	for	$x \text{ alive} \wedge \neg(x \text{ new})$
• $\forall x.\phi$	for	$\neg\exists x.\neg\phi$
• tt	for	$\forall x.(x \text{ alive})$
• ff	for	$\neg\text{tt}$

Table 4.1: Typical abbreviations of $\mathcal{A}ll\text{TTL}$.

New entities are always available	$G(\exists x.x \text{ new})$
The number of entities that are alive never exceeds 2	$G(\forall x.\forall y.\forall z.(x = y \vee x = z \vee y = z))$
The number of alive entities will never be less than 2	$G(\exists x.\exists y.(x \neq y))$
A fresh entity will always eventually be allocated	$GF(\exists x.x \text{ new})$
The number of entities that are continuously alive grows unboundedly	$G((F\exists x.x \text{ new}) \wedge \forall x.X(x \text{ alive}))$
Before x is deallocated, two new entities will be allocated	$x \text{ alive} \cup \exists y.(y \text{ new} \wedge (x \text{ alive} \cup \exists z.(z \text{ new} \wedge y \neq z \wedge x \text{ alive})))$
Every entity in the current state will be eventually deallocated	$\forall x.F(x \text{ dead})$
A deallocated entity cannot be reallocated (this will be a tautology)	$G(x \text{ dead} \Rightarrow X(x \text{ dead}))$
An entity that is identified now is no longer new in the next state (tautology)	$X(x \text{ dead} \vee x \text{ old})$

Table 4.2: Some example properties expressed in $\mathcal{A}ll\text{TTL}$.

4.2.2 Semantics

$\mathcal{A}ll\text{TTL}$ formulae are interpreted over infinite sequences of sets of entities. Every set contains the live entities in a particular moment in time (i.e., in a state).

Definition 4.2.2. An *allocation sequence* σ is an infinite sequence of sets of entities $E_0E_1E_2\cdots$ where $E_i \subseteq \text{Ent}$, for $i \in \mathbb{N}$.

Let $\sigma^i = E_iE_{i+1}\cdots$. For given σ , E_i^σ denotes the set of entities in the i -th state of σ . The semantics of $\mathcal{A}ll\text{TTL}$ -formulae is defined by the satisfaction relation $\sigma, N, \theta \models \phi$ where σ is an allocation sequence, $N \subseteq E_0^\sigma$ is the set of entities that is initially new, and $\theta : \text{LVAR} \rightarrow \text{Ent}$ is a partial valuation of the free variables in ϕ . Let N_i^σ denote the set of new entities in state i , defined by,

$$N_i^\sigma = \begin{cases} N & \text{if } i = 0 \\ E_i^\sigma \setminus E_{i-1}^\sigma & \text{otherwise.} \end{cases} \quad (4.1)$$

Similarly, let $\theta_i^\sigma : \text{LVAR} \rightarrow \text{Ent}$ denote the valuation at state i , i.e., the valuation that is undefined for the variables that θ maps outside E_i^σ , and coincides

with θ otherwise. Formally,

$$\theta_i^\sigma(x) = \begin{cases} \theta(x) & \text{if } \forall k \leq i : \theta(x) \in E_k^\sigma \\ \text{undefined} & \text{otherwise.} \end{cases} \quad (4.2)$$

The previous definition imposes that if x denotes $e \in E_i$ then e must have been continuously alive (*non-resurrection* condition for variable denotations). The condition avoids that contradictions like $\exists x.X(x \text{ dead} \wedge Xx \text{ alive})$ are fulfilled. Note that once a logical variable is mapped to an entity, then this association remains along σ unless the entity dies, i.e., is deallocated. The association remains undefined even if the entity re-occurs again in a state later on (see Section 4.2.4 for a more detailed discussion on these concepts).

Proposition 4.2.3. $\theta_{i+1}^\sigma = \theta_i^\sigma \upharpoonright E_{i+1}$.

The satisfaction relation \models is defined as follows:

$$\begin{aligned} \sigma, N, \theta \models x \text{ new} & \text{ iff } x \in \text{dom}(\theta) \text{ and } \theta(x) \in N \\ \sigma, N, \theta \models x = y & \text{ iff } x, y \in \text{dom}(\theta) \text{ and } \theta(x) = \theta(y) \\ \sigma, N, \theta \models x \text{ alive} & \text{ iff } x \in \text{dom}(\theta) \\ \sigma, N, \theta \models \exists x.\phi & \text{ iff } \exists e \in E_0^\sigma : \sigma, N, \theta\{e/x\} \models \phi \\ \sigma, N, \theta \models \neg\phi & \text{ iff } \sigma, N, \theta \not\models \phi \\ \sigma, N, \theta \models \phi \vee \psi & \text{ iff either } \sigma, N, \theta \models \phi \text{ or } \sigma, N, \theta \models \psi \\ \sigma, N, \theta \models X\phi & \text{ iff } \sigma^1, N_1^\sigma, \theta_1^\sigma \models \phi \\ \sigma, N, \theta \models \phi \cup \psi & \text{ iff } \exists i : (\sigma^i, N_i^\sigma, \theta_i^\sigma \models \psi \text{ and } \forall j < i : \sigma^j, N_j^\sigma, \theta_j^\sigma \models \phi). \end{aligned}$$

Here, $\theta\{e/x\}$ is defined as usual, i.e., $\theta\{e/x\}(x) = e$ and $\theta\{e/x\}(y) = \theta(y)$ for $y \neq x$.

Proposition 4.2.4. The following formulae are tautologies:

- $x = y \Rightarrow y = x$
- $x \text{ new} \Rightarrow x \text{ alive}$
- $x = y \Rightarrow X(x \text{ dead} \vee x = y)$
- $\forall x.x \text{ alive}$
- $X(x \text{ dead} \vee x \text{ old})$
- $(x = y \wedge y = z) \Rightarrow x = z$
- $(x = y \wedge x \text{ new}) \Rightarrow y \text{ new}$
- $X(x = y) \Rightarrow x = y$
- $G(x \text{ dead} \Rightarrow X(x \text{ dead}))$
- $x \text{ alive} \Leftrightarrow x = x$.

Proof. Straightforward application of the $\mathcal{A}\ell\ell\text{TL}$ semantics. As an example we prove only the case $x \text{ alive} \Leftrightarrow x = x$, the other cases are similar and omitted here. Let σ be an arbitrary allocation sequence and N, θ defined as above. Then $\sigma, N, \theta \models x \text{ alive} \Rightarrow x \in \text{dom}(\theta) \Rightarrow \theta(x) = \theta(x) \Rightarrow \sigma, N, \theta \models x = x$. Vice-versa $\sigma, N, \theta \models x = x \Rightarrow x \in \text{dom}(\theta) \wedge \theta(x) = \theta(x) \Rightarrow \sigma, N, \theta \models x \text{ alive}$. \square

Note that from the previous proposition it becomes clear that $x \text{ alive}$ can be introduced as syntactic sugar for $x = x$. Here we prefer to define it as primitive since it seems to be more intuitive. For the other approach we refer the reader to [44]. Another straightforward implication of the $\mathcal{A}\ell\ell\text{TL}$ semantics is that the universal quantification does not “commute” with the next operator. In fact, the quantification is over a dynamic domain that may change from state to state. Therefore we have the following:

Proposition 4.2.5. For all $\mathcal{A}\ell\ell\text{TL}$ -formulae ϕ : $\neg(\forall x.X\phi \Leftrightarrow X\forall x.\phi)$.

4.2.3 Folded allocation sequences

In order to come to a finite representation of allocation sequences there are several difficulties to overcome: the sequences are infinite themselves, and in general they range over an infinite set of entities (i.e., $|\bigcup_{i \in \mathbb{N}} E_i| = \omega$). The former problem can be solved, by generating allocation sequences as runs of a Büchi automaton; the latter problem requires a change in representing the allocation sequences. This change is based on the following observations:

- $\mathcal{A}\ell\ell\text{TL}$ -formulae cannot address entities directly, but only through logical variables. The choice of representation for the entities is therefore irrelevant from the point of view of the $\mathcal{A}\ell\ell\text{TL}$ semantics.
- $\mathcal{A}\ell\ell\text{TL}$ -formulae allow the direct comparison of entities only within a state. The interpretation will not change if we allow for reallocating entities from a state to the next state, as long as this is done injectively so that distinct entities remain distinguished.

These considerations bring us to the following definition.

Definition 4.2.6. For $E, E' \subseteq \text{Ent}$,

- A *reallocation* λ from E to E' is a partial injective function $\lambda : E \rightarrow E'$.
- A *folded allocation sequence* is an infinite alternating sequence

$$E_0 \lambda_0 E_1 \lambda_1 E_2 \lambda_2 \dots$$

where λ_i is a reallocation from E_i to E_{i+1} for $i \geq 0$.

Thus, entity e is considered to be deallocated if $e \notin \text{dom}(\lambda)$. We write λ_i^σ for the reallocation function of σ in state i . Note that for folded allocation sequence σ with associated initial set N and valuation θ ,

$$N_i^\sigma = \begin{cases} N & \text{if } i = 0 \\ E_i^\sigma \setminus \text{cod}(\lambda_{i-1}^\sigma) & \text{otherwise.} \end{cases} \quad (4.3)$$

Similarly,

$$\theta_i^\sigma = \begin{cases} \theta & \text{if } i = 0 \\ \lambda_{i-1}^\sigma \circ \theta_{i-1}^\sigma & \text{otherwise.} \end{cases} \quad (4.4)$$

(Hence $\theta_i^\sigma = \lambda_{i-1}^\sigma \circ \dots \circ \lambda_0^\sigma \circ \theta$ for all $i \in \mathbb{N}$.) Using these adapted definitions of N and θ , the same definition of satisfaction relation \models holds for folded allocation sequences. In the following we indicate by \models_u and \models_f the semantics of $\mathcal{A}\ell\ell\text{TL}$ in the unfolded⁴ case and in the folded case, respectively.

⁴In the following we will sometimes use *unfolded* allocation sequences in order to stress the difference w.r.t folded allocation sequences. When the context is clear and no ambiguities can arise, we will skip the adjectives folded or unfolded.

4.2.4 Relating unfolded and folded allocation sequences

At this point, after having defined two different models for \mathcal{AllTL} , namely, unfolded and folded allocation sequences, it is interesting to study their relation. For allocation sequence $\sigma = E_0E_1E_2\cdots$, let $id(\sigma)$ be the folded allocation sequence defined by $E_0id_0E_1id_1E_2id_2\cdots$ where $id_i = id \upharpoonright (E_i \cap E_{i+1})$. We have the following straightforward fact:

Proposition 4.2.7. For any \mathcal{AllTL} -formula ϕ we have

$$\sigma, N, \theta \models_u \phi \text{ iff } id(\sigma), N, \theta \models_f \phi.$$

Neither folded nor unfolded allocation sequences are fully abstract with respect to the validity of \mathcal{AllTL} -formulae, as several of such sequences may satisfy the same formulae. We therefore consider folded allocation sequences *modulo isomorphism*.

Definition 4.2.8.

- Two folded allocation sequences σ_1 and σ_2 are *isomorphic* ($\sigma_1 \cong \sigma_2$) if there exists an indexed family of bijections $(h_i)_{i \in \mathbb{N}}$ with $h_i : E_i^{\sigma_1} \rightarrow E_i^{\sigma_2}$ such that $\lambda_i^{\sigma_2} \circ h_i = h_{i+1} \circ \lambda_i^{\sigma_1}$ (i.e., the h_i 's are consistent with the reallocations).
- Two allocation triples $(\sigma_1, N_1, \theta_1)$, $(\sigma_2, N_2, \theta_2)$ are *isomorphic* (written $(\sigma_1, N_1, \theta_1) \cong (\sigma_2, N_2, \theta_2)$) if $\text{dom}(\theta_1) = \text{dom}(\theta_2)$, $\sigma_1 \cong \sigma_2$, and $N_2 = h_0(N_1)$ and $\theta_2 = h_0 \circ \theta_1$.

Isomorphic allocation triples satisfy the same set of \mathcal{AllTL} -formulae, as stated by the following property.

Proposition 4.2.9. For \mathcal{AllTL} -formula ϕ and folded allocation sequences σ, σ' :

$$(\sigma, N, \theta) \cong (\sigma', N', \theta') \Rightarrow (\sigma, N, \theta \models_f \phi \text{ iff } \sigma', N', \theta' \models_f \phi).$$

Proof. By a straightforward induction on the structure of ϕ . □

Unfolded and folded allocation sequences are related by \sqsubseteq^{fold} .

Definition 4.2.10.

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f) \text{ iff } (id(\sigma_u), N_u, \theta_u) \cong (\sigma_f, N_f, \theta_f).$$

Allocation triples in the relation \sqsubseteq^{fold} enjoy the property to satisfy the same set of \mathcal{AllTL} formulae.

Proposition 4.2.11. For \mathcal{AllTL} -formula ϕ , folded allocation sequence σ_f and allocation sequence σ_u :

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f) \Rightarrow (\sigma_u, N_u, \theta_u \models_u \phi \text{ iff } \sigma_f, N_f, \theta_f \models_f \phi).$$



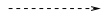

Symbol	Meaning
	old entity
	new entity
	reallocation
	single state of an allocation sequence

Figure 4.1: Visual notation for (folded) allocation sequences.

Proof. According to the definition $(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f)$ if and only if $(id(\sigma_u), N_u, \theta_u) \cong (\sigma_f, N_f, \theta_f)$. By Proposition 4.2.9, $id(\sigma_u), N_u, \theta_u \models_f \phi$ if and only if $\sigma_f, N_f, \theta_f \models_f \phi$. From Proposition 4.2.7, it now follows that $\sigma_u, N_u, \theta_u \models_u \phi$ if and only if $\sigma_f, N_f, \theta_f \models_f \phi$. \square

The following results show that allocation sequences and folded ones can both be used as models of \mathcal{ALLTL} -formulae:

Proposition 4.2.12. For an arbitrary folded allocation sequence σ_f and unfolded allocation sequence σ_u :

1. For every $(\sigma_u, N_u, \theta_u)$ there exists a $(\sigma_f, N_f, \theta_f)$ such that

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f)$$

2. For every $(\sigma_f, N_f, \theta_f)$ there exists a $(\sigma_u, N_u, \theta_u)$ such that

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f).$$

Proof. See Appendix A.1. \square

Visual notation. Throughout this chapter, for allocation sequences we use the visual notation indicated in Figure 4.1. Old (live) entities are depicted as circles. New (live) entities are denoted by patterned circles. Reallocations are indicated as dashed arrows between entities. A state of the (folded) allocation sequence (i.e. a set of entities) is a dashed box that encloses the live entities.

Example 4.2.13. Let $\phi \equiv \mathbb{G}(\exists x.x \text{ new})$ and $E_1 = \{e_1\}$, $E_{12} = \{e_1, e_2\}$, $E_{23} = \{e_2, e_3\}$. Consider the allocation sequences depicted in Figure 4.2 (top). If the initial set of new entities $N = E_1$, the unfolded allocation sequence

$$\sigma = E_1(E_{12}E_{23})^\omega$$

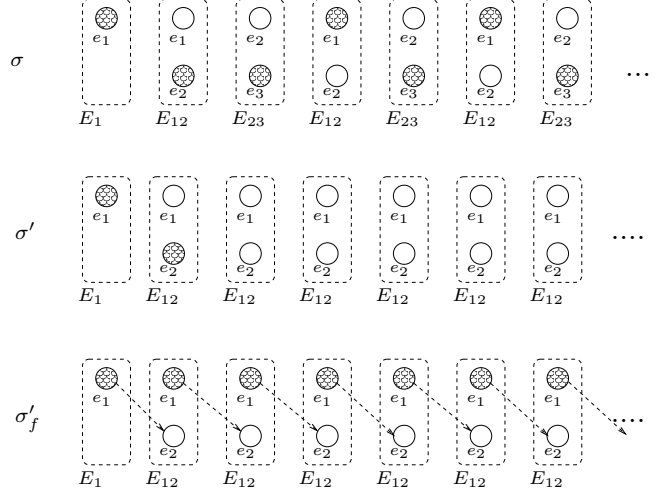


Figure 4.2: Folded and unfolded allocation sequences.

satisfies ϕ for any θ , whereas (cf. Figure 4.2 middle row)

$$\sigma' = E_1 E_{12}^\omega$$

does not, since after the second state, entities in E_{12} are continuously old. Let $\lambda_1 : E_1 \rightarrow E_{12}$ and $\lambda_2 : E_{12} \rightarrow E_{12}$ be two reallocations such that $\lambda_1(e_1) = e_2$, $\lambda_2(e_1) = e_2$ and $\lambda_2(e_2)$ is undefined. The folded allocation sequence

$$\sigma'_f = E_1 \lambda_1 (E_{12} \lambda_2)^\omega$$

has the same sets of entities as σ' (cf. Figure 4.2 bottom row). Nevertheless, σ'_f satisfies ϕ . In fact, by the reallocations λ_1 and λ_2 , the entity e_1 is new ($e_1 \notin \text{cod}(\lambda_1) \cup \text{cod}(\lambda_2)$) in every state. Moreover, in σ'_f the entity e_2 dies at every step while in σ' e_2 is continuously alive. Thus, the formula $\mathbb{G}(\forall x. (\text{XX}x \text{ dead}))$ is satisfied by σ'_f but not by σ' . \square

The metempsychosis metaphor. The difference in the use of entity identity in unfolded and folded allocation sequences can be explained in a more intuitive way by what we may call the *metempsychosis metaphor*. Quoting an adapted version⁵ from a 1911 Encyclopedia Britannica,

“*Metempsychosis* (or transmigration of the soul) is the doctrine that at death the soul passes into another living creature, human, animal, or even a plant.”

⁵<http://www.wikipedia.org>

In this doctrine, it is believed that a living creature (a being) has a soul connected in some way to the material body.

Having clarified these premises, the idea of the metempsychosis metaphor is to identify an entity as a *soul* that can be connected to a certain living creature which corresponds instead with what is denoted ideally by a logical variable. We have a countable number of souls (in the universe *Ent*) and at every moment only a finite number of them is associated with a respective living being.

Unfolded allocation sequences use metempsychosis in the strict sense. The soul (entity) is associated with a single living creature for its entire life. Since the living creature is born, till it dies, it is connected to the same soul. Once the being dies, its soul can reappear again in the world, but then in another living creature that is therefore born in that very precise moment. For example, consider the allocation sequence σ in Figure 4.2. Entity e_1 is connected with a being that is alive in the first and in the second state, but in the third state, e_1 's associated living creature dies. In the fourth state, e_1 reappears, but this time it has migrated (reincarnated) in a different being (newly born) than the previous one which is now, instead, definitely dead. This explains why a variable interpreted into e_1 in the first state is considered deallocated for ever in the third state even if e_1 reappears later. In terms of this metaphor, a logical variable x denotes the living creature connected to the entity $\theta(x)$ where x is interpreted. This latter observation may help to clarify also the non-resurrection condition for variable denotations imposed by (4.2) on θ (see page 73)⁶. For a living creature the presence of its own soul is essential to ensure the continuity of its life. Hence, in unfolded allocation sequences identity of entities plays a relevant role in the life of a being.

Using the same metaphor, we can observe that in the context of folded allocation sequences, transmigration of the soul applies as well, but on a generalised level. In particular, the precondition on the death of the living creature connected to the soul is now released. A soul can depart from its connected material body even if this is still alive. However, the implicit postulate that a being is associated to a soul in every moment of its life is still valid⁷. Therefore, in folded allocation sequences the separation between the body and soul is complete. For an individual, the continuation of life is ensured completely by the reallocations. The latter decides for every living creature if it will have a soul in the next state and, if so, which one. It is completely irrelevant which particular soul is connected to the body from time to time, as long as, there is one (i.e., the entity identity is inessential). And this is in contrast — as we have just seen — with the case of unfolded allocation sequences, where the life of a being is determined by the continuous presence of the same soul.

⁶Resurrection refers to the material body, not to the soul. It would be inconsistent to think that the entity resurrect since it is equated to a soul.

⁷To the best of our knowledge, we are not aware of any religion and/or philosophy in the ancient as well as in the contemporary world that actually believes in such system. Some sporadic cases of soul exchange between individuals, though, seems to occur in some science fiction novels.

4.3 Automata for dynamic allocation and deallocation

In Section 2.1.1 we introduced the notion of (generalised) Büchi automata [16]. In this section, we define two extensions of this formalism, namely: Allocational Büchi automata (ABA) that generate allocation sequences and High-level Allocational Büchi automata (HABA) that generate folded allocation sequences. Typically ABA are infinite state, whereas for the cases that we are interested in, the corresponding HABA is finite state. The HABA model is inspired by history-dependent (HD) automata [83, 84, 90]. The precise relationship between ABA and HABA is investigated in Section 4.3.3.

4.3.1 Allocational Büchi Automata

ABA are basically generalised Büchi automata where to each state a set of entities is associated. These entities, in turn, serve as valuation of logical variables used on $\mathcal{A}\ell\ell\text{TTL}$ -formulae.

Definition 4.3.1. An *Allocational Büchi Automaton* (ABA) \mathcal{A} is a tuple $\langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$, with

- $X \subseteq \text{LVAR}$ a finite set of logical variables;
- Q a (possibly infinite) set of states;
- $E : Q \rightarrow 2^{\text{Ent}}$ a function yielding for each state q a finite set E_q of entities;
- $\rightarrow \subseteq Q \times Q$ a transition relation;
- $I : Q \rightarrow 2^{\text{Ent}} \times (X \rightarrow \text{Ent})$ a partial function yielding for every *initial state* $q \in \text{dom}(I)$ an *initial valuation* (N, θ) , where $N \subseteq E_q$ is a finite set of entities, and $\theta : X \rightarrow E_q$ is a partial valuation of the variables in X ;
- $\mathcal{F} \subseteq 2^Q$ a set of sets of accept states.

Notational conventions: we write $q \rightarrow q'$ for $(q, q') \in \rightarrow$. We adopt the generalised Büchi acceptance condition, i.e. $\rho = q_0 q_1 q_2 \dots$ is a *run* of ABA \mathcal{A} if $q_i \rightarrow q_{i+1}$ for all $i \in \mathbb{N}$ and $|\{i | q_i \in F\}| = \omega$ for all $F \in \mathcal{F}$. Let $\text{runs}(\mathcal{A})$ denote the set of runs of \mathcal{A} . Run $\rho = q_0 q_1 q_2 \dots$ is said to *accept* the (unfolded) allocation sequence $\sigma = E_{q_0} E_{q_1} E_{q_2} \dots$. The initial valuation (N, θ) associated to an initial state facilitates the correspondence between models for $\mathcal{A}\ell\ell\text{TTL}$ -formulae and ABA runs. The language of an ABA \mathcal{A} is:

$$\mathcal{L}(\mathcal{A}) = \{(\sigma, N, \theta) | \exists \rho = q_0 q_1 \dots \in \text{runs}(\mathcal{A}) : \rho \text{ accepts } \sigma \wedge I(q_0) = (N, \theta)\}. \quad (4.5)$$

Example 4.3.2. Figure 4.3 depicts an infinite-state ABA \mathcal{A} for $X = \emptyset$ with initial state q_1 for which $I(q_1) = (\{e_1\}, \emptyset)$. In initial states, patterned circles denote new entities (as for allocation sequences). q_2, q_3, \dots are accept states

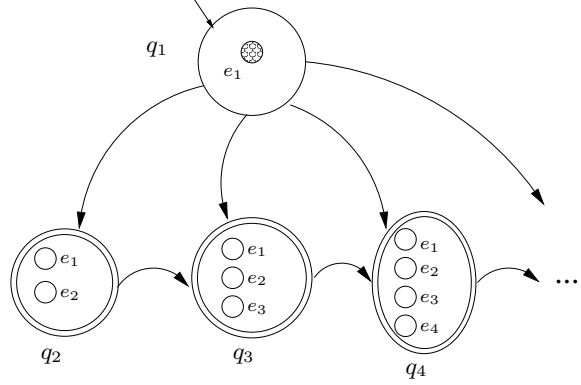


Figure 4.3: An example ABA

(double circles). For simplicity, we assume $|\mathcal{F}| = 1$. \mathcal{A} accepts allocation sequences that start with a single (new) entity and then move to a state where an arbitrary number of new entities is created. From that point on, at every step a single new entity is added, therefore the formula $\mathsf{G}(\exists x.x \text{ new})$ is satisfied by every triple $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{A})$. Note that entities are never deallocated, thus the formula $\mathsf{F}(\exists x.(\mathsf{F}x \text{ dead}))$ is not satisfied by any of these triples in $\mathcal{L}(\mathcal{A})$. \square

4.3.2 High-level Allocational Büchi Automata

Most of the ABAs describing interesting behaviours are infinite. The reason clearly is that the model although rather intuitive is also quite simple. It is, however, possible to improve the representation given by ABAs, by adding some ingredient that in many cases permits to finitely represent behaviours that are otherwise infinite, such as the allocation of an unbounded number of entities. This is done in the formalism introduced in this section.

Let $\infty \notin \text{Ent}$ be a special, distinguished entity, called *black hole*. Its role will become clear later on. We denote $E^\infty = E \cup \{\infty\}$ for arbitrary $E \subseteq \text{Ent}$.

Definition 4.3.3. For $E, E_1 \subseteq \text{Ent}$, an ∞ -reallocation is a partial function $\lambda : E^\infty \rightarrow E_1^\infty$ such that

- $\lambda(e) = \lambda(e') \neq \infty \Rightarrow e = e'$ for all $e, e' \in E$ and
- $\infty \in \text{dom}(\lambda) \Rightarrow \lambda(\infty) = \infty$.

That is, λ is injective when mapping away from ∞ and preserves ∞ .

Definition 4.3.4. A *High-level ABA* (HABA) \mathcal{H} is a tuple $\langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ with X, Q, I, \mathcal{F} as in Def. 4.3.1, and

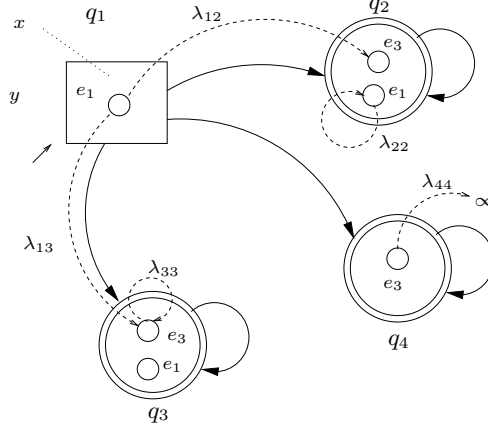


Figure 4.4: An example HABA

- $E : Q \rightarrow 2^{Ent} \times \mathbb{B}$, a function that associates to each state $q \in Q$ a finite set E_q of entities and a predicate B_q that, if false, expresses the existence of an unbounded number of entities in q not explicitly modelled by E_q ;
- $\rightarrow \subseteq Q \times (Ent^\infty \rightarrow Ent^\infty) \times Q$, such that for $q \rightarrow_\lambda q'$, λ is an ∞ -reallocation from E_q^∞ to $E_{q'}^\infty$ with
 - $\infty \in \text{dom}(\lambda)$ iff $E_q = (E, \text{ff})$ and $E_{q'} = (E', \text{ff})$, and
 - $\infty \in \text{cod}(\lambda) \Rightarrow E_{q'} = (E', \text{ff})$.

A HABA is a *symbolic* representation of a (possibly infinite) ABA. Predicate B_q holds in state q if and only if the number of entities in q is bounded. An unbounded state q (denoted $\lfloor q \rfloor$), possesses the distinguished entity ∞ that represents all entities that may be added to q (*imploded entities*). High-level state q thus represents all possible concrete states obtained from q by adding a finite number of entities to E_q . If a transition to state q' maps (implodes) entities into the black hole ∞ , these entities cannot be distinguished anymore from there on. We call this phenomenon *black hole abstraction*. Moreover, if $q \rightarrow_\lambda q'$, entities in the black hole are either preserved (if $\lfloor q' \rfloor$), or are destroyed (if $\lceil q' \rceil$). The black hole thus allows to abstract from the identity of entities when, for example, these are not relevant anymore. Note that $\infty \notin Ent$ implies $\infty \notin \text{cod}(\theta)$ for all $(q, N, \theta) \in I$.

Example 4.3.5. The visual notation for HABAs is reported in Figure 4.5. An example HABA is depicted in Figure 4.4 where $X = \{x, y\}$. In the initial state q_1 , variable x denotes (old) entity e_1 , while y is undefined. Entity e_3 in state q_2 represents the same entity as e_1 in q_1 , while e_1 (in q_2) represents a new entity. The set of accept states is $\{\{q_2, q_3, q_4\}\}$.



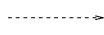


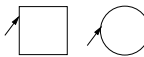

Symbol	Meaning
	old entity
	initial new entity
	∞ -reallocation
	Initial valuation
	Bounded/Unbounded state
	Bounded/Unbounded initial state
	Bounded/Unbounded accept state

Figure 4.5: Visual notation for HABAs

Run $q_1 \lambda_{12} (q_2 \lambda_{22})^\omega$ generates sequences where the initial entity dies after the second state, while the new entity created in the second state will be alive forever. Run $q_1 \lambda_{13} (q_3 \lambda_{33})^\omega$ generates sequences where the initial entity will be alive forever, and in each state a new entity is created. This new entity will die in the next state. Finally, run $q_1 \lambda_{14} (q_4 \lambda_{44})^\omega$ (the reallocation λ_{14} is not depicted since it is empty) generates sequences where the entity in the initial state dies immediately. Once q_4 is reached, a new entity e_3 is created at every step. Hence, in this run the number of entities grows unboundedly. \square

In the above example, we remarked that the runs of a HABA generate folded allocation sequences that compose the language of the automaton. For ABA, the correspondence between runs and sequences is trivial, since it corresponds to take the sequence of sets of entities corresponding to the run (cf. (4.5) page 79). For HABA however, the correspondence run/sequences is not at all straightforward, mostly due to reallocations and black-hole abstraction. The next definition makes this concept precise.

Definition 4.3.6. A run $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ of HABA $\mathcal{H} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ generates an allocation triple (σ, N, θ) , where $\sigma = E_0 \lambda_0^\sigma E_1 \lambda_1^\sigma \dots$ is a folded allocation sequence, if there is a *generator*, i.e., a family of functions $\phi_i : E_i \rightarrow E_{q_i}^\infty$ satisfying for all $i \geq 0$:

1. $\forall e, e' \in E_i. (\phi_i(e) = \phi_i(e') \neq \infty \Rightarrow e = e')$
2. $E_{q_i} \subseteq \text{cod}(\phi_i)$
3. $[q_i] \Rightarrow \infty \notin \text{cod}(\phi_i)$

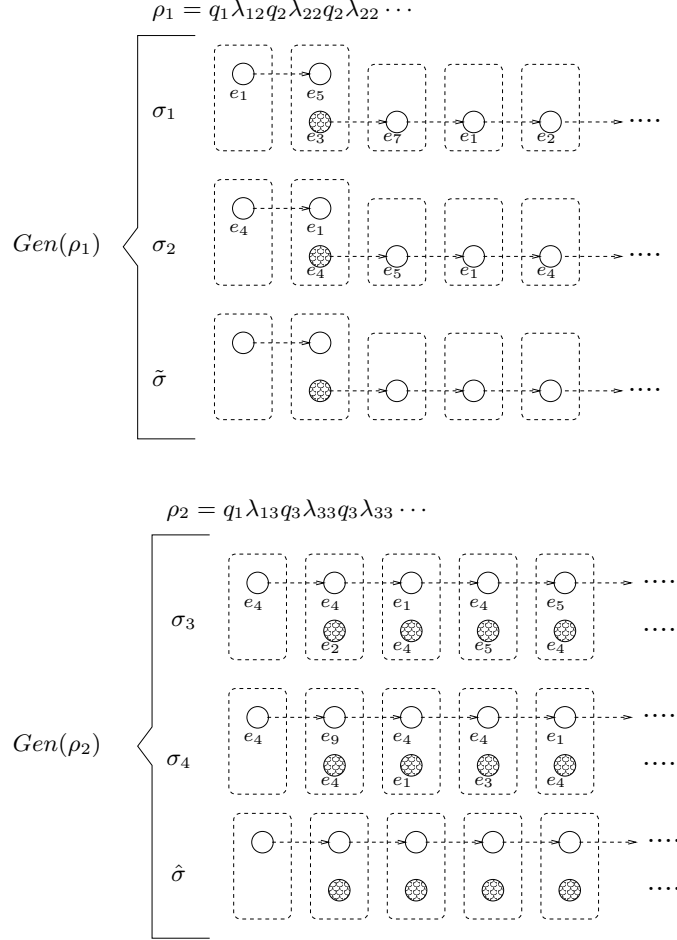


Figure 4.6: Example folded allocation sequences generated by runs ρ_1 and ρ_2 of the HABA of Fig. 4.4.

4. $\lambda_i \circ \phi_i = \phi_{i+1} \circ \lambda_i^\sigma$
5. $\forall e \in E_{i+1}. (\phi_{i+1}(e) = \infty \Rightarrow e \in \text{cod}(\lambda_i^\sigma))$
6. $I(q_0) = (\phi_0(N), \phi_0 \circ \theta)$

Condition 1 expresses that ϕ_i is injective except for entities imploded (mapped) onto ∞ . Condition 2 says that every entity in state q_i represents an entity of the state E_i of the allocation sequence. Condition 3 ensures that a bounded state does not generate states of the allocation sequence with entities corresponding to imploded ones. Condition 4 ensures that λ_i^σ is consistent to λ_i

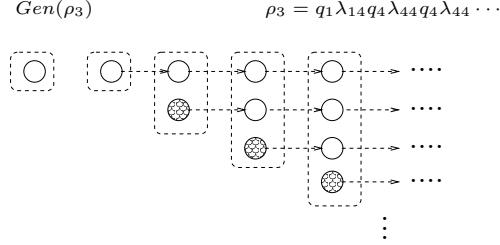


Figure 4.7: Folded allocation sequences generated by run ρ_3 of the HABA of Fig. 4.4.

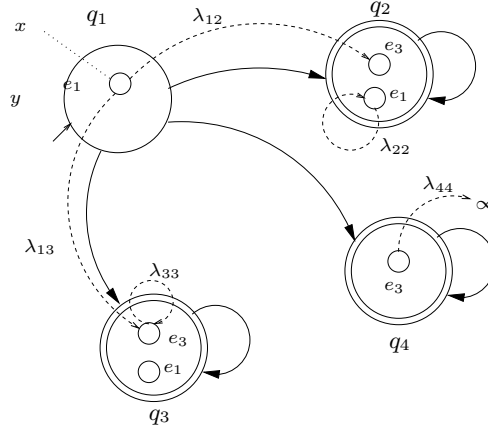


Figure 4.8: A modification of the HABA in Fig. 4.4.

in the reallocation of corresponding entities. Condition 5 states that only old entities may be imploded into ∞ . Thus, the number of new entities in the allocation sequence is the same as the number of new entities in the state of the model. The last condition relates the initial valuation of the allocation triple (σ, N, θ) to the initial valuation of HABA \mathcal{H} .

Runs of a HABA are defined in the same way as for ABA. Let $\text{runs}(\mathcal{H})$ denote the set of runs of \mathcal{H} . Using the definition of generator we can now define the language of \mathcal{H} as follows:

$$\mathcal{L}(\mathcal{H}) = \{(\sigma, N, \theta) \mid \exists \rho \in \text{runs}(\mathcal{H}) : \rho \text{ generates } (\sigma, N, \theta)\}. \quad (4.6)$$

The set of allocation sequences generated by $\rho \in \text{runs}(\mathcal{H})$ is denoted by

$$\text{Gen}(\rho) = \{\sigma \mid \rho \text{ generates } (\sigma, N, \theta)\}.$$

Example 4.3.7. Figure 4.6 shows some sequences generated by the HABA in Figure 4.4. In particular let

$$\begin{aligned}\rho_1 &= q_1 \lambda_{12} q_2 \lambda_{22} q_2 \lambda_{22} \cdots \\ \rho_2 &= q_1 \lambda_{13} q_3 \lambda_{33} q_3 \lambda_{33} \cdots\end{aligned}$$

then $\sigma_1, \sigma_2 \in \text{Gen}(\rho_1)$ and $\sigma_3, \sigma_4 \in \text{Gen}(\rho_2)$. Again, note how the identity of the entities is not relevant and in fact, in general, the set $\text{Gen}(\rho_1)$ contains all the sequences isomorphic to σ_1 (and/or σ_2) and $\text{Gen}(\rho_2)$ contains all the sequences isomorphic to σ_3 (and/or σ_4). We can abstract from the identity of the entities and represent sequences by anonymous entities depicted just by a circle⁸. This shows the “general pattern” followed by all isomorphic sequences. For example in Figure 4.6, sequences in $\text{Gen}(\rho_1)$ follow the general pattern represented by $\tilde{\sigma}$ while those in $\text{Gen}(\rho_2)$ follow the pattern of $\hat{\sigma}$.

Figure 4.7 depicts the pattern of the sequences in $\text{Gen}(\rho_3)$ where

$$\rho_3 = q_1 \lambda_{14} q_4 \lambda_{44} q_4 \lambda_{44} \cdots$$

The number of entities in $\text{Gen}(\rho_3)$ grows at every step.

Finally, consider the HABA in Figure 4.8 which is similar to the one in Figure 4.4 except that the initial state q_1 is now unbounded. In this case, $\text{Gen}(\rho_1)$, $\text{Gen}(\rho_2)$ and $\text{Gen}(\rho_3)$ contain also those sequences with an arbitrary number of (additional) imploded entities. This situation is represented in Figure 4.9 where, in order to stress the difference w.r.t. the bounded case, we have repeated $\tilde{\sigma}$ from Figure 4.6. The two sequences σ_1 and σ_2 in the bottom (of Fig. 4.9) are obtained by adding to $E_0^{\tilde{\sigma}}$ two and three imploded entities (represented in the shadow subpart of the state) respectively. In this example, imploded entities are preserved by every transition because q_2 is unbounded (recall condition $\lambda(\infty) = \infty$ in the definition of ∞ -reallocation and condition on HABA transitions). \square

Reallocations and black-hole abstraction. HABA exploit two abstraction techniques in order to cope with several cases of infinite state-space explosion: reallocations (and therefore local names) and black hole abstraction. Reallocations are a typical characteristic of HD-automata and have been proved — in the work of Montanari and Pistore [83, 84, 90] — to be a powerful abstraction mechanism. They permit to collapse, into a single state, a set of states that differ from one another only up to renaming of names. This technique results to be very effective when applied to the semantics of history-dependent calculi such as the π -calculus [82]. In that context for example, a state performing a single input action would generate an infinite bunch of transitions connecting the source state to an infinite number of target states whose only difference is the fresh (new) name representing the value received as input⁹. However,

⁸In the metempsychosis metaphor this would correspond to looking only at the live being while abstracting from which soul it has at any moment.

⁹We refer the reader to the aforementioned literature on HD-automata for further details on the benefit of local names for history dependent calculi.

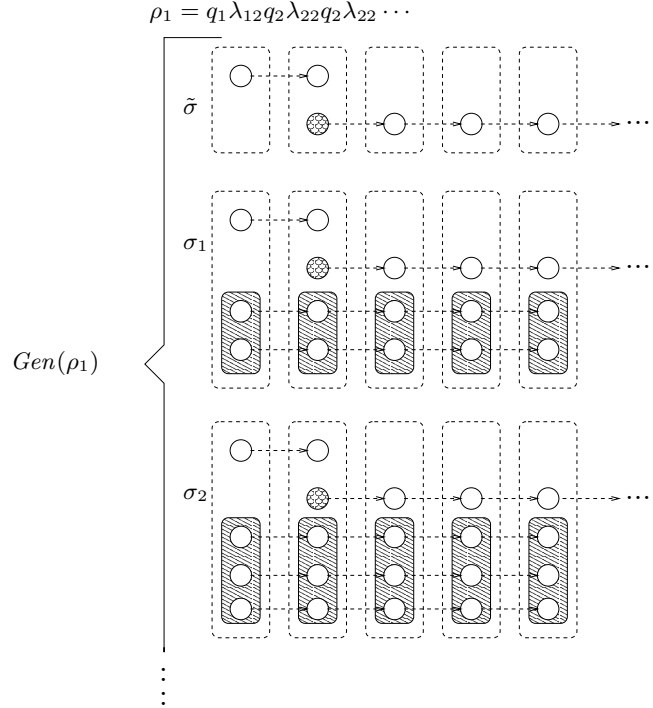


Figure 4.9: Folded allocation sequences generated for by HABA of Fig. 4.8 where $\lfloor q_1 \rfloor$.

what is probably more interesting for us is that the benefits derived by the reallocation mechanism in the context of those calculi is also inherited in our setting for the characterisation of the birth and death behaviour of entities. As a very simple example, consider the sequence of states

$$q'_3 q''_3 q'''_3 \dots \tag{4.7}$$

of the automaton in Figure 4.10. They only differ from one another by the entity that is born and dies at every state. The birth and death behaviour in this example can be translated into words as follows:

“there exists one entity that is always alive, and at every state a new entity is born which is then deallocated in the next state”.

For the purpose of describing the previous behaviour it is not necessary to distinguish among different entity identities. For this particular example, without reallocation the best we can do is to reduce the sequence in two distinct states and two transitions, e.g.:

$$q'_3 \rightarrow q''_3 \quad \text{and} \quad q''_3 \rightarrow q'_3$$

Using reallocations we can even avoid to distinguish between q'_3 and q''_3 and describe this behaviour with only one state and one transition, namely q_3 and $q_3 \rightarrow_{\lambda_{33}} q_3$ of the automaton in Figure 4.4.

Although reallocations, as we have just seen, permit state-space reductions in case of states equal up to renaming (i.e., isomorphic), this is still not enough to achieve a finite-state representation in many situations. The black hole abstraction helps in several of these cases, as for example, for some behaviours involving the allocation of an unbounded number of entities. Consider, the sequence of states:

$$q'_4 q''_4 q'''_4 q^{iv}_4 \dots \quad (4.8)$$

of the automaton in Figure 4.10. These states are not isomorphic, therefore only the application of the reallocation mechanism is not very effective here. Note however, that the sequence (4.8) has a rather constant behaviour, namely only one entity at a time is allocated and none of the alive entities die. In situations like this, the black hole in combination with the reallocation mechanism permits to collapse (4.8) in the single state and transition: $q_4 \rightarrow_{\lambda_{44}} q_4$ of the automaton in Figure 4.4.

4.3.3 The duality between ABA and HABA

The relationship between ABAs and HABAs strongly depends on the relation between unfolded and folded allocation sequences discussed in Section 4.2.4. More precisely, we will establish a connection between a HABA \mathcal{H} and a class of particular ABAs, called the expansions of \mathcal{H} , whose elements accept $\mathcal{L}(\mathcal{H})$ up to isomorphism.

Definition 4.3.8. For HABA \mathcal{H} and ABA \mathcal{A} , let $\psi : Q_{\mathcal{A}} \rightarrow Q_{\mathcal{H}}$ be surjective, and $(\phi_q)_{q \in Q_{\mathcal{A}}}$ be a family of functions $\phi_q : E_q \rightarrow E_{\psi(q)}^{\infty}$. Then:

- \mathcal{A} -transition $q_1 \rightarrow q_2$ *expands* \mathcal{H} -transition $q'_1 \rightarrow_{\lambda} q'_2$ if $q'_1 = \psi(q_1)$, $q'_2 = \psi(q_2)$ and
 - (i) $\lambda \circ \phi_{q_1} = \phi_{q_2} \upharpoonright (E_{q_1} \cap E_{q_2})$ and
 - (ii) $|E_{q'_2} \setminus \text{cod}(\lambda)| = |E_{q_2} \setminus E_{q_1}|$.
- \mathcal{A} is an *expansion* of \mathcal{H} if the following conditions are satisfied:
 1. $\forall e, e' \in E_q. (\phi_q(e) = \phi_q(e') \neq \infty \Rightarrow e = e')$;
 2. $E_{\psi(q)} \subseteq \text{cod}(\phi_q)$;
 3. $\lceil \psi(q) \rceil \Rightarrow \infty \notin \text{cod}(\phi_q)$;
 4. for all $q_1 \in Q_{\mathcal{A}}$,
 - a) for all $\psi(q_1) \rightarrow_{\lambda} q'_2$ there exists $q_1 \rightarrow q_2$ that expands $\psi(q_1) \rightarrow_{\lambda} q'_2$
 - b) for all $q_1 \rightarrow q_2$ there exists λ such that $q_1 \rightarrow q_2$ expands $\psi(q_1) \rightarrow_{\lambda} \psi(q_2)$;

5. $I_{\mathcal{A}} : q \mapsto \begin{cases} (\phi_q^{-1}(N), \phi_q^{-1} \circ \theta) & \text{if } q' = \psi(q) \text{ and } I_{\mathcal{H}}(q') = (N, \theta) \\ \text{undefined} & \text{otherwise} \end{cases}$
6. $\mathcal{F}_{\mathcal{A}} = \{\{\psi(q) | q \in F\} | F \in \mathcal{F}_{\mathcal{H}}\}$.

The first three conditions are taken from Def. 4.3.6. Intuitively, these force the number of entities in an expanded state to exceed the number of entities of the original state, and require equality if the original state is bounded. Note that transition $q_1 \rightarrow q_2$ in the ABA must preserve the reallocation of $q'_1 \rightarrow_{\lambda} q'_2$ (condition (i)) as well as the number of new entities (condition (ii)).

A HABA \mathcal{H} whose initial states are all bounded has a single expansion (up to isomorphism). On the contrary, if there exists at least an unbounded initial state, then \mathcal{H} has an infinite number of expansions, corresponding to the possible number of imploded entities contained in the black hole. This effect is similar to that of the set of sequences generated by a single run as we have seen in Example 4.3.7 and in Figure 4.9.

Some notions for folded and unfolded allocation sequences are lifted to accepted languages in the following way. For HABAs \mathcal{H}_1 and \mathcal{H}_2 , let $\mathcal{L}(\mathcal{H}_1) \cong \mathcal{L}(\mathcal{H}_2)$ iff for all $(\sigma_1, N_1, \theta_1) \in \mathcal{L}(\mathcal{H}_1)$ there exists a $(\sigma_2, N_2, \theta_2) \in \mathcal{L}(\mathcal{H}_2)$ such that $(\sigma_1, N_1, \theta_1) \cong (\sigma_2, N_2, \theta_2)$ and vice versa. For ABA \mathcal{A} accepting $\mathcal{L}(\mathcal{A})$, let

$$id(\mathcal{L}(\mathcal{A})) = \{(id(\sigma), N, \theta) \mid (\sigma, N, \theta) \in \mathcal{L}(\mathcal{A})\}. \quad (4.9)$$

Note that $id(\mathcal{L}(\mathcal{A}))$ is the folded version of the unfolded language $\mathcal{L}(\mathcal{A})$. For HABA \mathcal{H} and ABA \mathcal{A} let $\mathcal{L}(\mathcal{A}) \sqsubseteq^{fold} \mathcal{L}(\mathcal{H})$ iff for all $(\sigma_u, N_u, \theta_u) \in \mathcal{L}(\mathcal{A})$ there exists $(\sigma_f, N_f, \theta_f) \in \mathcal{L}(\mathcal{H})$ such that $(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f)$. Similarly, $\mathcal{L}(\mathcal{A}) \supseteq^{fold} \mathcal{L}(\mathcal{H})$ iff for all $(\sigma_f, N_f, \theta_f) \in \mathcal{L}(\mathcal{H})$ there exists $(\sigma_u, N_u, \theta_u) \in \mathcal{L}(\mathcal{A})$ such that $(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f)$. Then:

Lemma 4.3.9. For HABA \mathcal{H} and any expansion $Exp(\mathcal{H})$:

- (a) $\mathcal{L}(Exp(\mathcal{H})) \supseteq^{fold} \mathcal{L}(\mathcal{H})$ and
- (b) $\mathcal{L}(Exp(\mathcal{H})) \sqsubseteq^{fold} \mathcal{L}(\mathcal{H})$.

Proof. See Appendix A.2. □

Hence, $\mathcal{L}(Exp(\mathcal{H}))$ corresponds precisely to the unfolded version of $\mathcal{L}(\mathcal{H})$. A consequence of the previous lemma, as well as an alternative way to express it, is stated by:

Theorem 4.3.10. For HABA \mathcal{H} and any expansion $Exp(\mathcal{H})$:

$$\mathcal{L}(\mathcal{H}) \cong id(\mathcal{L}(Exp(\mathcal{H}))).$$

Proof. See Appendix A.2. □

The previous theorem will be useful for proving the equivalence of the concrete and symbolic operational semantics of the programming language we define in Section 4.4.

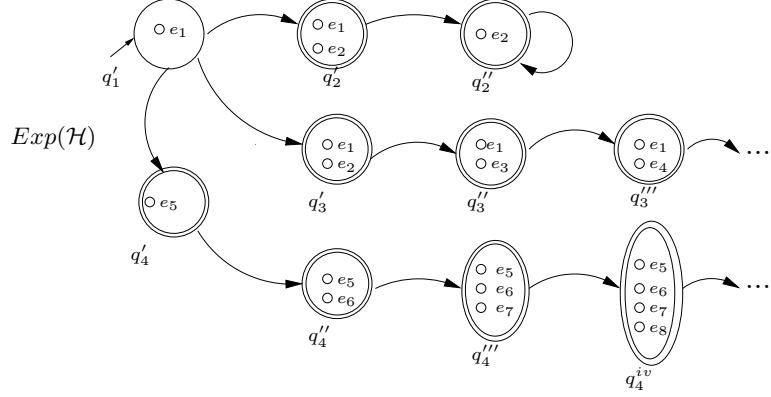


Figure 4.10: The infinite-state expansion of the HABA in Fig. 4.4.

Definition 4.3.11. Given a HABA \mathcal{H} and an $\mathcal{A}llTL$ -formula ϕ we say that

- ϕ is \mathcal{H} -satisfiable if there exists $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$ such that $\sigma, N, \theta \models \phi$;
- ϕ is \mathcal{H} -valid if for all $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}) : \sigma, N, \theta \models \phi$.

For ABA \mathcal{A} a similar definition for \mathcal{A} -satisfiability and \mathcal{A} -validity can be given. From the previous definition it follows that ϕ is \mathcal{H} -valid (\mathcal{A} -valid) if and only if $\neg\phi$ is not \mathcal{H} -satisfiable (\mathcal{A} -satisfiable).

As stated in the following corollary, from Theorem 4.3.10 it follows that a HABA \mathcal{H} and any of its expansions satisfy the same set of $\mathcal{A}llTL$ formulae.

Corollary 4.3.12. For any $\mathcal{A}llTL$ -formula ϕ , HABA \mathcal{H} and expansion $Exp(\mathcal{H})$:

$$\phi \text{ is } \mathcal{H}\text{-satisfiable} \Leftrightarrow \phi \text{ is } Exp(\mathcal{H})\text{-satisfiable.}$$

Example 4.3.13. Figure 4.10 shows the (infinite) expansion of the HABA in Figure 4.4. The HABA in Figure 4.8 has an infinite number of expansions due to its unbounded initial state. The expansion shown in Figure 4.11 corresponds to the case where there are two entities in the black hole. These entities are depicted in a shadow area to be easily distinguished from the others. \square

4.4 Programming allocation and deallocation

This section introduces a simple programming language \mathcal{L} capturing the essence of allocation and deallocation. It is used for providing an intuition about the setup and the sort of behaviour that can be modelled by HABAs. Two semantics for \mathcal{L} are defined, a concrete semantics with ABA as underlying model, and a symbolic semantics using HABA. The relation between these semantics is shown to correspond to expansion in the sense of Def. 4.3.8.

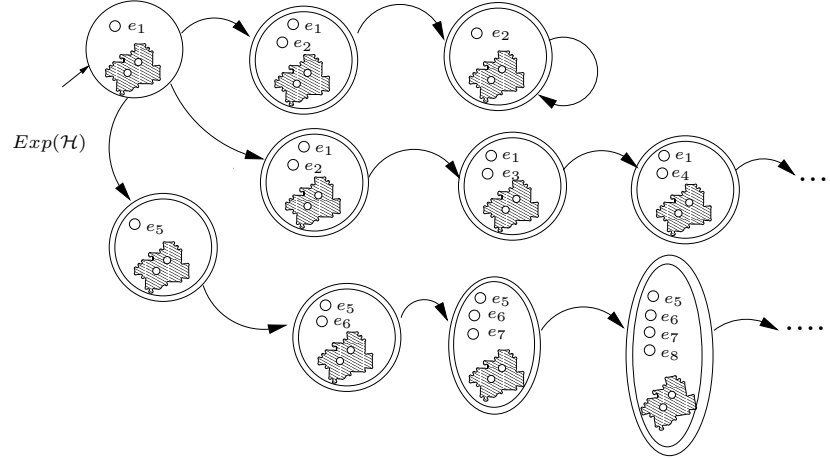


Figure 4.11: An infinite-state expansion of the HABA in Fig. 4.8.

4.4.1 Syntax

For PVAR a set of program variables with $v, v_i \in \text{PVAR}$ and $\text{PVAR} \cap \text{LVAR} = \emptyset$, the set of statements of \mathcal{L} is given by:

$$\begin{aligned}
 (p \in \mathcal{L}) & ::= \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k) \\
 (s \in \text{Stat}) & ::= \text{new}(v) \mid \text{del}(v) \mid v := v \mid \text{skip} \mid s; s \mid \text{if } b \text{ then } s \text{ else } s \text{ fi} \\
 & \quad \mid \text{while } b \text{ do } s \text{ od} \\
 (b \in \text{Bexp}) & ::= v = v \mid b \vee b \mid \neg b
 \end{aligned}$$

A program p is thus a parallel composition of a finite number of statements preceded by the declaration of a finite number of global variables.

Informal semantics of \mathcal{L} . $\text{new}(v)$ creates (i.e., allocates) a new entity that will be referred to by the program variable v . The old value of v is lost. Thus, if v is the only variable that refers to entity e , say, then after the execution of $\text{new}(v)$, e cannot be referenced anymore. In particular, e cannot be deallocated anymore. In other words, there is no automatic garbage collection¹⁰. $\text{del}(v)$ destroys (i.e., deallocates) the entity associated to v , and makes v undefined. The assignment $v := w$ passes the reference held by w (if any) to v . Again, the entity v was referring to might become unreferenced (for ever). Sequential composition, while loop, skip, and conditional statement have the standard interpretation. For the sake of simplicity, new and del create and destroy,

¹⁰In Chapter 5 we will study a model with garbage collection.

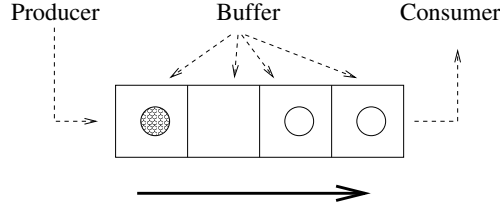


Figure 4.12: The buffer system.

respectively, a single entity only; generalisations such as

$$\begin{aligned} &\text{new}(v_1, \dots, v_n) \\ &\text{del}(v_1, \dots, v_n) \end{aligned}$$

in which several entities are considered simultaneously can be added in a straightforward manner.

Example 4.4.1. The following program PC is an implementation of a producer/consumer system depicted in Figure 4.12. A producer process creates new entities and stores them in the first place on the left side of a shared buffer. A buffer process moves these entities from the left to the right. The consumer process consumes entities in the first position on the right of the buffer¹¹.

$$\begin{aligned} PC &\equiv \text{decl } v_1, v_2, w : (Prod \parallel Buff_2 \parallel Cons) \text{ where} \\ Prod &\equiv \text{while tt do} \\ &\quad \text{if } (v_1 \text{ dead}) \text{ then } \text{new}(v_1) \text{ else skip fi} \\ &\quad \text{od} \\ Buff_2 &\equiv \text{while tt do} \\ &\quad \text{if } (v_1 \text{ alive}) \text{ then } v_2 := v_1; v_1 := w \text{ else skip fi} \\ &\quad \text{od} \\ Cons &\equiv \text{while tt do} \\ &\quad \text{if } (v_2 \text{ alive}) \text{ then } \text{del}(v_2) \text{ else skip fi} \\ &\quad \text{od} \end{aligned}$$

For this particular instance of the system, the component $Buff_2$ implements a process handling a two-place buffer consisting of the variable v_1 (first position) and v_2 (second position). The producer $Prod$ produces a new entity when the first place of the buffer v_1 is empty (i.e. v_1 is dead). An entity in the second position of the buffer is consumed by the consumer $Cons$. Note that in the if-then-else-fi statement in $Buff_2$, using $v_2 := v_1; \text{del}(v_1)$ would be wrong as the entity referenced by v_1 and v_2 would then be deallocated. The statement, $v_1 := w$ deletes the reference to v_1 entity, since w is assumed to be undefined (and

¹¹For the boolean variables, we write v alive for $v = v$ and v dead for $v \neq v$ (cf. Proposition 4.2.4).

therefore it will remain constantly undefined). It is not difficult to see that the proposed implementation suffers from *memory leak* as $Buff_2$ may overwrite v_2 before it is actually consumed. Absence of memory leak can be easily expressed by the formula:

$$G(\forall x.Fx \text{ dead}) \quad (4.10)$$

essentially saying: *a produced entity is always eventually consumed*. Using the operational semantics defined in Section 4.4.3 and the model checking algorithm of Section 4.5 it is possible to automatically verify that (4.10) is refuted by some computations of PC (or alternatively its negation can be satisfied).

An improved implementation uses w as temporary variable in order to pass v_1 to v_2 :

$$\begin{aligned} PC' &\equiv \text{decl } v_1, v_2, w : (Prod \parallel Buff_2' \parallel Cons) \text{ where} \\ Buff_2' &\equiv \text{while tt do} \\ &\quad \text{if } (v_1 \text{ alive}) \text{ then } w := v_1; v_1 := v_2; v_2 := w \text{ else skip fi} \\ &\quad \text{od} \end{aligned}$$

and $Prod$, $Cons$ defined as above. The implementation PC' improves PC since the memory leak problem has been removed. In fact, the assignment $v_1 := v_2$ is used to make v_1 undefined. In case v_2 points to an entity, its reference is not lost and goes to v_1 that since remains alive block $Prod$. On the other hand, the entity v_1 holds at the beginning of the if statement is not lost since its reference is given temporarily to w . Therefore (4.10) is valid in PC' . Nevertheless, by looking more carefully, we can observe that PC' does not always remove entities from the buffer in the same order in which they are introduced — that as a matter of fact — is a desirable property if we want to implement a FIFO queue. Exploiting the fact that in \mathcal{L} only one entity at a time can be created, and the computation starts with an empty set of live entities, we can express for PC' the FIFO policy by the \mathcal{ALLTL} -formula:

$$G(\forall x.XF(\exists y.y \text{ new} \Rightarrow y \text{ alive} \cup x \text{ dead})) \quad (4.11)$$

that in words translates to: *entities are always consumed in the same order they are produced*. To repair the FIFO order we propose yet another implementation for the buffer system:

$$\begin{aligned} PC'' &\equiv \text{decl } v_1, v_2, w : (Prod \parallel Buff_2'' \parallel Cons) \text{ where} \\ Buff_2'' &\equiv \text{while tt do} \\ &\quad \text{if } (v_1 \text{ alive} \wedge v_2 \text{ dead}) \text{ then} \\ &\quad \quad w := v_1; v_1 := v_2; v_2 := w \\ &\quad \text{else skip fi} \\ &\quad \text{od} \end{aligned}$$

Again, $v_1 := v_2$ deletes only the reference to v_1 entity that can be then moved to v_2 via w . The correctness of the solution comes from the guard of the conditional statement. In fact, only if the second position of the buffer is empty the entity pointed to by v_1 is shifted to v_2 .

Some other example properties expressible in \mathcal{AllTL} for the program PC'' are¹²:

- An entity in the buffer is not consumed before the insertion of another one (this implies also that the buffer is never empty): $\mathbf{G}\forall x.(x \text{ alive} \cup (\exists y.x \neq y))$.
- The buffer will be always eventually full: $\mathbf{GF}(\forall x.\forall y.\forall z.(x = y \vee x = z \vee y = z))$.

□

Example 4.4.2. The following program, where $g(i) = (i+1) \bmod 4$, models the implementation of a naive solution to the dining philosopher problem:

```

DPhil  ≡  decl  $v_1, v_2, v_3, v_4 : \text{Init}; (Ph_1 \parallel Ph_2 \parallel Ph_3 \parallel Ph_4)$  where
Phi   ≡  while tt do
           while  $v_i$  dead do skip od
           del( $v_i$ );
           while  $v_{g(i)}$  dead do skip od
           del( $v_{g(i)}$ );
           new( $v_i$ );
           new( $v_{g(i)}$ )
         od
Init   ≡  new( $v_1$ ); new( $v_2$ ); new( $v_3$ ); new( $v_4$ )

```

The variables v_i and $v_{g(i)}$ represent the left and the right chopstick of philosopher Ph_i , respectively. If v_i and $v_{g(i)}$ are defined, then the chopsticks are on the table. Taking the chopsticks from the table is represented by destroying the corresponding entities, while putting the chopsticks back on the table is modelled by creating new entities. Some desirable \mathcal{AllTL} properties of this program, are:

- No memory leak (i.e. the number of entities never exceeds four):

$$\mathbf{G}(\forall x_1.\forall x_2.\forall x_3.\forall x_4.\forall x_5.(x_1 = x_2 \vee x_1 = x_3 \vee x_1 = x_4 \vee x_1 = x_5 \vee x_2 = x_3 \vee x_2 = x_4 \vee x_2 = x_5 \vee x_3 = x_4 \vee x_3 = x_5 \vee x_4 = x_5)).$$

- Eventually the system deadlocks:

$$\mathbf{FG}(\forall x.x \text{ dead}).$$

- Eventually two philosophers will eat at the same time:

$$\mathbf{F}(\forall x.x \text{ dead} \wedge \mathbf{FX}\exists x.x \text{ new}).$$

¹²The specifications are correct because of the absence of memory leak in PC'' .

The last two properties make the assumption that it has been already checked the absence of memory leak. \square

Although some interesting problems can be programmed, it is obvious that \mathcal{L} is rather simple. Other constructs like `wait` or some syntactic sugar like `repeat` may be easily included, without extending the language in an essential way. In Chapter 5 we will define an extension of \mathcal{L} that deals with a simplified mechanism of navigation and therefore permits to express more involved examples.

4.4.2 Concrete semantics

A concrete semantics of our example language is given in terms of ABA. Let Par denote the compound statements, i.e., $r(\in Par) ::= s \mid r \parallel s$.

Definition 4.4.3 (concrete automaton \mathcal{A}_p). The concrete semantics of $p = \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$ is the ABA $\mathcal{A}_p = \langle \emptyset, Q, E, \rightarrow, I, \mathcal{F} \rangle$ where

- $Q \subseteq Par \times 2^{Ent} \times (PVAR \rightarrow Ent)$, where for state $q = (r, E_q, \gamma_q) \in Q$, r is the compound statement to be executed, E_q is the set of entities alive and γ_q maps program variables to Ent (with $\text{dom}(\gamma_q) = \{v_1, \dots, v_n\}$; if γ_q is undefined on v we write $\gamma_q(v) = \perp$);
- $E(r, E', \gamma) = E'$;
- $\rightarrow \subseteq Q \times Q$ is the smallest relation satisfying the rules in Table 4.4.2;
- $\text{dom}(I) = \{(s_1 \parallel \dots \parallel s_k, \emptyset, \emptyset)\}$ and $I(s_1 \parallel \dots \parallel s_k, \emptyset, \emptyset) = (\emptyset, \emptyset)$;
- let

$$\begin{aligned} \widehat{F}_i &= \{(s'_1 \parallel \dots \parallel s'_k, E, \gamma) \in Q \mid s'_i = \text{skip} \vee s'_i = \text{while } b \text{ do } s \text{ od}; s''\} \\ \widetilde{F}_i &= \{(s'_1 \parallel \dots \parallel s'_k, E, \gamma) \in Q \mid s'_i = \text{skip} \vee s'_i = s; \text{while } b \text{ do } s \text{ od}; s''\} \end{aligned}$$

$$\text{then } \mathcal{F} = \{\widehat{F}_i \mid 0 < i \leq k\} \cup \{\widetilde{F}_i \mid 0 < i \leq k\}.$$

A few remarks are in order. \mathcal{A}_p has a single initial state $s_1 \parallel \dots \parallel s_k$. The set of accept states for the i -th sequential component consists of all states in which the component has either terminated ($s_i = \text{skip}$) or is processing a loop (which could be infinite). The combination of sets \widehat{F}_i and \widetilde{F}_i guarantees the progress of the component i .

Definition 4.4.4. The semantics of the boolean expressions is given by the function $\mathcal{V} : Bexp \times (PVAR \rightarrow Ent) \rightarrow \mathbb{B}$ defined by

$$\begin{aligned} \mathcal{V}(v = w)(\gamma) &= \begin{cases} \text{tt} & \text{if } v, w \in \text{dom}(\gamma) \text{ and } \gamma(v) = \gamma(w) \\ \text{ff} & \text{otherwise} \end{cases} \\ \mathcal{V}(b_1 \vee b_2)(\gamma) &= \mathcal{V}(b_1)(\gamma) \vee \mathcal{V}(b_2)(\gamma) \\ \mathcal{V}(\neg b)(\gamma) &= \neg \mathcal{V}(b)(\gamma). \end{aligned}$$

(ASGN-c)	$\frac{}{v := w, E, \gamma \rightarrow \text{skip}, E, \gamma\{\gamma(w)/v\}}$
(NEW-c)	$\frac{}{\text{new}(v), E, \gamma \rightarrow \text{skip}, E \cup \{e\}, \gamma\{e/v\}} \quad e = \min(Ent \setminus E)$
(DEL-c)	$\frac{}{\text{del}(v), E, \gamma \rightarrow \text{skip}, E \setminus \{\gamma(v)\}, \gamma\{\perp/v'\}} \quad v' \in \gamma^{-1}(\gamma(v))$
(IF ₁ -c)	$\frac{\mathcal{V}(b)(\gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, E, \gamma \rightarrow s_1, E, \gamma}$
(IF ₂ -c)	$\frac{\neg\mathcal{V}(b)(\gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, E, \gamma \rightarrow s_2, E, \gamma}$
(WHILE-c)	$\frac{}{\text{while } b \text{ do } s \text{ od}, E, \gamma \rightarrow \text{if } b \text{ then } s; \text{while } b \text{ do } s \text{ od else skip fi}, E, \gamma}$
(SEQ ₁ -c)	$\frac{s_1, E, \gamma \rightarrow s'_1, E', \gamma'}{s_1; s_2, E, \gamma \rightarrow s'_1; s_2, E', \gamma'}$
(SEQ ₂ -c)	$\frac{}{\text{skip}; s_2, E, \gamma \rightarrow s_2, E, \gamma}$
(PAR ₁ -c)	$\frac{1 \leq j \leq k \wedge s_j, E, \gamma \rightarrow s'_j, E', \gamma'}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, E, \gamma \rightarrow s_1 \parallel \dots \parallel s'_j \parallel \dots \parallel s_k, E', \gamma'}$
(PAR ₂ -c)	$\frac{}{\text{skip} \parallel \dots \parallel \text{skip}, E, \gamma \rightarrow \text{skip} \parallel \dots \parallel \text{skip}, E, \gamma}$

Table 4.3: Operational rules for concrete semantics.

We assume w.l.o.g. that the set Ent is totally ordered; this is convenient for selecting a fresh entity in a deterministic way (cf. the rule **NEW-conc** in Table 4.4.2). Some brief explanation of the rules of the concrete semantics follows:

- (**ASGN-c**) An assignment $v := w$ is performed by changing the reference of the variable v to $\gamma(w)$. After the execution, the assignment statement is replaced by **skip** that is either consumed in the context of a sequential composition by rule (**SEQ₂-c**) or is blocked. This general pattern is followed also by rules (**NEW-c**) and (**DEL-c**). Note that there does not exist a rule for **skip**.
- (**NEW-c**) The reference of the first (fresh) entity e available from Ent according to the total order is assigned to v .
- (**DEL-c**) Entity $\gamma(v)$ is deallocated and every reference to this entity is

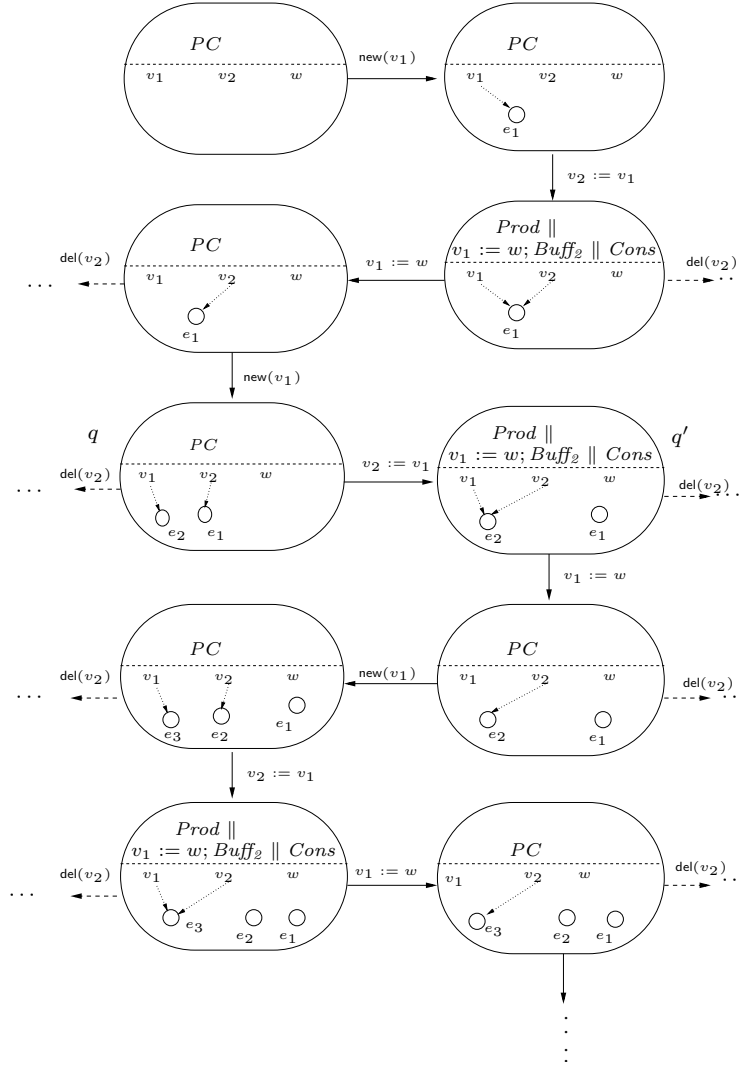


Figure 4.13: Initial sequence of states A_{PC} of Example 4.4.1 that makes the automaton infinite.

cancelled.

- (IF₁-c)/(IF₂-c)/(WHILE-c) Straightforward.
- (SEQ₁-c)/(SEQ₂-c) In a sequential composition, when the first statement is reduced to a skip statement, it is consumed.
- (PAR₁-c) If one of the components of the compound statement performs

a step, the compound statement can do so.

- (PAR₂-c) A self-loop in an accept state with a terminated compound statement ensures that, in a run, it is visited infinitely many times.

Example 4.4.5. The ABA in Figure 4.13 shows a sub-part of the automaton \mathcal{A}_{PC} representing the concrete semantics of the erroneous implementation PC described in Example 4.4.1. The figure focuses on the part of the automaton corresponding to a computation that makes \mathcal{A}_{PC} infinite because of memory leakage. In fact, following transitions depicted with a solid arrow, the number of non-reachable entities grows unboundedly. \square

4.4.3 Symbolic semantics

The semantics defined in the previous section has the disadvantage that models may become infinite due to an unbounded number of entity creations (cf. Example 4.4.5). In order to circumvent this problem we define a symbolic semantics of \mathcal{L} in terms of HABA, that (later on) will be shown to be equivalent. The main distinction with the concrete semantics is the treatment of the association of program variables to entities. Here instead, entities are represented by a partial partitioning of a subset of PVAR, i.e., the set E of entities is of the form $\{X_1, \dots, X_n\}$ with $X_i \subseteq \text{PVAR}$ and $X_i \cap X_j = \emptyset$ (for $i \neq j$). Note that we do not require $\bigcup_i X_i = \text{PVAR}$ which would make it a full partitioning. Variable v is defined if and only if $v \in X_i$ for some i . Then, v refers to the entity represented by the set X_i . Otherwise, v is undefined. Using this approach, there is no need to represent (in a state) a mapping from the set of program variables onto the entities.

Example 4.4.6. Consider Figure 4.13. In the symbolic semantics, state q will be represented by:

$$\{\{v_1\}, \{v_2\}\}$$

because v_1 and v_2 are aliases. Whereas state q' will be represented as

$$\{\{v_1, v_2\}\}$$

because v_1 and v_2 are no longer aliases. \square

This kind of representation is possible only because of the use of reallocation that make entity identities not relevant. In the concrete semantics this would be impossible due to the global notion of entity identity.

Definition 4.4.7 (symbolic automaton \mathcal{H}_p). The symbolic semantics of $p = \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$ is the HABA $\mathcal{H}_p = \langle \emptyset, Q, E, \rightarrow, I, \mathcal{F} \rangle$ where

- $Q \subseteq \text{Par} \times 2^{2^{\text{PVAR}}}$, i.e., a state $q = (r, E)$ consists of a compound statement and a set of entities; we have $[q]$ iff $\emptyset \in E$ (i.e., we represent the black hole by \emptyset).

(ASGN-s)	$\frac{}{v := w, E \rightarrow_\lambda \text{skip}, \{X_i \setminus \{v\} w \notin X_i\} \cup \{X_i \cup \{v\} w \in X_i\}}$ $\lambda : X_i \mapsto \begin{cases} X_i \setminus \{v\} & \text{if } w \notin X_i \\ X_i \cup \{v\} & \text{otherwise} \end{cases}$
(NEW-s)	$\frac{}{\text{new}(v), E \rightarrow_\lambda \text{skip}, \{X_i \setminus \{v\} X_i \in E\} \cup \{\{v\}\}} \quad \lambda(X_i) = X_i \setminus \{v\}$
(DEL-s)	$\frac{v \in X_i}{\text{del}(v), E \rightarrow_\lambda \text{skip}, (E \setminus \{X_i\})} \quad \lambda : X_j \mapsto \begin{cases} X_j & \text{if } j \neq i \\ \perp & \text{otherwise} \end{cases}$
(IF ₁ -s)	$\frac{\mathcal{V}(b)(E)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, E \rightarrow_{id} s_1, \bar{E}}$
(IF ₂ -s)	$\frac{\neg \mathcal{V}(b)(E)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, E \rightarrow_{id} s_2, \bar{E}}$
(WHILE-s)	$\frac{}{\text{while } b \text{ do } s \text{ od}, E \rightarrow_{id} \text{if } b \text{ then } s; \text{while } b \text{ do } s \text{ od else skip fi}, \bar{E}}$
(SEQ ₁ -s)	$\frac{s_1, E \rightarrow_\lambda s'_1, E'}{s_1; s_2, E \rightarrow_\lambda s'_1; s_2, E'}$
(SEQ ₂ -s)	$\frac{}{\text{skip}; s_2, E \rightarrow_{id} s_2, \bar{E}}$
(PAR ₁ -s)	$\frac{1 \leq j \leq k \wedge s_j, E \rightarrow_\lambda s'_j, E'}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, E \rightarrow_\lambda s_1 \parallel \dots \parallel s'_j \parallel \dots \parallel s_k, E'}$
(PAR ₂ -s)	$\frac{}{\text{skip} \parallel \dots \parallel \text{skip}, E \rightarrow_{id} \text{skip} \parallel \dots \parallel \text{skip}, \bar{E}}$

Table 4.4: Operational rules for symbolic semantics

- $E(r, E') = E' \setminus \{\emptyset\}$;
- \rightarrow is the smallest relation defined by the rules in Table 4.4.3 such that for $r, E \rightarrow_\lambda r', E'$ we have $\emptyset \in E \Rightarrow \emptyset \in \text{dom}(\lambda)$.

and I and \mathcal{F} are defined in the same way as for the concrete semantics.

The condition on \emptyset in the definition of \rightarrow can be seen as a kind of “preservation law” of the black hole. In fact, once a state implodes into an unbounded one, the black hole generated by this implosion will last forever. Note that in the definition of HABA (Def. 4.3.4) this is not always the case. Whenever entity X_i is not referenced by any program variable, the state will become unbounded. Entity X_i will then be mapped by λ onto \emptyset , which can be viewed as a “black hole” collecting every non-referenced entity. These entities share the

property that they cannot be deallocated anymore, thus they will have exactly the same future, namely they will be “floating” in the black hole *ad infinitum*¹³.

Some explanations on the rules of the symbolic semantics are in order:

- (NEW-s) If v is the only variable having a reference to an entity X_i (i.e., $X_i = \{v\}$), the state becomes unbounded (if this is not already the case) and the black hole \emptyset implodes X_i since it cannot be referred to anymore. In this case, $E' = E \cup \{\emptyset\}$, i.e., the sets E and E' have the same entities. However, X_i represents a new entity in the target state since $X_i \notin \text{cod}(\lambda)$.

If v is either undefined or there exists another variable denoting v 's entity, a new entity $\{v\}$ is created. λ maps every entity onto itself.

- (ASGN-s) If v is defined but is the only variable that has a reference to its entity, the assignment causes the loss of the reference of the entity denoted by v . Therefore, this entity is imploded onto \emptyset . If w is undefined also v becomes undefined. Regardless whether the state is bounded or not, after the transition the state becomes unbounded because $\emptyset \in E'$.

If there is another variable denoting v 's entity or v is undefined then v 's reference is changed.

- (DEL-s) Straightforward. The entity X_i associated with v is removed from the live entities.

(IF₁-s), (IF₂-s), (WHILE-s), (SEQ₁-s), (SEQ₂-s), (PAR₁-s) and (PAR₂-s) are similar to the corresponding rules of the concrete semantics and are not explained further.

Example 4.4.8. Consider the program PC of Example 4.4.1 that — as we have already observed in Example 4.4.5 — has an infinite-state concrete semantics due to memory leak. Figure 4.14 depicts the corresponding symbolic semantics \mathcal{H}_{PC} of PC . Note how, by using black-hole abstraction and reallocations, the automaton is *finite-state*. \square

¹³This property enjoyed by imploded entities is only a choice in the design of this particular example semantics. It is neither an inherent characteristic of \mathcal{L} nor of HABAs. Here the choice is supported by the fact that we want to reason — among other properties of the system — about memory leak. Moving to an operational semantics with automatic garbage collector is straightforward. For example, this could be done by changing the condition on the black-hole imposed in the transitions. It could be substituted by $\emptyset \in E \Rightarrow \emptyset \notin \text{dom}(\lambda)$ which would correspond to a very eager garbage collector that runs at every transition. Alternatively, the simple operational rule

$$\overline{[q] \rightarrow_{id} [q]}$$

could be added that corresponds to the possibility for the system in an unbounded state to choose nondeterministically between the execution of the statement or the execution of the garbage collector. This last choice would be more coherent with the real existing systems using garbage collection. In Chapter 5 we take the opposite direction and we define an operational semantics with automatic garbage collection.

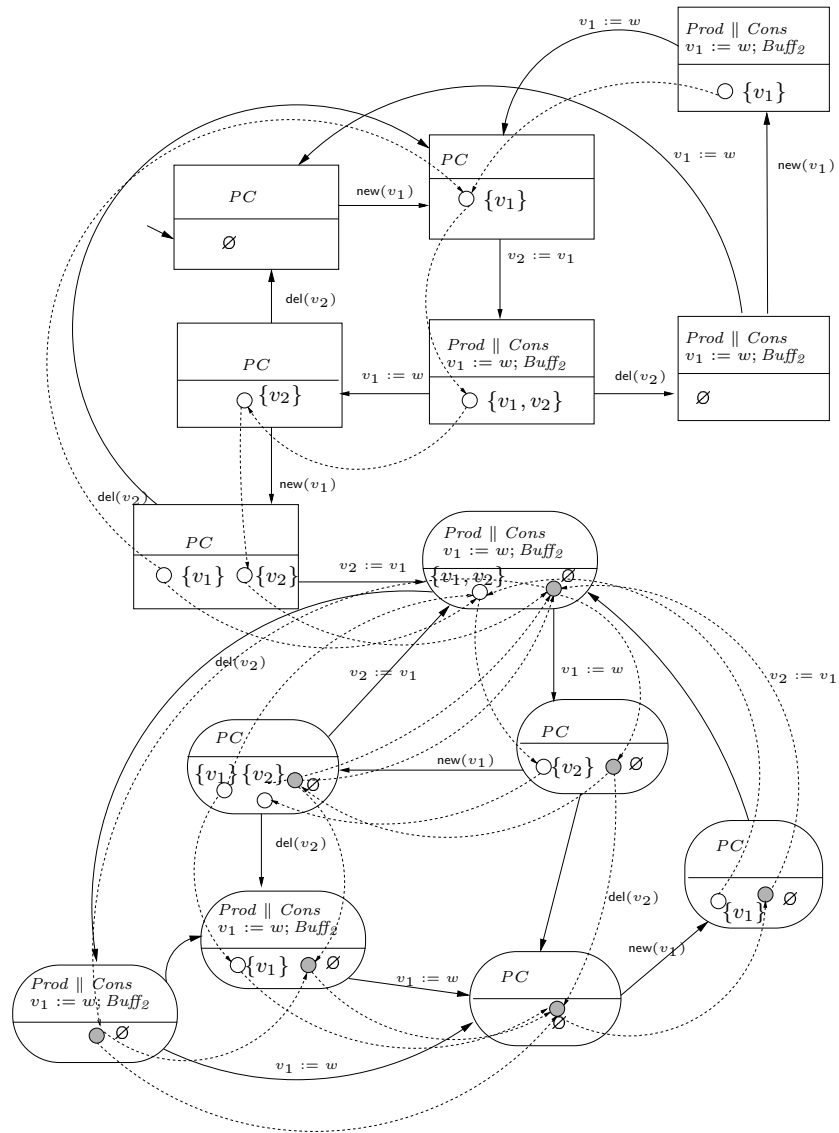


Figure 4.14: Symbolic semantics of PC of Example 4.4.1.

4.4.4 Relating the concrete and symbolic semantics

The theory developed in Section 4.3.3 can be applied here in order to compare the concrete and the symbolic semantic of a given program p w.r.t. the languages accepted by the two automata and, consequently, the set of \mathcal{AllTL} -formulae satisfied. In particular, the next theorem relates \mathcal{A}_p and \mathcal{H}_p .

Theorem 4.4.9. For any $p \in \mathcal{L}$: \mathcal{A}_p is an expansion of \mathcal{H}_p .

Proof. See Appendix A.3. □

It thus follows that \mathcal{H}_p and \mathcal{A}_p are equivalent: due to Theorem 4.3.10, they accept the same language up to isomorphism and therefore (by Corollary 4.3.12) they satisfy the same \mathcal{AllTL} -formulae. Theorem 4.4.9 increases its relevance — at least for the application of model checking techniques — when combined with the essential property of the symbolic semantics ensured by the next result:

Theorem 4.4.10. For any $p \in \mathcal{L}$: \mathcal{H}_p is finite state.

Proof. See Appendix A.3. □

More precisely, for B_k the number of partitions of a set of k elements, $|s_{max}|$ the size of the longest sequential statement in p and m the number of sequential components in p , we have:

$$|Q_{\mathcal{H}_p}| \leq |s_{max}|^m \cdot \left[1 + 2 \cdot \sum_{k=1}^{|\text{PVAR}|} \binom{|\text{PVAR}|}{k} B_k \right].$$

B_k is known as the *Bell number* and has a non-trivial asymptotic behaviour [87], nevertheless, $O(2^n) < O(B_n) < O(n!)$. Thus, $Q_{\mathcal{H}_p}$ is exponential in the number of sequential components m in p . Whereas if m is constant then $|Q_{\mathcal{H}_p}|$ can be approximated by $O(2^{|\text{PVAR}|} \cdot B_{|\text{PVAR}|})$. As the symbolic semantics is deterministic, the number of transitions is of the order $O(m \cdot |Q_{\mathcal{H}_p}|)$.

Although the result given in this section holds for a simple language, we believe that it may be extended to more interesting programming languages. For instance, \mathcal{L} may be enhanced, in order to model the precise mechanism of creation and destruction of objects in object-oriented programming languages. In Chapter 5 we will extend \mathcal{L} in order to deal with dynamic references.

4.5 Model checking \mathcal{AllTL}

In this section, we define an algorithm for model-checking \mathcal{AllTL} -formulae against a HABA. The algorithm extends the tableau method for LTL — introduced for the first time in [77] and summarised in Section 2.1.6 — to \mathcal{AllTL} .

We will evaluate \mathcal{AllTL} -formulae on states of a HABA by mapping the free variables of the formula to entities of the state. It should be clear that, in principle, any such mapping resolves all basic propositions: a freshness proposition

x new holds if and only if x is mapped to an entity that is new in the state, and an entity equation $x = y$ holds if and only if x and y are mapped to the same entity. In turn, the basic propositions determine the validity of arbitrary formulae.

There are, however, two obstacles to this principle, the first of which is slight and the other more difficult to overcome:

- It is not always uniquely determined whether or not an entity is fresh in a state. Our model allows states in which a given entity is considered fresh when arriving by one incoming transition (since it is not in the codomain of the reallocation associated with that transition), but not when arriving by another (since according to the reallocation, the entity is the image of an entity in the previous state).

This obstacle is dealt with by *duplicating* the states where such an ambiguity exists. In general, there will therefore be as many duplicates of a given state as it has incoming transitions with distinct codomains.

- For variables (of the formula in question) that are mapped to the black hole, entity equations are not resolved, since it is not clear whether the variables are mapped to distinct entities that have imploded into the black hole, or to the same one.

To deal with this obstacle, we introduce an intermediate layer in the evaluation of the formula on the state. This additional layer consists of a *partial partitioning* of the free variables; that is, a set of nonempty, disjoint subsets of the set of all free variables. An entity equation is then resolved by the question whether the equated variables are in the same partition. It is the partitions, rather than the individual variables, that are mapped to the entities of the state.

4.5.1 Duplication

The first aforementioned problem is illustrated by the HABA \mathcal{H} in Figure 4.15. Here, entity e_3 in state q_2 is old if transition $q_1 \rightarrow_{\lambda_{12}} q_2$ is taken. On the contrary, e_3 is new in q_2 if we consider the transition $q_3 \rightarrow_{\lambda_{32}} q_2$ (where $\lambda_{32} = \emptyset$). Furthermore, in the initial state q_3 the entity e_2 is new. However, it becomes old after the transition $q_4 \rightarrow_{\lambda_{43}} q_3$.

Definition 4.5.1. For a HABA $\mathcal{H} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$, the *duplication* of \mathcal{H} is the HABA $\mathcal{H}_\delta = \langle X, Q', E', \rightarrow', I', \mathcal{F}' \rangle$ where

- $Q' = \{(q, E_q \setminus \text{cod}(\lambda)) \mid q \in Q \wedge q' \rightarrow_\lambda q\} \cup \{(q, N) \mid I(q) = (N, \theta)\}$;
- $E'(q, M) = E_q$;
- \rightarrow' is defined by the following rule:

$$\frac{q \rightarrow_\lambda q'}{(q, M) \rightarrow'_\lambda (q', E_{q'} \setminus \text{cod}(\lambda))}$$

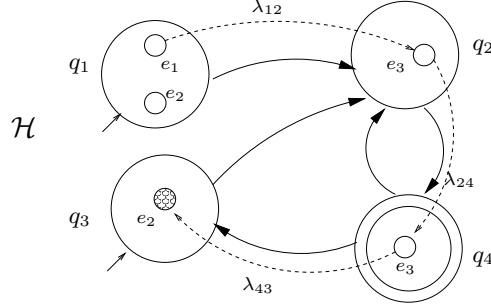


Figure 4.15: Ambiguity of the “new” entities: e_3 can be either new or old depending from the incoming transition.

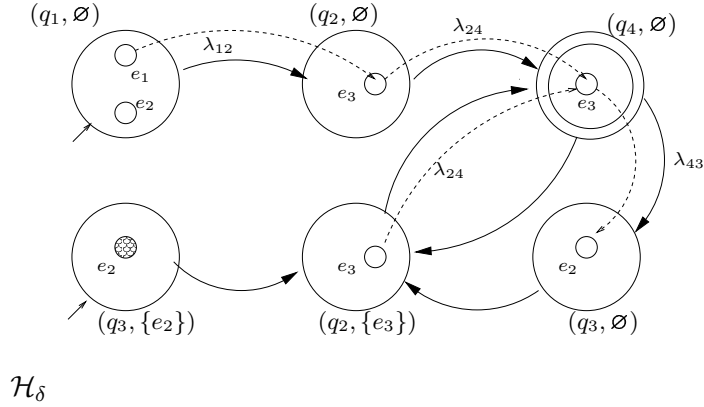


Figure 4.16: The duplication \mathcal{H}_δ of the HABA in Fig. 4.15.

- $I' : (q, N) \mapsto \begin{cases} (N, \theta) & \text{if } I(q) = (N, \theta) \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\mathcal{F}' = \{ \{ (q, M) \in Q' \mid q \in F_i \} \mid F_i \in \mathcal{F} \}$.

The set of variables X is unchanged. States are pairs (q, M) where $M \subseteq E_q$ is the subset of entities that are considered new in (q, M) . Initial states are defined according to initial valuations in I . In the definition of the transition relation, for every $q \rightarrow_\lambda q'$ in \mathcal{H} , a corresponding transition $(q, M) \rightarrow_\lambda (q', M')$ in \mathcal{H}_δ is defined, provided that M' corresponds precisely to the set of entities that are new according to λ , that is $M' = E_{q'} \setminus \text{cod}(\lambda)$. The set \mathcal{F}' contains those states resulting by the duplication of the original accept states.

Example 4.5.2. The duplication \mathcal{H}_δ of the HABA in Figure 4.15 is shown in Figure 4.16. The original state q_2 is duplicated in (q_2, \emptyset) (where e_3 is old) and

$(q_2, \{e_3\})$ (where e_3 is new). The initial state $(q_3, \{e_2\})$ is explicitly added in order to have e_2 new. \square

A HABA and its duplication are equivalent in the sense of the following lemma.

Lemma 4.5.3. $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{H}_\delta)$.

Proof. See Appendix A.4. \square

Assumptions. In the remainder of this chapter as well as in Appendix A where the proofs are reported, we assume that the necessary duplication has been carried out already: that is, we will assume that a state $q \in Q$ is a pair where the second component is a set $N_q \subseteq E_q$ that contains the entities that are *new in* q ; i.e., such that

- $q' \rightarrow_\lambda q$ implies $E_q \setminus \text{cod}(\lambda) = N_q$
- $I(q) = (N, \theta)$ implies $N = N_q$.

Note that, we can henceforth assume that I has just θ as its image — the component N is now uniquely associated with q .

Another assumption needed below is that every quantified variable actually appears free in the sub-formula; that is, we only consider formulae $\exists x.\phi$ for which $x \in \text{fv}(\phi)$. Note that this imposes no real restriction, since $\exists x.\phi$ is equivalent to $\exists x.(x \text{ alive} \wedge \phi)$.

Before we can present the model checking construction, we have to introduce a number of auxiliary notions.

4.5.2 Valuations

A valuation of a formula in a given state is an interpretation of the free variables of the formula as entities of the state. Such an interpretation establishes the validity of at least the *atomic propositions* within the formula, i.e., the sub-formulae of the form $x = y$ (which holds if x and y are interpreted as the same entity) and $x \text{ new}$ (which holds if x is interpreted as a fresh entity). Because we also want to allow the black hole as an interpretation, though, it is not enough to have a simple mapping from variables to entities: such a mapping still would not reveal whether two entities mapped to the black hole are mapped to *the same instance* imploded into the black hole. Therefore we first collect the variables into disjoint sets, the elements of which are considered equal, and map these sets of variables to entities.

Definition 4.5.4 (Valuations). Let $E \subseteq \text{Ent}^\infty$. An E -valuation is a triple (ϕ, Ξ, Θ) where ϕ is an $\mathcal{A}\ell\ell\text{TL}$ -formula and

- Ξ is a partial partitioning of $\text{fv}(\phi)$; that is, $\Xi = \{X_1, \dots, X_n\}$ such that $\emptyset \subset X_i \subseteq \text{fv}(\phi)$ for $1 \leq i \leq n$ and $X_i \cap X_j = \emptyset$ for $1 \leq i < j \leq n$ (but not necessarily $\bigcup_i X_i = \text{fv}(\phi)$, which would make it a *full* partitioning).

- $\Theta: \Xi \rightarrow E$ is a function mapping the partitions of Ξ to E , such that Θ is injective whenever it maps away from ∞ — i.e., $\Theta(X_i) = \Theta(X_j) \neq \infty$ implies $i = j$.

This is easily lifted to the states of a HABA: (ϕ, Ξ, Θ) is a q -valuation (for some $q \in Q_{\mathcal{H}}$) if it is an E_q -valuation (if $\lceil q \rceil$) or E_q^∞ -valuation (if $\lfloor q \rfloor$). We write $V_q(\phi)$, ranged over by v , to denote the set of q -valuations of ϕ , and V_q to denote the set of *all* q -valuations.

Preliminary notations. We denote the components of a valuation v as $(\phi_v, \Xi_v, \Theta_v)$. From a partition interpretation Θ we can easily construct a “proper” (partial) interpretation $\bar{\Theta}: fv(\phi) \rightarrow Ent^\infty$ by *flattening*¹⁴ Θ :

$$\bar{\Theta}: x \mapsto \Theta(X) \quad \text{if } x \in X \in \text{dom}(\Theta). \quad (4.12)$$

A technicality: below we will need to restrict partial partitionings Ξ and mappings Θ of a valuation (ϕ, Ξ, Θ) to sub-formulae of ϕ , which means restricting the underlying sets of (free) variables upon which Ξ and Θ are built to those of that sub-formula. For this purpose, we define

$$\begin{aligned} \Xi \upharpoonright \psi &= \{X \cap fv(\psi) \mid X \in \Xi, X \cap fv(\psi) \neq \emptyset\} \\ \Theta \upharpoonright \psi &= \{(X \cap fv(\psi), \Theta(X)) \mid X \in \text{dom}(\Theta), X \cap fv(\psi) \neq \emptyset\} . \end{aligned}$$

We can now define the *atomic proposition valuations* of a state q of a HABA. These are those q -valuations of basic $\mathcal{A}llTL$ propositions (i.e., freshness predicates and entity equations) that make the corresponding properties true.

Definition 4.5.5. Let \mathcal{H} be a HABA and $q \in Q_{\mathcal{H}}$. The *atomic proposition valuations* of q are defined by the set $AV_q \subseteq V_q$ of all triples (ϕ, Ξ, Θ) for which one of the following holds:

- $\phi = \text{tt}$;
- $\phi = (x = y)$, and $x, y \in X$ for some $X \in \Xi$;
- $\phi = (x \text{ new})$, and $x \in X$ for some $X \in \Xi$ such that $\Theta(X) \in N_q$.

In our setting, it is straightforward to see that $x = y$ is true in v if both x and y belong to the same set. Similarly $x \text{ new}$ holds when the set X — containing x — is mapped by Θ to some fresh entity in N_q .

Closure. Along the lines of [77], we associate to each state q of a HABA a set of q -valuations, specifically aimed at establishing the validity of a given formula ϕ . For this purpose, we first collect all $\mathcal{A}llTL$ -formulae whose validity is possibly relevant to the validity of ϕ into the so-called *closure* of ϕ . This includes especially all sub-formulae of ϕ , but also $\neg\psi$ if ψ is in the closure, $X\neg\psi$ if $X\psi$ is in the closure, and $X(\psi_1 \cup \psi_2)$ if $\psi_1 \cup \psi_2$ is in the closure. Formally:

¹⁴Here the term “flattening” does not have to be confused with the same term used in the OCL jargon. There flattening stands for an automatic mechanism that deals with nested collection types (cf. Section 3.4.3).

Definition 4.5.6. Let ϕ be an $AllTL$ -formula. The *closure* of ϕ , $CL(\phi)$, is the smallest set of formulae (identifying $\neg\neg\psi$ with ψ) such that:

- $\phi, \text{tt}, \text{ff} \in CL(\phi)$;
- $\neg\psi \in CL(\phi)$ iff $\psi \in CL(\phi)$;
- if $\psi_1 \vee \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2 \in CL(\phi)$;
- if $\exists x.\psi \in CL(\phi)$ then $\psi \in CL(\phi)$;
- if $\mathbf{X}\psi \in CL(\phi)$ then $\psi \in CL(\phi)$;
- if $\neg\mathbf{X}\psi \in CL(\phi)$ then $\mathbf{X}\neg\psi \in CL(\phi)$;
- if $\psi_1 \mathbf{U} \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2, \mathbf{X}(\psi_1 \mathbf{U} \psi_2) \in CL(\phi)$.

Example 4.5.7. The closure of the formula $\phi_1 \equiv \exists x.(x \text{ new} \wedge x \neq y)$ is:

$$CL(\phi_1) = \{\text{tt}, \text{ff}, \phi_1, \neg\phi_1, (x \text{ new} \wedge x \neq y), \\ \neg(x \text{ new} \wedge x \neq y), x \text{ new}, \neg(x \text{ new}), x \neq y, x = y\}.$$

Similarly, the closure of $\phi_2 \equiv \mathbf{X}\exists x.x \neq y$ is:

$$CL(\phi_2) = \{\text{tt}, \text{ff}, \phi_2, \neg\phi_2, \mathbf{X}\neg(\exists x.x \neq y), \\ \neg\mathbf{X}\neg(\exists x.x \neq y), \exists x.x \neq y, \neg(\exists x.x \neq y), x \neq y, x = y\}.$$

□

Since valuations map (sets of) variables of a given formula to entities, possibly to the black hole, it is important to know how many of these variables have to be taken into account at most. This is obviously bounded by the number of variables occurring (free or bound) in ϕ , but in fact we can be a little more precise: the number is given by $K(\phi)$ defined as

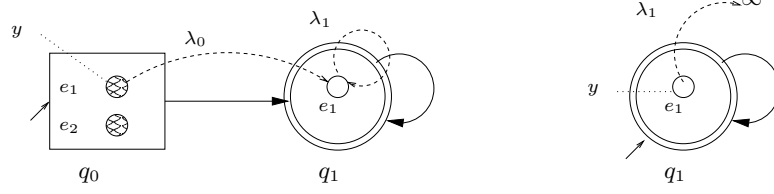
$$K(\phi) = \max \{ |fv(\psi)| \mid \psi \in CL(\phi) \} . \quad (4.13)$$

The interesting case for the model checking construction is when one or more variables are indeed mapped to the black hole. Among other things, we will then have to make sure that sufficiently many entities of the state have imploded into the black hole to meet the demands of the valuation. For this purpose, we introduce the *black number* of a function, which is the number of entities that such a function maps (implodes) into the black hole. For an arbitrary set A and (partial) mapping $\alpha: A \rightarrow Ent^\infty$ this is defined by

$$\Omega(\alpha) = |\{a \in A \mid \alpha(a) = \infty\}| . \quad (4.14)$$

4.5.3 Tableau graph for $AllTL$

We now construct a graph that will be — as anticipated in Section 2.1.6 — the basis of the model checking algorithm. The main property of the tableau graph is that a model for the formula ϕ can be extracted from it if and only

Figure 4.17: HABA \mathcal{H}_1 (left) and \mathcal{H}_2 (right).

if the formula is satisfiable in \mathcal{H} . The nodes of this graph, called *atoms* after [77], are built from states of a HABA, valuations of formulae from the closure, and a bound on the black number.

Definition 4.5.8. Given a HABA \mathcal{H} and an $\mathcal{A}ll\mathcal{T}L$ -formula ϕ , an *atom* is a triple (q, D, k) where $q \in Q_{\mathcal{H}}$, $D \subseteq \{v \in V_q(\psi) \mid \psi \in CL(\phi), \Omega(\Theta_v) \leq k\}$ and $k \leq K(\phi)$ if $\lfloor q \rfloor$ or $k = 0$ if $\lceil q \rceil$, such that for all $v = (\psi, \Xi, \Theta) \in V_q$ with $\psi \in CL(\phi)$ and $\Omega(\Theta) \leq k$:

- if $v \in AV_q$, then $v \in D$;
- if $\psi = \neg\psi'$, then $v \in D$ iff $(\psi', \Xi, \Theta) \notin D$;
- if $\psi = \psi_1 \vee \psi_2$, then $v \in D$ iff $(\psi_i, \Xi \upharpoonright \psi_i, \Theta \upharpoonright \psi_i) \in D$ for $i = 1$ or $i = 2$;
- if $\psi = \exists x.\psi'$, then $v \in D$ iff there exists a $(\psi', \Xi', \Theta') \in D$ such that $\Xi = \Xi' \upharpoonright \psi$, $\Theta = \Theta' \upharpoonright \psi$ and $x \in \bigcup \Xi'$;
- if $\psi = \neg X\psi'$, then $v \in D$ iff $(X\neg\psi', \Xi, \Theta) \in D$;
- if $\psi = \psi_1 \cup \psi_2$, then $v \in D$ iff either $(\psi_2, \Xi \upharpoonright \psi_2, \Theta \upharpoonright \psi_2) \in D$, or both $(\psi_1, \Xi \upharpoonright \psi_1, \Theta \upharpoonright \psi_1) \in D$ and $(X\psi, \Xi, \Theta) \in D$.

The set of all atoms for a given formula ϕ constructed on top of \mathcal{H} is denoted $A_{\mathcal{H}}(\phi)$, ranged over by A, B . We denote the components of an atom A by (q_A, D_A, k_A) .

Example 4.5.9. Consider the HABA \mathcal{H}_1 depicted in the left part of Figure 4.17 where e_1 and e_2 are new in q_0 and $X = \{y\}$. Recall the formula

$$\phi_1 \equiv \exists x.(x \text{ new} \wedge x \neq y)$$

and its closure from Example 4.5.7. We compute the set of atoms $A_{\mathcal{H}_1}(\phi_1)$. Since $\lceil q_0 \rceil$, for q_0 there is only one atom which is of the form $(q_0, D_0, 0)$. According to Def. 4.5.5 for the component D_0 , the atomic proposition valuation $(\psi, \Xi, \Theta) \in AV_{q_0}$ such that $\psi \in CL(\phi_1)$ are v_0, v_1, v_2, v_3, v_4 reported in Table 4.5 and by the first clause of Def. 4.5.8 they are contained in D_0 . By the clause for negation of Def. 4.5.8 we have that also $v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12} \in$

D_0 . Furthermore,

$$\begin{aligned} v_6, v_1 \in D_0 &\Rightarrow v_{13} \in D_0 \\ v_7, v_2 \in D_0 &\Rightarrow v_{14} \in D_0 \\ v_{10}, v_1 \in D_0 &\Rightarrow v_{15} \in D_0 \\ v_{11}, v_2 \in D_0 &\Rightarrow v_{16} \in D_0. \end{aligned}$$

By negation, we obtain $v_{17}, v_{18}, v_{19}, v_{20}, v_{21} \in D_0$, and by the clause on existential quantification:

$$\begin{aligned} v_{13}, v_{14} \in D_0 &\Rightarrow v_{22} \in D_0 \\ v_{15} \in D_0 &\Rightarrow v_{23} \in D_0 \\ v_{16} \in D_0 &\Rightarrow v_{24} \in D_0. \end{aligned}$$

Note that in D_0 there are no valuations for $\neg\exists x.(x \text{ new} \wedge x \neq y)$.

For state q_1 we have three atoms:

$$(q_1, D_1, 0), \quad (q_1, D_2, 1), \quad (q_1, D_3, 2).$$

Sets D_1 , D_2 and D_3 differ because of the black number. In fact, valuation (ψ, Ξ, Θ) in D_1 must have $\Omega(\Theta) = 0$, while in D_2 and D_3 it must be $\Omega(\Theta) \leq 1$ and $\Omega(\Theta) \leq 2$, respectively. D_1 , D_2 , and D_3 are computed following the same pattern used for D_0 (and in fact many valuations $v \in D_0$ are also in D_1 , D_2 and D_3). In particular, D_1 valuations are indicated in Table 4.6. Note that there are no valuations $(x \text{ new}, \Xi, \Theta)$ for any Ξ, Θ . Therefore, by negation, in D_1 (see Table 4.6) we have

$$v_{17}, v_{18}, v_{20}, v_{27}, v_{25}, v_{26} \in D_1$$

Since there are no valuations for $x \text{ new} \wedge x \neq y$, it implies there are no valuations for $\exists x.x \text{ new} \wedge x \neq y$. This, in turn, implies by negation: $v_{28}, v_{29} \in D_1$. For the second atom of q_1 we have

$$D_2 = D_1 \cup B_1$$

where B_1 (valuations with black number 1) contains the valuations reported in Table 4.7. Finally,

$$D_3 = D_2 \cup B_2$$

where B_2 (valuations with black number 2) contains the valuations in Table 4.8. As for D_1 , both D_2 and D_3 do not contain any valuation of the kind $(\exists x.(x \text{ new} \wedge x \neq y), \Xi, \Theta)$. In fact in q_1 , the formula ϕ_1 does not hold. \square

Example 4.5.10. Using the valuations v of the previous example, we discuss a more interesting case. Consider the formula $\phi_2 \equiv \mathbf{X}\exists x.x \neq y$ and its closure from Example 4.5.7. The set of atoms is

$$\begin{aligned} A_{\mathcal{H}_1}(\phi_2) = &\{(q_0, D_4, 0), (q_0, D'_4, 0), (q_1, D_5, 0), (q_1, D'_5, 0), \\ &(q_1, D_6, 1), (q_1, D'_6, 1), (q_1, D_7, 2), (q_1, D'_7, 2)\}. \end{aligned}$$

D_0	
v_0	$= (\text{tt}, \emptyset, \emptyset)$
v_1	$= (x \text{ new}, \{\{x\}\}, \{x\} \mapsto e_1)$
v_2	$= (x \text{ new}, \{\{x\}\}, \{x\} \mapsto e_2)$
v_3	$= (x = y, \{\{x, y\}\}, \{x, y\} \mapsto e_1)$
v_4	$= (x = y, \{\{x, y\}\}, \{x, y\} \mapsto e_2)$
v_5	$= (x \neq y, \emptyset, \emptyset)$
v_6	$= (x \neq y, \{\{x\}\}, \{x\} \mapsto e_1)$
v_7	$= (x \neq y, \{\{x\}\}, \{x\} \mapsto e_2)$
v_8	$= (x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_9	$= (x \neq y, \{\{y\}\}, \{y\} \mapsto e_2)$
v_{10}	$= (x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto e_1, \{y\} \mapsto e_2)$
v_{11}	$= (x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto e_2, \{y\} \mapsto e_1)$
v_{12}	$= (\neg(x \text{ new}), \emptyset, \emptyset)$
v_{13}	$= (x \text{ new} \wedge x \neq y, \{\{x\}\}, \{x\} \mapsto e_1)$
v_{14}	$= (x \text{ new} \wedge x \neq y, \{\{x\}\}, \{x\} \mapsto e_2)$
v_{15}	$= (x \text{ new} \wedge x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto e_1, \{y\} \mapsto e_2)$
v_{16}	$= (x \text{ new} \wedge x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto e_2, \{y\} \mapsto e_1)$
v_{17}	$= (\neg(x \text{ new} \wedge x \neq y), \emptyset, \emptyset)$
v_{18}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$
v_{19}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_2)$
v_{20}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x, y\}\}, \{x, y\} \mapsto e_1)$
v_{21}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x, y\}\}, \{x, y\} \mapsto e_2)$
v_{22}	$= (\exists x.(x \text{ new} \wedge x \neq y), \emptyset, \emptyset)$
v_{23}	$= (\exists x.(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_2)$
v_{24}	$= (\exists x.(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$

Table 4.5: Valuations in the set D_0 .

We compute its elements in details. Instead of a single atom for q_0 , since ϕ_2 involves a next operator, we construct two atoms $(q_0, D_4, 0)$, $(q_0, D'_4, 0)$. In fact a formula of the kind $X\psi$ does not imply anything concerning the validity of ψ in q_0 . This is also the reason why in the Definition 4.5.8 we do not have a special clause for the X operator. Thus we distinguish two possibilities, one in which $X\psi$ holds (case D_4) and one where $\neg X\psi$ holds (case D'_4). By the same argument as the previous example, we have

$$v_0, v_3, \dots, v_{11} \in D_4, D'_4.$$

D_1	
v_0	$= (\mathbf{tt}, \emptyset, \emptyset)$
v_3	$= (x = y, \{\{x, y\}\}, \{x, y\} \mapsto e_1)$
v_5	$= (x \neq y, \emptyset, \emptyset)$
v_6	$= (x \neq y, \{\{x\}\}, \{x\} \mapsto e_1)$
v_8	$= (x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{17}	$= (\neg(x \text{ new} \wedge x \neq y), \emptyset, \emptyset)$
v_{18}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$
v_{20}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x, y\}\}, \{x, y\} \mapsto e_1)$
v_{27}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}\}, \{x\} \mapsto e_1)$
v_{25}	$= (\neg(x \text{ new}), \emptyset, \emptyset)$
v_{26}	$= (\neg(x \text{ new}), \{\{x\}\}, \{x\} \mapsto e_1)$
v_{28}	$= (\neg\exists x.(x \text{ new} \wedge x \neq y), \emptyset, \emptyset)$
v_{29}	$= (\neg\exists x.(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto e_1).$

Table 4.6: Valuations set D_1

B_1	
v_{30}	$= (x = y, \{\{x, y\}\}, \{x, y\} \mapsto \infty)$
v_{31}	$= (x \neq y, \{\{x\}\}, \{x\} \mapsto \infty)$
v_{32}	$= (x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$
v_{33}	$= (x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto e_1, \{y\} \mapsto \infty)$
v_{34}	$= (x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto \infty, \{y\} \mapsto e_1)$
v_{35}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}\}, \{x\} \mapsto \infty)$
v_{36}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto \infty)$
v_{37}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x, y\}\}, \{x, y\} \mapsto \infty)$
v_{38}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}, \{y\}\}, \{x\} \mapsto e_1, \{y\} \mapsto \infty)$
v_{39}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}, \{y\}\}, \{x\} \mapsto \infty, \{y\} \mapsto e_1)$
v_{40}	$= (\neg\exists x.(x \text{ new} \wedge x \neq y), \{\{y\}\}, \{y\} \mapsto \infty).$

Table 4.7: Valuations set B_1 .

B_2	
v_{41}	$= (x \neq y, \{\{x\}, \{y\}\}, \{x\} \mapsto \infty, \{y\} \mapsto \infty)$
v_{42}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}, \{y\}\}, \{x\} \mapsto \infty, \{y\} \mapsto \infty)$
v_{43}	$= (\neg(x \text{ new} \wedge x \neq y), \{\{x\}, \{y\}\}, \{x\} \mapsto \infty, \{y\} \mapsto \infty).$

Table 4.8: Valuations set B_2 .

D_4	
$v_0, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}$	
v_{44}	$= (\exists x.x \neq y, \emptyset, \emptyset)$
v_{45}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_2)$
v_{46}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{47}	$= (\mathbf{X}\exists x.x \neq y, \emptyset, \emptyset)$
v_{48}	$= (\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{49}	$= (\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_2)$
v_{56}	$= (\neg\mathbf{X}\neg(\exists x.x \neq y), \emptyset, \emptyset)$
v_{57}	$= (\neg\mathbf{X}\neg(\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$
v_{58}	$= (\neg\mathbf{X}\neg(\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_2)$
D'_4	
$v_0, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}$	
v_{44}	$= (\exists x.x \neq y, \emptyset, \emptyset)$
v_{45}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_2)$
v_{46}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{50}	$= (\neg\mathbf{X}\exists x.x \neq y, \emptyset, \emptyset)$
v_{51}	$= (\neg\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{52}	$= (\neg\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_2)$
v_{53}	$= (\mathbf{X}\neg(\exists x.x \neq y), \emptyset, \emptyset)$
v_{54}	$= (\mathbf{X}\neg(\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$
v_{55}	$= (\mathbf{X}\neg(\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_2)$

Table 4.9: Valuations set D_4 and D'_4 .

Moreover,

$$\begin{aligned}
v_6, v_7 \in D_4, D'_4 &\Rightarrow v_{44} \in D_4, D'_4 \\
v_{10} \in D_4, D'_4 &\Rightarrow v_{45} \in D_4, D'_4 \\
v_{11} \in D_4, D'_4 &\Rightarrow v_{46} \in D_4, D'_4.
\end{aligned}$$

This implies that in D_4 and D'_4 there are no triples of the form $(\neg\exists x.x \neq$

D_5	D'_5
$(\text{tt}, \emptyset, \emptyset)$	$(\text{tt}, \emptyset, \emptyset)$
$(x = y, \{\{x, y\}\}, \{x, y\} \mapsto e_1)$	$(x = y, \{\{x, y\}\}, \{x, y\} \mapsto e_1)$
$(x \neq y, \emptyset, \emptyset)$	$(x \neq y, \emptyset, \emptyset)$
$(x \neq y, \{\{x\}\}, \{x\} \mapsto e_1)$	$(x \neq y, \{\{x\}\}, \{x\} \mapsto e_1)$
$(x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$	$(x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
$(\exists x.x \neq y, \emptyset, \emptyset)$	$(\exists x.x \neq y, \emptyset, \emptyset)$
$(\neg \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$	$(\neg \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
$(\mathbf{X} \exists x.x \neq y, \emptyset, \emptyset)$	$(\neg \mathbf{X} \exists x.x \neq y, \emptyset, \emptyset)$
$(\mathbf{X} \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$	$(\neg \mathbf{X} \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
$(\neg \mathbf{X} \neg (\exists x.x \neq y), \emptyset, \emptyset)$	$(\mathbf{X} \neg (\exists x.x \neq y), \emptyset, \emptyset)$
$(\neg \mathbf{X} \neg (\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$	$(\mathbf{X} \neg (\exists x.x \neq y), \{\{y\}\}, \{y\} \mapsto e_1)$

Table 4.10: Set of valuations D_5 and D'_5 in Example 4.5.9.

y, Ξ, Θ) since all possible partitionings of $fv(\exists x.x \neq y)$ are in valuations with the formula $\exists x.x \neq y$. Note how D_4 has valuations with \mathbf{X} (i.e. $v_{47}, v_{48}, v_{49} \in D_4$) whereas D'_4 those with $\neg \mathbf{X}$ (i.e., $v_{50}, v_{51}, v_{52} \in D'_4$) which in turn implies (by definition of atom):

$$\begin{aligned} v_{56}, v_{57}, v_{58} &\in D_4 \\ v_{53}, v_{54}, v_{55} &\in D'_4 \end{aligned}$$

respectively. The atoms of state q_1 are¹⁵: $(q_1, D_5, 0)$, $(q_1, D'_5, 0)$, $(q_1, D_6, 1)$, $(q_1, D'_6, 1)$, $(q_1, D_7, 2)$, $(q_1, D'_7, 2)$. The computation of D_5 and D'_5 is similar to D_4 and D'_4 , therefore we skip intermediate steps and we indicate only the resulting valuations in Table 4.10 (note that for both D_5 and D'_5 the first valuations correspond to v_0, v_3, v_5, v_6, v_8).

For D_6 and D'_6 we take

$$D_5 \setminus \{(\neg \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)\} \text{ and } D'_5 \setminus \{(\neg \exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)\}$$

respectively, and we extend them in order to include valuations with black number 1 (recall that atoms with D_6 and D'_6 have $\Omega(\Theta) = 1$), that are

$$v_{30}, v_{31}, v_{32}, v_{33}, v_{34} \in D_6, D'_6$$

(they are reported in Table 4.7) and moreover

$$\begin{aligned} v_{33} \in D_6, D'_6 &\Rightarrow v_{59} \in D_6, D'_6 \\ v_{34} \in D_6, D'_6 &\Rightarrow v_{60} \in D_6, D'_6. \end{aligned}$$

In particular, in D_6 and D'_6 there are no triples $(\neg(\exists x.x \neq y), \Xi, \Theta)$ for any Ξ , and Θ . This conforms with the intuition that the black number is 1, i.e., there exists one entity in the black hole distinct from e_1 . Thus $\neg \exists x.x \neq y$ cannot hold in q_1 .

¹⁵Again the primed version of a set D will be used for formulae of the kind $\neg \mathbf{X}\psi$.

D_6	
$D_5 \setminus \{(\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)\}$	
$v_{30}, v_{31}, v_{32}, v_{33}, v_{34}$	
v_{59}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$
v_{60}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{61}	$= (\mathbf{X}\exists x.x \neq y, \emptyset, \emptyset)$
v_{62}	$= (\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{63}	$= (\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$
v_{64}	$= (\neg\mathbf{X}\neg\exists x.x \neq y, \emptyset, \emptyset)$
v_{65}	$= (\neg\mathbf{X}\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{66}	$= (\neg\mathbf{X}\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$

D'_6	
$D'_5 \setminus \{(\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)\}$	
$v_{30}, v_{31}, v_{32}, v_{33}, v_{34}$	
v_{59}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$
v_{60}	$= (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{67}	$= (\neg\mathbf{X}\exists x.x \neq y, \emptyset, \emptyset)$
v_{68}	$= (\neg\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{69}	$= (\neg\mathbf{X}\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$
v_{70}	$= (\mathbf{X}\neg\exists x.x \neq y, \emptyset, \emptyset)$
v_{71}	$= (\mathbf{X}\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$
v_{72}	$= (\mathbf{X}\neg\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto \infty)$

Table 4.11: Valuations sets D_6 and D'_6 .

Again note that we distinguish between D_6 and D'_6 for valuations with \mathbf{X} and $\neg\mathbf{X}$. In particular:

$$\begin{aligned} v_{61}, v_{62}, v_{63}, v_{64}, v_{65}, v_{66} &\in D_6 \\ v_{67}, v_{68}, v_{69}, v_{70}, v_{71}, v_{72} &\in D'_6. \end{aligned}$$

Finally, it is easy to define D_7 and D'_7 :

$$D_7 = D_6 \cup \{v_{41}\} \text{ and } D'_7 = D'_6 \cup \{v_{41}\}.$$

□

Definition 4.5.11. The *tableau graph* for a HABA \mathcal{H} and an $\mathcal{A}\ell\ell\text{TL}$ -formula ϕ , denoted $G_{\mathcal{H}}(\phi)$, consists of vertices $A_{\mathcal{H}}(\phi)$ and edges $\rightarrow \subseteq A_{\mathcal{H}}(\phi) \times (Ent^\infty \rightarrow Ent^\infty) \times A_{\mathcal{H}}(\phi)$ determined by

$$\begin{aligned} (q, D, k) \rightarrow_\lambda (q', D', k') \quad \text{iff} \quad & q \rightarrow_\lambda q', \\ & \forall \mathbf{X}\psi \in CL(\phi): (\mathbf{X}\psi, \Xi, \Theta) \in D \Leftrightarrow (\psi, \Xi, \lambda \circ \Theta) \in D', \\ & k' = \begin{cases} \min(K(\phi), k + \Omega(\lambda)) & \text{if } \lfloor q' \rfloor \\ 0 & \text{if } \lceil q' \rceil. \end{cases} \end{aligned}$$

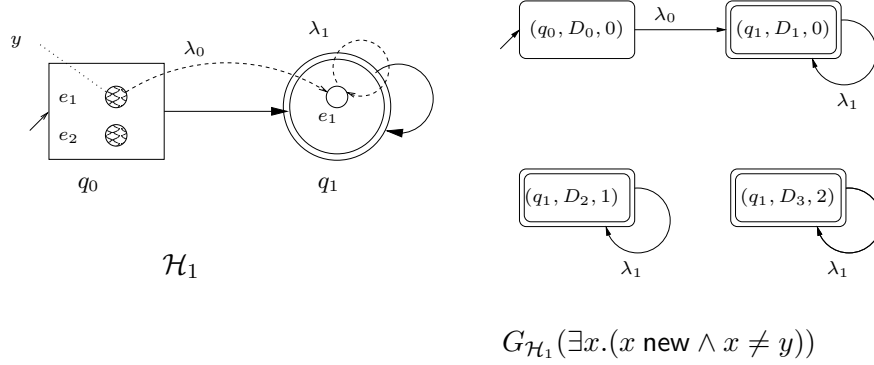


Figure 4.18: HABA \mathcal{H}_1 and corresponding graph $G_{\mathcal{H}_1}(\exists x.(x \text{ new} \wedge x \neq y))$.

In the tableau graph $G_{\mathcal{H}}(\phi)$ there exists a transition between atoms (q, D, k) and (q', D', k') if in \mathcal{H} this transition exists between the underlying states q and q' . In this way, $G_{\mathcal{H}}(\phi)$ mimics \mathcal{H} . Furthermore, the transition must be sound w.r.t. the set of valuations contained in the two atoms. More precisely, according to the second condition, every valuation with an outer-most next operator such as $(X\psi, \Xi, \Theta) \in D$ must correspond to a valuation $(\psi, \Xi, \Theta') \in D'$ and vice-versa. Note that in order to be sound w.r.t. the $\mathcal{A}\ell\ell\text{TL}$ semantics, the two functions Θ and Θ' — mapping the partitions of Ξ onto entities — must be consistent according to the reallocation λ , i.e., $\Theta' = \lambda \circ \Theta$. Finally, the last condition gives a constraint on the number of entities in the black hole assumed by the two atoms, i.e., k and k' . In particular, if q' is not bounded, k' corresponds to the sum (up to the upper bound $K(\phi)$) of the number of entities (i.e., k) in the black hole of the source state and the number of entities that implode during the transition. The latter is the black number of the reallocation, i.e. $\Omega(\lambda)$.

Note that if \mathcal{H} is finite-state, then $G_{\mathcal{H}}(\phi)$ can be effectively constructed: the set of atoms is finite for every given state due to the bound $K(\phi)$.

Example 4.5.12. The graph $G_{\mathcal{H}_1}(\exists x.(x \text{ new} \wedge x \neq y))$ for the set of atoms computed in Example 4.5.9 is shown in Figure 4.18 while $G_{\mathcal{H}_1}(X\exists x.x \neq y)$ for the set of atoms of Example 4.5.10 is shown in Figure 4.19. Atoms corresponding to accept states of the original HABA are drawn with a double circle¹⁶. In $G_{\mathcal{H}_1}(X\exists x.x \neq y)$, there are only four transitions (self-loops in atoms $(q_1, D_6, 1)$ and $(q_1, D_7, 2)$ and $(q_1, D_6, 1) \rightarrow_{\lambda_1} (q_1, D'_6, 1)$ and $(q_1, D_7, 2) \rightarrow_{\lambda_1} (q_1, D'_7, 2)$). No other transitions can be set because either the condition on the X operator or the condition on the black number of Def 4.5.11 is violated in any other case. For example, valuation $v_{48} = (X\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$ is in D_4 ,

¹⁶Formally speaking a tableau graph does not have accept states. However, here we distinguish these atoms from the others only in order to facilitate their identification that will be useful later.

however, $(\exists x.x \neq y, \{\{y\}\}, \lambda_0 \circ \{y\} \mapsto e_1) = (\exists x.x \neq y, \{\{y\}\}, \{y\} \mapsto e_1)$ does not belong to D_5 , therefore there is no transition $(q_0, D_4, 0) \rightarrow_{\lambda_0} (q_1, D_5, 0)$. While, $(q_0, D_4, 0) \rightarrow_{\lambda_0} (q_1, D_6, 1)$ because the condition on the k component is not fulfilled as the transition does not implode any entity. \square

Example 4.5.13. The right part of Figure 4.17 shows the HABA \mathcal{H}_2 with the only state q_1 (it represents a modified version of q_1 of \mathcal{H}_1). In this case λ_1 implodes e_1 and therefore in each state a new entity is created. For $\phi_2 \equiv \exists x.x \neq y$ the set of atoms contains precisely those atoms obtained in the previous example for state q_1 . In fact, in the definition of atoms, the transitions of the HABA are not taken into account. Atoms for a given q only depend on E_q , N_q and on the (un)boundedness of q . For \mathcal{H}_2 the resulting graph $G_{\mathcal{H}_2}(\phi_2)$ is depicted in the right part of Figure 4.20. In this case the third component k of the atom plays an active role. Since λ_1 implodes at each step an entity, the possible transitions in the graph are between atoms of the form $(\psi, D, k) \rightarrow_{\lambda_1} (\psi, D, k + 1)$ with $k = 0, 1$. This is because in general with a formula ϕ , we may need to distinguish up to $K(\phi)$ different situations according to how many entities are imploded into the black hole. Above this number we do not longer need to differentiate. This explains the transition $(q_1, D_7, 2) \rightarrow_{\lambda_1} (q_1, D_7, 2)$.

A *path* through a tableau graph is an infinite sequence of states and transitions, starting at an initial state of the HABA and satisfying the acceptance condition of the HABA, such that all “until”-subformulae in any of the atoms are satisfied somewhere further down the sequence.

Definition 4.5.14. An *allocation path* in $G_{\mathcal{H}}(\phi)$ is an infinite sequence $\pi = (q_0, D_0, k_0) \lambda_0 (q_1, D_1, k_1) \lambda_1 \dots$ such that:

1. $q_0 \lambda_0 q_1 \lambda_1 \dots \in \text{runs}(\mathcal{H})$;
2. for all $i \geq 0$, $(q_i, D_i, k_i) \rightarrow_{\lambda_i} (q_{i+1}, D_{i+1}, k_{i+1})$;
3. for all $i \geq 0$ and all $(\psi_1 \cup \psi_2, \Xi, \Theta) \in D_i$, there exists a $j \geq i$ such that $(\psi_2, \Xi \upharpoonright \psi_2, \lambda_{j-1} \circ \dots \circ \lambda_i \circ (\Theta \upharpoonright \psi_2)) \in D_j$.

Definition 4.5.15. Given an allocation path $\pi = (q_0, D_0, k_0) \lambda_0 (q_1, D_1, k_1) \lambda_1 \dots$ in $G_{\mathcal{H}}(\phi)$, we say that π *fulfils* ϕ if the underlying run $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ generates an allocation triple (σ, N, θ) with a generator $(h_i)_{i \in \mathbb{N}}$ such that

- $k_0 = \min(K(\phi), \Omega(h_0))$ and
- $\sigma, N, \theta \models \phi$.

If ϕ is clear from the context, we call π a *fulfilling path*.

Furthermore, recall that (cf. Definition 4.3.11) if there exists $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$ such that $\sigma, N, \theta \models \phi$ we say that ϕ is \mathcal{H} -satisfiable.

This sets the stage for the main results. We first state the correspondence between the fulfilment of a formula by a path and the presence of that formula in the initial atom of the path.

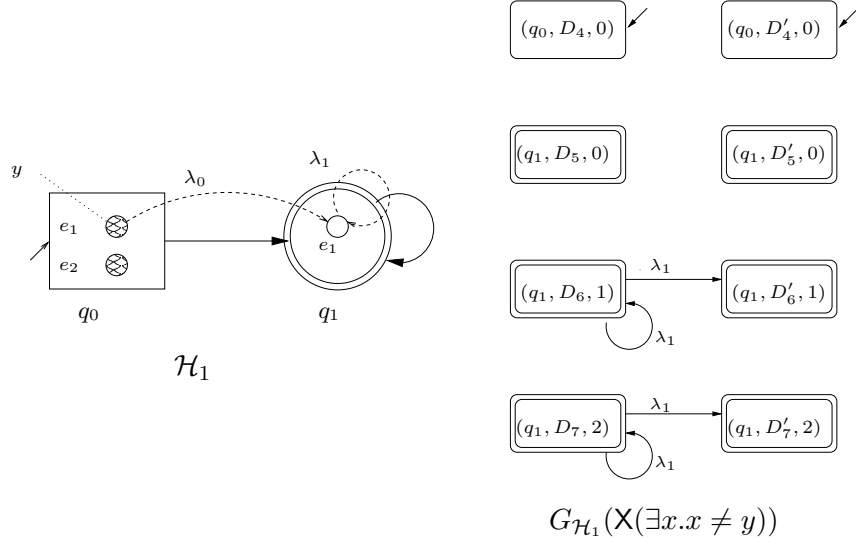


Figure 4.19: HABA \mathcal{H}_1 and corresponding graph $G_{\mathcal{H}_1}(X(\exists x.x \neq y))$.

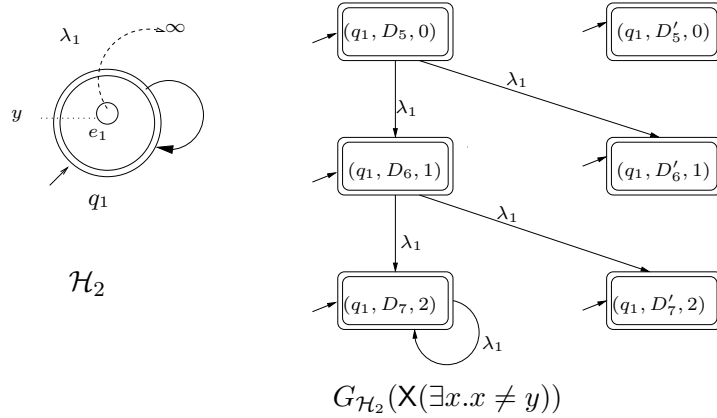


Figure 4.20: HABA \mathcal{H}_2 and corresponding graph $G_{\mathcal{H}_2}(X(\exists x.x \neq y))$.

Proposition 4.5.16. A path π in $G_{\mathcal{H}}(\phi)$ fulfils ϕ if and only if there exists $(\phi, \Xi, \Theta) \in D_0$ (for some Ξ, Θ) such that $I_{\mathcal{H}}(q_0) = \overline{\Theta}$.

Proof. See Appendix A.4. □

Furthermore, there is a correspondence between the satisfiability of a formula in the HABA and the existence of a fulfilling path in the tableau graph.

Proposition 4.5.17. ϕ is \mathcal{H} -satisfiable if and only if there exists a path in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ .

Proof. See Appendix A.4. \square

Example 4.5.18. The path $\pi = (q_0, D_0, 0)\lambda_0((q_1, D_1, 0)\lambda_1)^\omega$ of $G_{\mathcal{H}}(\exists x.(x \text{ new} \wedge x \neq y))$ in Figure 4.18 is fulfilling for $\phi_1 \equiv \exists x.(x \text{ new} \wedge x \neq y)$. On the other hand, $\exists x.x \neq y$ is not \mathcal{H}_1 -satisfiable. In fact, there are no paths in $G_{\mathcal{H}_1}(\exists x.x \neq y)$ (see Figure 4.19). Finally, $\exists x.x \neq y$ is \mathcal{H}_2 -satisfiable. The path $(q_1, D_5, 0)\lambda_1(q_1, D_6, 1)(\lambda_1(q_1, D_7, 2))^\omega$ is a fulfilling path (see Figure 4.20). \square

From now on we can (almost) rely on standard theory (see Section 2.1.6). The first observation is that a tableau graph can have infinitely many different paths, therefore looking for a fulfilling path for ϕ is still not an effective method for model checking. We need the following definitions.

Definition 4.5.19. A subgraph $G' \subseteq G_{\mathcal{H}}(\phi)$ is *self-fulfilling* if every node A in G' has at least an outgoing edge and for every $(\psi_1 \cup \psi_2, \Xi, \Theta) \in D_A$ there exists a node $B \in G'$ such that

- $A = A_0 \rightarrow_{\lambda_0} A_1 \rightarrow_{\lambda_1} \cdots \rightarrow_{\lambda_{i-2}} A_{i-1} \rightarrow_{\lambda_{i-1}} A_i = B$
- $(\psi_2, \Xi \upharpoonright \psi_2, \lambda_{i-1} \circ \cdots \circ \lambda_0 \circ (\Theta \upharpoonright \psi_2)) \in D_B$.

A *prefix* in $G_{\mathcal{H}}(\phi)$ is a sequence $A_0 \rightarrow_{\lambda_0} A_1 \rightarrow_{\lambda_1} \cdots \rightarrow_{\lambda_{i-2}} A_{i-1} \rightarrow_{\lambda_{i-1}} A_i$ such that A_0 is an initial atom (i.e., $q_{A_0} \in I_{\mathcal{H}}$) and A_i is in a self-fulfilling subgraph.

Let $\text{Inf}(\pi)$ denote the set of nodes that appear infinitely often in the path π . $\text{Inf}(\pi)$ is a strongly connected subgraph (SCS). We can prove the following implication:

Proposition 4.5.20. π is a fulfilling path in $G_{\mathcal{H}}(\phi) \Rightarrow \text{Inf}(\pi)$ is a self-fulfilling SCS of $G_{\mathcal{H}}(\phi)$.

Proof. See Appendix A.4. \square

Proposition 4.5.21. Let $G' \subseteq G_{\mathcal{H}}(\phi)$ be a self-fulfilling SCS such that

- there exists a prefix of G' starting at an initial atom A with $(\phi, \Xi, \Theta) \in D_A$ such that $I_{\mathcal{H}}(q_A) = \overline{\Theta}$;
- for all $F \in \mathcal{F}_{\mathcal{H}} : F \cap \{q \mid (q, D, k) \in G'\} \neq \emptyset$;

Then there exists a path π in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ such that $\text{Inf}(\pi) = G'$.

Proof. See Appendix A.4. \square

Finally, we can collect all the previous results into the following theorem, which is the main result of this chapter, stating in essence that the model-checking problem for $\mathcal{A}\ell\ell\text{TTL}$ over HABA is decidable.

Theorem 4.5.22 (Decidibility). For any HABA \mathcal{H} and formula ϕ , it is decidable whether or not ϕ is \mathcal{H} -satisfiable.

Proof. See Appendix A.4. \square

Example 4.5.23. At this point, it becomes interesting to have a final example that summarises the complete model-checking procedure. Consider $\phi_3 \equiv \mathsf{G}(\exists x.x \neq y)$. This formula expresses that there are always at least two entities. It can be rewritten as

$$\phi_3 \equiv \neg(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y).$$

Its closure is

$$\begin{aligned} CL(\phi_3) = & \{ \mathsf{tt}, \phi_3, \exists x.x \neq y, \mathsf{X}(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), \mathsf{X}\neg(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), \\ & x = y, \mathsf{ff}, \neg \phi_3, \neg \exists x.x \neq y, \neg \mathsf{X}(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), \\ & \neg \mathsf{X}\neg(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), x \neq y \}. \end{aligned}$$

We check whether ϕ_3 is \mathcal{H}_2 -satisfiable (\mathcal{H}_2 is depicted in Figure 4.17). Similarly to Example 4.5.10, the set of atoms is

$$\begin{aligned} A_{\mathcal{H}_2}(\phi_3) = & \{ (q_1, D_8, 0), (q_1, D'_8, 0), (q_1, D_9, 1), \\ & (q_1, D'_9, 1), (q_1, D_{10}, 2), (q_1, D'_{10}, 2) \}. \end{aligned}$$

We just give the final resulting sets of valuation since the computation follows the same pattern as the previous examples¹⁷. Sets D_9 and D'_9 contain also valuations with black number 1.

In particular note that in D_9 and D'_9 there are no triples of the form $(\neg \exists x.x \neq y, \Theta)$ for any Θ . Finally, we have $D_{10} = D_9 \cup \{v_{41}\}$ and $D'_{10} = D'_9 \cup \{v_{41}\}$.

The graph $G_{\mathcal{H}_2}(\phi_3)$ is depicted in Figure 4.21. The valuation

$$(\neg(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), \{y\} \mapsto e_1)$$

belongs neither to D_8 nor D'_8 , therefore the formula $\phi_3 \equiv \mathsf{G}(\exists x.x \neq y)$ is not \mathcal{H}_2 -satisfiable. This is in fact correct since in the first state there is only one entity. However, it is interesting to note that $(\mathsf{X}\neg(\mathsf{tt} \mathsf{U} \neg \exists x.x \neq y), \{y\} \mapsto e_1) \in D'_8$, therefore the path $\pi = (q_1, D'_8, 0)\lambda_1(q_1, D'_9, 1)(\lambda_0(q_1, D'_{10}, 2))^\omega$ is fulfilling for the formula $\mathsf{X}\phi_3 \equiv \mathsf{XG}(\exists x.x \neq y)$. Hence, $\mathsf{X}\phi_3$ is \mathcal{H}_2 -satisfiable. Again this is correct since in the second state there is already an imploded entity in the black hole. Although for this example it is easy to see that π fulfils $\mathsf{X}\phi_3$, Proposition 4.5.21 can also be used. In particular, $(q_1, D'_8, 0)$ is the initial atom containing the correct valuation of $\mathsf{X}\phi_3$. Furthermore, $(q_1, D'_8, 0)\lambda_1(q_1, D'_9, 1)$ is a prefix and $(q_1, D'_{10}, 2)\lambda_1(q_1, D'_{10}, 2)$ is a self-fulfilling SCS (note that in D'_{10} there are no valuations for U -formulae but only valuations for negations of U -formulae). Finally, the intersection between the SCS and the set $\mathcal{F}_{\mathcal{H}_2}$ is not empty since q_1 is the only accept state of \mathcal{H}_2 . \square

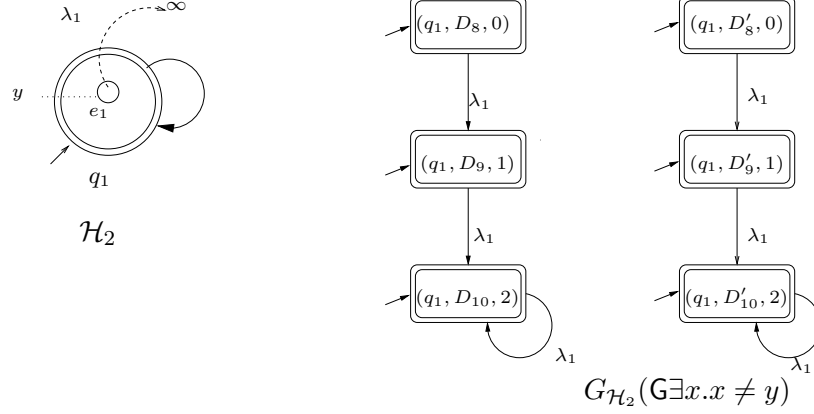
¹⁷Here, for the sake of brevity, in a valuation (ψ, Ξ, Θ) we skip the second component Ξ since it corresponds to $\text{dom}(\Theta)$.

D_8	D'_8
$v_0, v_3, v_5, v_6, v_8,$	v_0, v_3, v_5, v_6, v_8
$(\exists x.x \neq y, \emptyset)$	$(\exists x.x \neq y, \emptyset)$
$(\neg(\exists x.x \neq y), \{y\} \mapsto e_1)$	$(\neg(\exists x.x \neq y), \{y\} \mapsto e_1)$
$(X(\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(\neg X(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$(X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$(\neg X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$
$(\neg X\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(X\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$(\neg X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$(X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$
$((\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$((\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$((\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$

Table 4.12: Set of valuations D_8 and D'_8 .

D_9	D'_9
$D_8 \setminus \{(\neg(\exists x.x \neq y), \{y\} \mapsto e_1)\}$	$D'_8 \setminus \{(\neg(\exists x.x \neq y), \{y\} \mapsto e_1)\}$
$v_{30}, v_{31}, v_{32}, v_{33}, v_{34}$	$v_{30}, v_{31}, v_{32}, v_{33}, v_{34}$
$(\exists x.x \neq y, \emptyset)$	$(\exists x.x \neq y, \emptyset)$
$(\exists x.x \neq y, \{y\} \mapsto e_1)$	$(\exists x.x \neq y, \{y\} \mapsto e_1)$
$(\exists x.x \neq y, \{y\} \mapsto \infty)$	$(\exists x.x \neq y, \{y\} \mapsto \infty)$
$(X(\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(\neg X(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$(X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$(\neg X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$
$(X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$	$(\neg X(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$
$(\neg X\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(X\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$(\neg X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$(X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$
$(\neg X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$	$(X\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$
$((\text{tt U } \neg\exists x.x \neq y), \emptyset)$	$(\neg(\text{tt U } \neg\exists x.x \neq y), \emptyset)$
$((\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$	$(\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto e_1)$
$((\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$	$(\neg(\text{tt U } \neg\exists x.x \neq y), \{y\} \mapsto \infty)$

Table 4.13: Set of valuations D_9 and D'_9 .

Figure 4.21: HABA \mathcal{H}_2 and corresponding graph $G_{\mathcal{H}_2}(\mathbf{G}\exists x.x \neq y)$.

4.5.4 Complexity

We discuss briefly and in a rather informal manner on the complexity of the algorithm described in Section 4.5. The aim is to find an upper bound to the worst-case time complexity of model-checking. First we investigate the construction of the tableau graph. This involves as a preliminary step the duplication of the HABA. Then we give an upper bound on the number of steps needed to decide if a SCS is self-fulfilling.

Duplication For a HABA \mathcal{H} , let

$$T_{\max} = \max \{ |\{(q, \lambda, q') \mid (q, \lambda, q') \in \rightarrow_{\mathcal{H}}\}| \mid q, q' \in Q_{\mathcal{H}} \},$$

i.e., T_{\max} is the largest number of transitions between two states of \mathcal{H} . The number of states of the duplication \mathcal{H}_{δ} is bounded by:

$$|Q_{\mathcal{H}_{\delta}}| \leq |Q_{\mathcal{H}}|^2 \cdot T_{\max} \quad (4.15)$$

as in the worst case for a given state $q \in Q_{\mathcal{H}}$ we have to create a new state $q' \in Q_{\mathcal{H}_{\delta}}$ for every incoming transitions of q .

The number of transitions we have to add to construct \mathcal{H}_{δ} is linear to $|Q_{\mathcal{H}_{\delta}}|$ because one new transition is enough for every new state.

Note that the largest number of transitions between two states does not increase after the duplication. Therefore when needed we will use T_{\max} of \mathcal{H} .

Graph construction. For a given $\mathcal{A}\ell\text{TL}$ -formula ϕ , note that $O(|CL(\phi)|) = O(|\phi|)$ as well as $O(K(\phi)) = O(|\phi|)$. The construction of the graph is done on the duplication of \mathcal{H} . We have:

$$|A_{\mathcal{H}}(\phi)| \leq 2^{|\phi|} \cdot |Q_{\mathcal{H}_{\delta}}| \cdot |\phi| \quad (4.16)$$

because, for every state $q \in Q_{\mathcal{H}_\delta}$ we have $K(\phi)$ atoms if $\lfloor q \rfloor$ and each atom must be duplicated for every formula of the type $X\psi \in CL(\phi)$.

The complexity of the construction of a single atom (q, D, k) is dictated by the size of the set of valuations D . It is bounded by the following:

$$|D| \leq |\phi| \cdot B_{|\phi|} \cdot \frac{(|E_q| + 1)!}{(|E_q| + 1 - |\phi|)!} \quad (4.17)$$

where $B_{|\phi|}$ is the $|\phi|$ -th Bell number. This formula can be explained as follows: for a $\psi \in CL(\phi)$ we have a valuation for every partial partition of $fv(\psi)$. There are $B_{fv(\psi)}$ of these valuations. We can substitute $B_{fv(\psi)}$ by $B_{|\phi|}$ because $B_{|\phi|} \leq B_{fv(\psi)}$. For each of these partitions we need to consider every injection to E_q (last term of (4.17)). We add 1 in order to consider the black hole. Finally, since in D there are valuations for each $\psi \in CL(\phi)$, we multiply by the first term $|\phi|$.

Let $E_{\max} = \max\{|E_q| \mid q \in Q_{\mathcal{H}}\}$, i.e., E_{\max} is the cardinality of the largest set of entities in any state of \mathcal{H} (and therefore of \mathcal{H}_δ). Furthermore, let us denote with D_{\max} the cardinality of the largest set of valuations in $A_{\mathcal{H}}(\phi)$, i.e., $D_{\max} = \max\{|D| \mid (q, D, k) \in A_{\mathcal{H}}(\phi)\}$. Since $\frac{(|E_q| + 1)!}{(|E_q| + 1 - |\phi|)!} \leq (|E_q| + 1)^{|\phi|}$, then by (4.17) we obtain:

$$D_{\max} \leq |\phi| \cdot B_{|\phi|} \cdot (E_{\max} + 1)^{|\phi|}. \quad (4.18)$$

Thus, we conclude that the complexity for the construction of a single atom is of the order $O(|\phi| \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|})$. The number of edges T_G in $G_{\mathcal{H}}(\phi)$ is bounded by¹⁸

$$T_G \leq |A_{\mathcal{H}}(\phi)|^2 \cdot T_{\max}. \quad (4.19)$$

Finally, the cost of the construction of $G_{\mathcal{H}}(\phi)$ is given by summing the complexity of building $A_{\mathcal{H}}(\phi)$ and the set of edges respectively. Using (4.15), (4.16), (4.18) and (4.19) we have:

$$\begin{aligned} & O(|A_{\mathcal{H}}(\phi)| \cdot D_{\max} + |A_{\mathcal{H}}(\phi)|^2 \cdot T_{\max}) \\ &= O(|A_{\mathcal{H}}(\phi)| \cdot |\phi| \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|} + |A_{\mathcal{H}}(\phi)|^2 \cdot T_{\max}) \\ &= O(2^{|\phi|} \cdot |Q_{\mathcal{H}_\delta}| \cdot |\phi|^2 \cdot (B_{|\phi|} \cdot E_{\max}^{|\phi|} + 2^{|\phi|} \cdot |Q_{\mathcal{H}_\delta}| \cdot T_{\max})) \\ &= O(2^{|\phi|} \cdot |Q_{\mathcal{H}}|^2 \cdot T_{\max} \cdot |\phi|^2 \cdot (B_{|\phi|} \cdot E_{\max}^{|\phi|} + 2^{|\phi|} \cdot |Q_{\mathcal{H}}|^2 \cdot T_{\max}^2)). \end{aligned}$$

Thus, constructing the graph is:

¹⁸In principle, T_{\max} is of the order $O(E_{\max}^{E_{\max}})$ as we can have every possible injective partial mapping from a set of cardinality E_{\max} to another set of cardinality E_{\max} . However, it is reasonable to consider that for big E_{\max} such a bound becomes unrealistic. It is hard to imagine such a HABA with so many transitions between two states. For example, the symbolic semantics presented in Section 4.4 is deterministic. T_{\max} is equal to the number of parallel components of the program and does not depend on E_{\max} . By this consideration, for a more sensible general picture of the model checking algorithm computational cost, it seems reasonable to keep T_{\max} itself in the complexity measure.

- $O(|Q_{\mathcal{H}}|^4)$, i.e., polynomial in the number of states of \mathcal{H} ;
- $O(2^{|\phi|} \cdot |\phi|^2 \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|})$, i.e., super-exponential on the size of ϕ ;
- $O(E_{\max}^{|\phi|})$, i.e., polynomial in the largest number of entities in \mathcal{H} .

Deciding if a SCS is self-fulfilling. To decide whether a given strongly connected component G is self-fulfilling, we propose an algorithm consisting of the following steps:

- Construct a linking structure with nodes $(A, \psi_1 \cup \psi_2, \Theta)$ where $A = (g, D, k)$ is an atom in G and $(\psi_1 \cup \psi_2, \Xi, \Theta) \in D$, and edges

$$(A', \psi_1 \cup \psi_2, \Theta') \rightarrow (A, \psi_1 \cup \psi_2, \Theta)$$

for all $A \rightarrow_{\lambda} A'$ with $\Theta' = \lambda \circ \Theta$. (Note that the links in this newly created structure go in the reverse direction w.r.t. the edges of the tableau graph.)

- Appoint in this structure all nodes $(A, \psi_1 \cup \psi_2, \Theta)$ as *initial* for which $(\psi_2, \Xi \upharpoonright \psi_2, \Theta \upharpoonright \psi_2) \in D_A$.
- Perform a reachability analysis on the structure thus obtained.

We claim that G is self-fulfilling iff all nodes of this linking structure are reachable. For if all nodes are reachable then for all $A \in G$ and all $(\psi_1 \cup \psi_2, \Xi, \Theta) \in D_A$ there is a path

$$(A_n, \psi_1 \cup \psi_2, \Theta_n) \rightarrow \cdots \rightarrow (A_0, \psi_1 \cup \psi_2, \Theta_0)$$

with $A_0 = A$, $\Theta_0 = \Theta$, $A_0 \rightarrow_{\lambda_0} \cdots \rightarrow_{\lambda_{n-1}} A_n$ and $(\psi_2, \Xi \upharpoonright \psi_2, \lambda_{n-1} \circ \cdots \circ \lambda_0 \circ (\Theta_0 \upharpoonright \psi_2)) \in D_{A_0}$; hence G is self-fulfilling. Vice versa, if for all $A \in G$ and all $(\psi_1 \cup \psi_2, \Xi, \Theta) \in D_A$ there is a node $B \in G$ such that

- $A = A_0 \rightarrow_{\lambda_0} \cdots \rightarrow_{\lambda_{n-1}} A_n = B$
- $(\psi_2, \Xi \upharpoonright \psi_2, \lambda_{n-1} \circ \cdots \circ \lambda_0 \circ (\Theta \upharpoonright \psi_2)) \in D_B$

then also $(A_n, \psi_1 \cup \psi_2, \Theta_n) \rightarrow \cdots \rightarrow (A_0, \psi_1 \cup \psi_2, \Theta_0)$ with $\Theta_i = \lambda_{i-1} \circ \cdots \circ \lambda_0 \circ \Theta$ is a path through the linking structure, and $(A_n, \psi_1 \cup \psi_2, \Theta_n)$ is an initial node in that structure.

The cost of this analysis equals the cost of building the linking structure plus the cost of the reachability analysis, which is linear in the size of that linking structure. The size of this structure is dictated by the number of edges, which equals the number of edges in G times the number of “U-valuations” in each atom of G — which in turn is linear to D_{\max} . Thus we obtain the following worst case cost for establishing whether a given SCS G is self-fulfilling:

$$O(T_G \cdot D_{\max})$$

where T_G is the number of edges in G .

The number of SCS in $G_{\mathcal{H}}(\phi)$ is of the order $O(2^{|A_{\mathcal{H}}(\phi)}| \cdot T_{\max})$. Hence, the worst time complexity for checking if there exists a self-fulfilling SCS is:

$$\begin{aligned} & O(2^{|A_{\mathcal{H}}(\phi)}| \cdot T_{\max} \cdot T_G \cdot D_{\max}) \\ &= O(2^{|Q_{\mathcal{H}_\delta}| \cdot |\phi| \cdot 2^{|\phi|}} \cdot T_{\max}^2 \cdot 2^{2|\phi|} \cdot |Q_{\mathcal{H}_\delta}|^2 \cdot |\phi|^3 \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|}) \\ &= O(2^{(2^{|\phi|} \cdot |Q_{\mathcal{H}_\delta}| + 2)|\phi|} \cdot T_{\max}^2 \cdot |Q_{\mathcal{H}_\delta}|^2 \cdot |\phi|^3 \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|}) \\ &= O(2^{(2^{|\phi|} \cdot |Q_{\mathcal{H}}|^2 \cdot T_{\max} + 2)|\phi|} \cdot T_{\max}^4 \cdot |Q_{\mathcal{H}}|^4 \cdot |\phi|^3 \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|}) \end{aligned}$$

If we single out the different parameters we have:

- $O(2^{2^{|\phi|}|\phi|} \cdot |\phi|^3 \cdot B_{|\phi|} \cdot E_{\max}^{|\phi|})$ in the size of the formula ϕ ;
- $O(2^{|Q_{\mathcal{H}}|^2} \cdot |Q_{\mathcal{H}}|^4)$ in the size of the model \mathcal{H} ;
- $O(E_{\max}^{|\phi|})$, i.e., polynomial in the largest number of entities in \mathcal{H} .

Discussion. Clearly, here the presence of entities in the states and the valuations that must be taken into account produces a rather expensive overhead with respect to the tableau algorithm for LTL presented in [77] (and summarised in Section 2.1.6) that is only exponential in $|\phi|$. However, as remarked in [77] most interesting properties about safety and liveness can be formulated by small size formulae.

Concerning the parameter E_{\max} , in order to have an idea how it may grow, we can consider the programming language \mathcal{L} of Section 4.4. There E_{\max} is bounded by the number of variables in the program.

Although, the bound given here is rather rough, more problematic is the complexity with respect to size of the model since in principle nothing can be assumed for the cardinality of $Q_{\mathcal{H}}$. The LTL algorithm is linear in the size of the model (instead of exponential) because it uses *maximal* strong connected components (MSCC) instead of SCS (that are an exponential number). It is very likely that the same optimisation would provide similar benefits in the algorithm defined in this section. Nevertheless, in this thesis the emphasis is in the decidability result while optimisations of the algorithm is postponed to future work.

4.6 Related work

History-dependent automata. History-dependent (HD) automata [83, 84, 90] are the main inspiration for HABAs. An HD-automaton is an automaton where states, transitions and labels are equipped with a set of local names that can be created dynamically¹⁹. After the definition of HABAs in this chapter it is interesting to briefly relate the two formalisms. Reallocation of entities in

¹⁹For a summary on HD-automata in this thesis see Section 2.2.

HABAs resembles the reallocation of names in HD-automata. HD-automata and HABAs differ in the way in which entities are referred to. More precisely, in a HABA, entities can only be addressed by means of logical variables that can be compared by $\mathcal{A}llTL$ -formulae. More important, though, is the novelty introduced in HABAs by the black hole abstraction. This key feature allows us to deal with a possibly unbounded number of entities. Another difference concerns the accepted language. HABAs are automata on infinite words (because of the generalised Büchi acceptance condition), whereas HD-automata are defined on finite words.

Spatial logic. Related to $\mathcal{A}llTL$, concerning properties of freshness, is the recent Spatial Logic (SL) [21, 17]. SL is defined for the Ambient Calculus [19] and has modalities that refer to space as well as time. Freshness can be identified in SL using a special quantifier, and has a somewhat different interpretation than in $\mathcal{A}llTL$. More precisely, in SL “fresh” means distinct from any name used in formula and in the model satisfying it. If there is a fresh name, there are infinitely many of them (Gabbay-Pitts property [50]). In contrast, in $\mathcal{A}llTL$, if an entity is fresh it means that the entity is used in the current state and did not appear previously. This conceptual difference has several consequences. For instance, there exist non-contradictory $\mathcal{A}llTL$ -formulae where more than one distinct fresh entity is identified in the same state. Another difference between SL and $\mathcal{A}llTL$ concerns quantification. In SL, quantification is over a fixed (countable) set of names, whereas in $\mathcal{A}llTL$, quantification ranges over entities that are alive in the current state. This set is not fixed from state to state. Therefore, e.g., the $\mathcal{A}llTL$ formula $\forall x.X\phi$ is not equivalent to $X\forall x.\phi$ (cf. Proposition 4.2.5).

Tableau-based methods. As we have seen in Chapter 2, there are basically two approaches to model-checking temporal logics: the automata-theoretic approach (for LTL [109] and CTL [47, 73]) and the tableau method. Tableaux are typically used for the solution of more general problems, like satisfiability. For model checking, the tableau approach was first developed for CTL [6, 26]. Our algorithm is strongly based on the tableau method for LTL presented in [77].

Model checking and logics for object-oriented systems. Model checking tools for object-oriented systems are becoming more and more popular. However — in the approaches we are aware of — such as Bandera [32], Java PathFinder [58] and others (see Section 1.4 for a global overview of software model checkers) dynamic creation of objects is only supported to a limited extent: the number of created objects must be bounded a priori. Moreover, also the property specification formalisms are not tailored towards dynamic aspects of objects (such as allocation and de-allocation).

The paper [112] deals with unbounded number of Java objects and threads. The approach is mostly based on abstract interpretation and 3-valued logic.

The main idea is to conservatively represent (via 3-valued logical structures) many configurations using a single abstract configuration. This clearly relates with the black hole abstraction. However, the paper deals only with *safety* properties and in particular, the emphasis is on the interference between Java threads, deadlocks, shared abstract data types, and illegal thread interactions. Furthermore, this technique may *falsely* report that a safety property may be violated, although it can never miss a violation. The recent paper [113] (again based on 3-valued logic) presents a formalism, called *Evolution Temporal Logic (ETL)*, that in many aspects is rather close to $\mathcal{A}\ell\ell\text{TL}$. As $\mathcal{A}\ell\ell\text{TL}$, ETL permits to express properties concerning the allocation and deallocation of objects and threads. It uses models with reallocations corresponding to those employed by HD-automata and by the version of HABAs defined in this chapter. An abstract-interpretation algorithm, again, sound but not complete, for verifying the properties is also proposed.

5

Dynamic References

5.1 Introduction

Pointers (references) are a powerful programming mechanism. They are very flexible but at the same time error prone due to the so-called *complexity of pointer swing* [67, 94] resulting from aliasing: apparently unrelated expressions may be altered by the assignment to an entity in memory referred to by more than one pointer. It is difficult to control and to reason about this phenomenon. Run-time safety violations (e.g., dereferencing null or disposed pointers) easily arise and their detection cannot be ensured by type systems. Although the advent of object-oriented languages has limited the use of pointers (at least at the programming level), the possibility for an object *to refer to* other objects is still present in any practical object-oriented model describing interactions among objects. For example, proper concepts of object-orientation such as *object composition* reduces to references between objects. Thus, for reasoning about object-oriented systems it is indispensable to be able to reason about the way objects refer to each other.

In order to capture essential information of systems dealing with references, we introduce a logic, called $\mathcal{N}allTL$, whose primitives address dynamic allocation and deallocation of entities and their dynamic pointers. It should not be surprising that $\mathcal{N}allTL$ is both a subset of BOTL as well as an extension (to pointers) of $\mathcal{A}llTL$. Typical properties expressible in $\mathcal{N}allTL$ are, for example:

- in object-oriented systems:
 - *every object reachable from a particular object will be eventually deallocated.*

– objects o_1 and o_2 belong to disjoint lists.

- in security: no untrusted agent will have a reference to a secret datum.

Along the line of the model checking algorithm defined for $\mathcal{All}TL$ (cf. Section 4.5), in this chapter, we define another algorithm that checks whether a $\mathcal{Nall}TL$ formula is *not satisfiable* in a given model. Also this algorithm is based on the construction of a tableau graph [77], however in this particular case due to a more sophisticated abstraction which allows to describe rather expressive models, the algorithm may produce false counterexamples.

The models we propose in this chapter are an enhanced version of High-level Allocational Büchi Automata (HABA) defined in Chapter 4 that in turn are based on History Dependent Automata [84]. Besides the ability to describe the dynamic allocation and deallocation of entities typical of HABA (and HD-automata), the extension we discuss here deals with *references* (pointers) between entities that are alive in the same state. From state to state, references can be created, removed and modified: i.e., the model describes dynamic topological structures of alive entities.

In order to achieve finite-state models, we propose an abstraction that is able to deal with some kind of unbounded systems. In HABAs we use a special class of entities that model finite but *unbounded* chains of “concrete” entities. Our unbounded entities are somehow a specialisation (to chains) of what in the literature is known as *summary node* [100]. Unbounded entities provide HABAs with the capacity to describe, by a single state, an infinite number of topological structures that can be represented at different *level of abstraction*. A precise characterisation of the relation between the different levels will result in the notion of *morphism*.

As we have done in Chapter 4, we show that HABAs with references can be used as a mathematical model for the definition of the semantics of a simple language with basic object-based notions such as object creation, and navigation. For this language we define two semantics: a concrete one, that is infinite state and a symbolic one that is finite state. The relation between the two semantics is then studied.

This chapter is organised as follows: in Section 5.2 the syntax and semantics of $\mathcal{Nall}TL$ is defined; in Section 5.3 we introduce ABA and HABA with references as well as the notion of morphism and an enhanced version of reallocation. Then, in Section 5.4 we study how to relate ABA and HABA with references. The programming language is defined in Section 5.5 and its operational semantics in Section 5.6. The model checking algorithm is given in Section 5.7. Finally, we conclude the chapter with an overview of other techniques used for the analysis of pointer structures (Section 5.8). Proofs are reported in Appendix B.

5.2 A logic for navigation

In this section, we will enhance $\mathcal{A}llTL$, such that properties about *navigation* can be expressed. Navigations are essentially possible when entities reference each other. In this thesis, we restrict ourselves to the case where every entity e has precisely one outgoing reference¹. This reference is defined for entity e , if e has a pointer (denoted by $.a$), otherwise it is undefined. Let $LVAR$ be a (countable) set of logical variables.

Definition 5.2.1. Let $x \in LVAR$. The syntax of $\mathcal{N}allTL$ is defined by the following grammar:

$$\begin{aligned} (\alpha \in)Nav & ::= nil \mid x \mid \alpha.a \\ \phi & ::= \alpha = \alpha \mid \alpha \text{ new} \mid \alpha \rightsquigarrow \alpha \mid \exists x:\phi \mid \neg\phi \mid \phi \vee \phi \mid X\phi \mid \phi \cup \phi. \end{aligned}$$

We refer to elements of the set Nav as *navigation expressions*. For instance, the expression $x.a$ denotes the entity referred to by the entity denoted by x (if any). Similarly, $x.a.a$ denotes the entity referred to by $x.a$. The suffix of a navigation expression can be arbitrarily long, therefore we can specify lists of unbounded length. We write $x.a^n$ ($n \in \mathbb{N}$) as a shorthand for x followed by n successive pointers, that is formally:

$$\begin{aligned} x.a^0 & \equiv x \\ x.a^{n+1} & \equiv (x.a^n).a \end{aligned}$$

As in $\mathcal{A}llTL$, formula $\alpha \text{ new}$ holds if the entity denoted by α is fresh in the current state, i.e., the entity denoted by α did not exist in the previous state of the computation. Formula $\alpha_1 = \alpha_2$ holds if expressions α_1 and α_2 are aliases (i.e., they denote the same entity) in the current state. The predicate $\alpha_1 \rightsquigarrow \alpha_2$ (read α_1 *reaches* α_2 or α_1 *leads-to* α_2) means that from the entity denoted by α_1 there exists a reference path reaching the entity denoted by α_2 (i.e., $\alpha_1.a^k = \alpha_2$ for some $k \geq 0$). As known in the literature, reachability is not a first-order graph property [34] therefore the operator \rightsquigarrow provide $\mathcal{N}allTL$ with some second order capabilities.

Besides standard LTL abbreviations like G, F, and the others described in Table 4.1 for $\mathcal{A}llTL$, throughout this chapter we use the typical $\mathcal{N}allTL$ abbreviations reporter in Table 5.1.

5.2.1 Semantics

Let Ent be a countable universe of entities ranged over by e, e', e_1, \dots . $\mathcal{N}allTL$ formulae are interpreted over infinite sequences of sets of entities (as $\mathcal{A}llTL$) and mappings defining information about mutual references of entities.

¹For the purpose of the definition of the logic this restriction to a single outgoing reference is not essential. It is only used to increase the precision in the abstraction that we will define later. Lifting the definitions given here to the general case is not difficult (in that case we would obtain a formalism very close to BOTL. See 3).

α dead	for	$\alpha = nil$
α alive	for	$\neg(\alpha \text{ dead})$
$\alpha_1 \not\rightsquigarrow \alpha_2$	for	$\neg(\alpha_1 \rightsquigarrow \alpha_2)$

Table 5.1: Typical $\mathcal{N}\ell\ell\text{TL}$ abbreviations.

Let \perp be a special value such that $\perp \notin Ent$. \perp is used to represent *undefined*. For a set E we write $E^\perp = E \cup \{\perp\}$.

Definition 5.2.2. An *allocation sequence* σ is an infinite sequence of pairs $(E_0, \mu_0)(E_1, \mu_1)(E_2, \mu_2) \cdots$ where for $i \in \mathbb{N}$:

- $E_i \subseteq Ent$
- $\mu_i : E_i^\perp \rightarrow E_i^\perp$ such that $\mu_i(\perp) = \perp$.

Function μ_i gives the reference of the entity pointed to by its argument. Each μ_i is strict.

Let $\theta : \text{LVAR} \rightarrow Ent^\perp$ be a valuation of the logical variables. The semantics of the navigation expression α is given by $\llbracket \cdot \rrbracket : Nav \times (Ent^\perp \rightarrow Ent^\perp) \times (\text{LVAR} \rightarrow Ent^\perp) \rightarrow Ent$ defined as:

$$\begin{aligned} \llbracket nil \rrbracket_{\mu, \theta} &= \perp \\ \llbracket x \rrbracket_{\mu, \theta} &= \theta(x) \\ \llbracket \alpha.a \rrbracket_{\mu, \theta} &= \mu(\llbracket \alpha \rrbracket_{\mu, \theta}). \end{aligned}$$

Let $\sigma^i = (E_i, \mu_i)(E_{i+1}, \mu_{i+1}) \cdots$. For a given allocation sequence σ , let E_i^σ and μ_i^σ denote the first and the second component in the i -th state of σ , respectively. The semantics of $\mathcal{N}\ell\ell\text{TL}$ is defined in terms of the satisfaction relation $\sigma, N, \theta \models \phi$, where σ is an allocation sequence, $N \subseteq E_0^\sigma$ is the set of entities initially new and θ is a valuation of the free logical variables of the formula ϕ .

Let $N_i^\sigma \subseteq Ent$ (i.e., $\perp \notin N_i^\sigma$) denote the set of new entities in state i , defined as $N_0^\sigma = N$ and $N_{i+1}^\sigma = E_{i+1}^\sigma \setminus E_i^\sigma$. Let $\theta_i^\sigma : \text{LVAR} \rightarrow Ent^\perp$ be a valuation of the logical free variables in state i (of σ) where:

$$\theta_i^\sigma(x) = \begin{cases} \theta(x) & \text{if } \forall k \leq i : \theta(x) \in E_k^\sigma \\ \perp & \text{otherwise.} \end{cases}$$

Note that once a logical variable is mapped to an entity, then this association remains along σ unless the entity dies, i.e., is deallocated.

The expressions $x.a.a$ and $y.a$ eventually will become aliases	$F(x.a.a = y.a)$
If $x_1.a^2$ and $x_2.a$ are aliases, the entity associated to y is deallocated before $x_1.a^2$ and $x_2.a$ are not aliases anymore	$G(x_1.a^2 = x_2.a \wedge y \text{ alive} \Rightarrow (x_1.a^2 = x_2.a \text{ U } y \text{ dead}))$
Eventually, v will point to an entity in a non-empty cycle	$F(\exists x : x \neq v \wedge x \rightsquigarrow v \wedge v \rightsquigarrow x)$
Variables v and w always point to disjoint parts of the heap (<i>non-interference</i>)	$G(\forall x : v \rightsquigarrow x \Rightarrow w \not\rightsquigarrow x)$
Every entity reachable from v will be eventually deallocated	$\forall x : (v \rightsquigarrow x \Rightarrow Fx \text{ dead})$
All and only the entities currently reachable from v will be eventually deallocated	$(\forall x : v \rightsquigarrow x \Rightarrow Fx \text{ dead}) \wedge (\forall x : v \not\rightsquigarrow x \Rightarrow Gx \text{ alive})$
v 's list will be (and remains to be) eventually reversed	$\forall x : \forall y : (v \rightsquigarrow x \wedge x.a = y) \Rightarrow FG(y.a = x)$
A tautology	$x.a \text{ alive} \Rightarrow x \text{ alive}$

Table 5.2: Some example properties expressible in $\mathcal{N}allTL$.

The satisfaction relation \models for $\mathcal{N}allTL$ is defined as follows:

$$\begin{aligned}
\sigma, N, \theta \models \alpha_1 = \alpha_2 & \quad \text{iff} \quad \llbracket \alpha_1 \rrbracket_{\mu_0^\sigma, \theta} = \llbracket \alpha_2 \rrbracket_{\mu_0^\sigma, \theta} \\
\sigma, N, \theta \models \alpha \text{ new} & \quad \text{iff} \quad \llbracket \alpha \rrbracket_{\mu_0^\sigma, \theta} \in N \\
\sigma, N, \theta \models \alpha_1 \rightsquigarrow \alpha_2 & \quad \text{iff} \quad \exists k \geq 0 : (\mu_0^\sigma)^k(\llbracket \alpha_1 \rrbracket_{\mu_0^\sigma, \theta}) = \llbracket \alpha_2 \rrbracket_{\mu_0^\sigma, \theta} \\
\sigma, N, \theta \models \exists x : \phi & \quad \text{iff} \quad \exists e \in E_0^\sigma : \sigma, N, \theta\{e/x\} \models \phi \\
\sigma, N, \theta \models \neg \phi & \quad \text{iff} \quad \sigma, N, \theta \not\models \phi \\
\sigma, N, \theta \models \phi \vee \psi & \quad \text{iff} \quad \text{either } \sigma, N, \theta \models \phi \text{ or } \sigma, N, \theta \models \psi \\
\sigma, N, \theta \models X\phi & \quad \text{iff} \quad \sigma^1, N_1^\sigma, \theta_1^\sigma \models \phi \\
\sigma, N, \theta \models \phi \text{ U } \psi & \quad \text{iff} \quad \exists i : (\sigma^i, N_i^\sigma, \theta_i^\sigma \models \psi \text{ and } \forall j < i : \sigma^j, N_j^\sigma, \theta_j^\sigma \models \phi).
\end{aligned}$$

Note that the proposition $\alpha_1 \rightsquigarrow \alpha_2$. is satisfied in case $\llbracket \alpha_2 \rrbracket_{\mu_0^\sigma, \theta} = \perp$ and $\llbracket \alpha_1 \rrbracket_{\mu_0^\sigma, \theta}$ can reach an entity with an undefined pointer.

Example 5.2.3. In Table 5.2 are reported some example properties expressible in $\mathcal{N}allTL$. \square

5.3 ABA and HABA with references

Throughout this chapter, let $M \in \mathbb{N}$ be a global constant, $\mathbb{M} = \{1, \dots, M\}$ and $\mathbb{M}^* = \mathbb{M} \cup \{*\}$ where $*$ is a special distinguished symbol.

We now introduce the concept of cardinality which is the base of the abstraction mechanism we will develop (and use) throughout this chapter.

Definition 5.3.1. Let $E \subseteq Ent$. A function $\mathcal{C}_E : E \rightarrow \mathbb{M}^*$ is called *cardinality function* (on E).

A cardinality function \mathcal{C}_E associates to every entity in E a number ($\leq M$) or the special symbol $*$. $\mathcal{C}_E(e) = k \leq M$ means that the cardinality of e is precisely k . $\mathcal{C}_E(e) = *$ means that the cardinality of e is some natural number larger than M . If the set E is clear from the context we simply write \mathcal{C} for \mathcal{C}_E .

Definition 5.3.2. The *unitary* cardinality function on E is the cardinality function $\mathbf{1}_E : E \rightarrow \mathbb{M}^*$ such that $\mathbf{1}_E(e) = 1$ for all $e \in E$.

We need to define the sum on cardinalities:

$$n \oplus m = \begin{cases} n + m & \text{if } n, m \in \{0, \dots, M\} \text{ and } n + m \leq M \\ * & \text{otherwise.} \end{cases}$$

We lift the notion of cardinality to sets of entities as follows: if $E \subseteq Ent$ then

$$\mathcal{C}(E) = \sum_{e \in E} \mathcal{C}(e).$$

The notation \sum stands for a sum that uses the \oplus operator defined above instead of the standard sum on \mathbb{N} . Moreover, for $n \in \mathbb{N}$ let

$$\lceil n \rceil_M = \begin{cases} n & \text{if } n \leq M \\ * & \text{otherwise.} \end{cases}$$

The global constant M is mostly fixed and therefore if no confusion arise we will not indicate it explicitly and we write $\lceil n \rceil$ instead of $\lceil n \rceil_M$. The cardinality function imposes a distinction among entities. For the sake of exposition, in this chapter, we identify three different classes:

- *concrete entities*: those with cardinality one;
- *multiple entities*: those with cardinality neither 1 nor $*$;
- *unbounded entities*: those with cardinality $*$.

For a set of entities E , we write E^* for the subset of its unbounded entities, i.e., $E^* = \{e \in E \mid \mathcal{C}_E(e) = *\}$. *Bounded* entities are either concrete or multiple.

5.3.1 Morphisms

A *configuration* $\gamma = (E, \mu, \mathcal{C}_E)$ represents a weighted directed graph where $E \subseteq Ent$ is the set of nodes, μ defines the set of arcs, and \mathcal{C}_E represents the weights associated to the nodes. Figure 5.1 depicts three configurations as weighted graphs: γ_1 , γ_2 and γ_3 . Circles represent concrete entities. Filled circles represents multiple/unbounded entities. Numbers (or stars) associated with entities denote cardinalities.

Multiple and unbounded entities provide us with the possibility to *abstract* from specific portions of the graph (configuration) thus obtaining a more compact representation of the original graph. In this chapter, we apply the following abstraction:

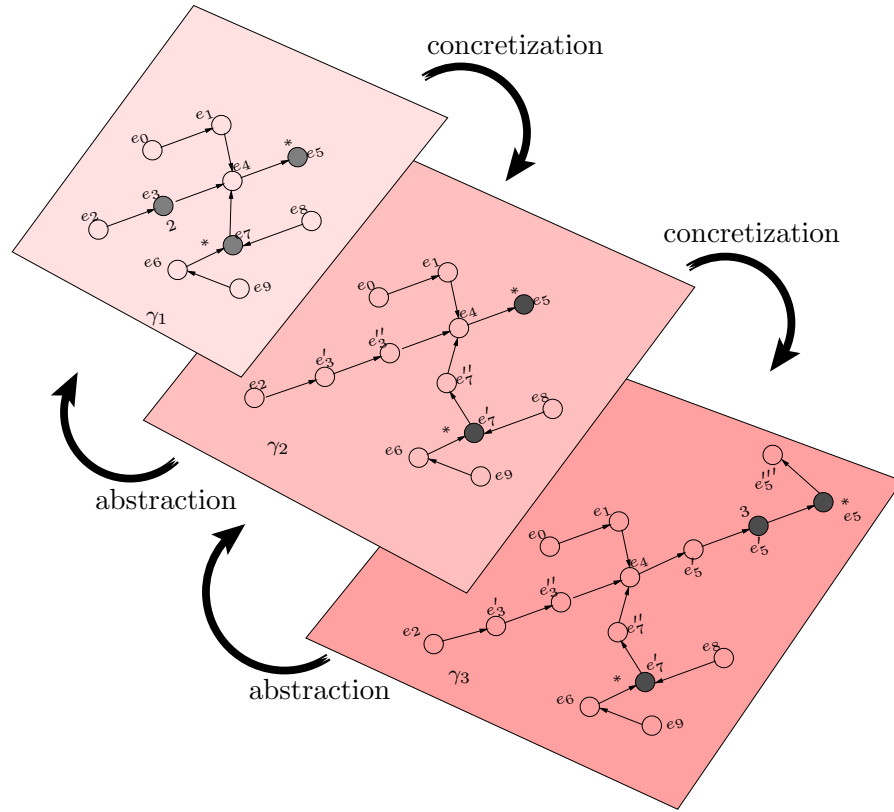


Figure 5.1: A pointer structure and some of its different levels of abstraction.

multiple and unbounded entities represent *chains* of more concrete (i.e., with lower cardinality) entities.

The length of the chain represented by a multiple entity e must be consistent with the cardinality of e . Unbounded entities represent chains of arbitrary length strictly larger than M . For example, in configuration γ_1 depicted in Figure 5.1, entities e_3 , e_5 and e_7 are not concrete. In γ_2 , e_3 has been replaced by the chain of entities e'_3 and e''_3 . Since $\mathcal{C}_{\gamma_1}(e) = 2$, this is the only allowed concretization. We say that γ_2 is a more concrete configuration than γ_1 , or symmetrically, γ_1 is more abstract than γ_2 . Moreover, the other difference between γ_1 and γ_2 is that e_7 in γ_1 is replaced by the chain composed by e'_7 and e''_7 in γ_2 . If we think of e_7 as a chain of arbitrary length (larger than M), this process corresponds to splitting this chain into a chain of arbitrary length (i.e., e'_7) followed by a concrete entity (i.e., e''_7). This phenomenon is sometimes known in the literature as *materialisation* [100]. Note that we can split e_7 in

infinitely many ways. Any of such splitting yields a graph with a different level of abstraction than the original one. The level of abstraction depends on the number of abstract/concrete entities used. In the figure, the three different abstraction levels are represented by planes of configurations. Thus, γ_2 is more concrete than γ_1 . Configuration γ_3 , where entity e_5 of γ_2 has been replaced by the chain consisting of e'_5 , e''_5 , e_5 , and e'''_5 , is more concrete than γ_2 (and therefore than γ_1).

There exists an intimate relation among γ_1 , γ_2 and γ_3 . Any of these configurations can be obtained by the other by replacing some unbounded/multiple entities by a chain of more concrete ones or vice-versa by replacing chains of entities by more abstract ones. These considerations emphasise that these configurations represent the *same pointer structure* but at *different levels of abstraction*.

In this section, we try to give a general characterisation for such a relation between configurations (i.e., same pointer structure but different abstraction level) that will result in a notion of graph abstraction that we call, adopting categorical jargon, *morphism*.

Preliminary notation. Throughout this chapter, let

$$\text{CONF} = 2^{\text{Ent}} \times (\text{Ent} \rightarrow \text{Ent}) \times (\text{Ent} \rightarrow \mathbb{M}^*) \quad (5.1)$$

be the set of all configurations ranged over by γ . In the context of a configuration $(E, \mu, \mathcal{C})^2$, it is sometimes convenient to consider instead of μ , the relation $\prec \subseteq E \times E$ induced by μ . This is defined as

$$\prec = \{(e, \mu(e)) \mid e, \mu(e) \in \text{Ent}\}.$$

Note that $(\perp, \perp), (e, \perp), (\perp, e) \notin \prec$ for any $e \in \text{Ent}$. We write $e \prec e'$ as shorthand for $(e, e') \in \prec$. Furthermore we will freely interchange (E, μ, \mathcal{C}) and (E, \prec, \mathcal{C}) .

For $e \in E$, let $\text{indegree}(e) = |\{e' \mid (e', e) \in \prec\}|$. A sequence of E 's elements e_1, \dots, e_j is a *chain* (of length j) if $e_i \prec e_{i+1}$ for $1 \leq i < j$. A set of nodes $E' \subseteq E$ with $|E'| = j \geq 1$ defines a chain of length j if there exists a bijection $f : \{1, \dots, j\} \rightarrow E'$ such that $f(1), \dots, f(j)$ is a chain. In this case, we define $\text{first}(E') = f(1)$ and $\text{last}(E') = f(j)$. E' defines a *pure* chain if $\forall e' \in \{f(2), \dots, f(j)\} : \text{indegree}(e') = 1$ and f is unique. It follows from this definition that chains consisting of only one element are pure. Moreover, let E' be a chain such that $E' \subseteq E^\perp$. Then, by the definition of \prec , if $\perp \in E'$ then $E' = \{\perp\}$. In the following, let \preceq be the reflexive closure of \prec , i.e., $\preceq = \prec \cup \{(e, e) \mid e \in E\}$.

The next definition defines our notion of graph transformation.

²From now on, for a triple (E, μ, \mathcal{C}) it is clear that the domain of \mathcal{C} is E .

Definition 5.3.3. (morphism) Let $\gamma_1 = (E_1, \prec_1, \mathcal{C}_1)$, $\gamma_2 = (E_2, \prec_2, \mathcal{C}_2)$ be two configurations. A *morphism* h from γ_1 to γ_2 is a surjective function $h : E_1 \rightarrow E_2$ such that:

- 1m. $\forall e \in E_2 : h^{-1}(e)$ is a pure chain;
- 2m. $\forall e, e' \in E_2 : e \prec_2 e' \Rightarrow \text{last}(h^{-1}(e)) \prec_1 \text{first}(h^{-1}(e'))$;
- 3m. $\forall e, e' \in E_1 : e \prec_1 e' \Rightarrow h(e) \preceq_2 h(e')$;
- 4m. $\forall e \in E_2 : \mathcal{C}_2(e) = \mathcal{C}_1(h^{-1}(e))$;

We write $h : \gamma_1 \xrightarrow{\quad} \gamma_2$ or $\gamma_1 \xrightarrow{h} \gamma_2$ if h is a morphism from γ_1 to γ_2 .

Condition 1m allows to abstract only pure chains of entities by a single entity. A single element is a chain of length 1. We require a chain to be pure in order to maintain the branching topology of γ_1 . Conditions 2m and 3m preserve the dependency of the entities when going from γ_1 to γ_2 . Condition 4m requires that the sum of the cardinalities of entities mapped onto the same element e must be equal to the cardinality of e .

The existence of a morphism between two configurations ensures the correspondence of the abstract shape of the graphs represented by the configurations. The correspondence is up to a certain degree of abstractness given by the morphism itself.

Example 5.3.4. For the configurations in Figure 5.1, there exists a morphism $h_{21} : \gamma_2 \xrightarrow{\quad} \gamma_1$ defined as: $h_{21} \upharpoonright \{e_0, e_1, e_2, e_4, e_5, e_6, e_8, e_9\} = id$ and

$$\begin{aligned} h_{21}(e'_3) &= e_3 & h_{21}(e''_3) &= e_3 \\ h_{21}(e'_7) &= e_7 & h_{21}(e''_7) &= e_7. \end{aligned}$$

Moreover, there exists a morphism $h_{32} : \gamma_3 \xrightarrow{\quad} \gamma_2$ defined as:

$$h_{32} \upharpoonright \{e_0, e_1, e_2, e'_3, e''_3, e_4, e_6, e'_7, e''_7, e_8, e_9\} = id$$

and

$$h_{32}(e_5) = e_5 \quad h_{32}(e'_5) = e_5 \quad h_{32}(e''_5) = e_5 \quad h_{32}(e'''_5) = e_5.$$

□

Proposition 5.3.5. For a configuration γ there exists a morphism, called *identity* morphism, $id_\gamma : \gamma \xrightarrow{\quad} \gamma$ defined by $id_\gamma(e) = e$ for all $e \in E_\gamma$.

Proof. It is straightforward to verify that id_γ satisfies all the conditions of Def. 5.3.3. □

Definition 5.3.6. Given two morphisms $h : \gamma_1 \xrightarrow{\quad} \gamma_2$ and $h' : \gamma_2 \xrightarrow{\quad} \gamma_3$, the *composition* $h' \circ h : \gamma_1 \xrightarrow{\quad} \gamma_3$ is defined by $(h' \circ h)(e) = h'(h(e))$ for all $e \in E_{\gamma_1}$.

The composition of two morphisms is also a morphism as stated by the following:

Proposition 5.3.7. If $h : \gamma \succ \rightarrow \gamma'$ and $h' : \gamma' \succ \rightarrow \gamma''$ then $h' \circ h : \gamma \succ \rightarrow \gamma''$.

Proof. It can be verified that $h' \circ h$ satisfies all the conditions of Def. 5.3.3. \square

Propositions 5.3.5 and 5.3.7 imply that the set of configurations equipped with morphisms forms a category.

Example 5.3.8. We can define the morphism $h_{31} : \gamma_3 \succ \rightarrow \gamma_1$ using the morphisms in Example 5.3.4 as $h_{31} = h_{21} \circ h_{32}$. \square

Definition 5.3.9. Morphism $h : \gamma_1 \succ \rightarrow \gamma_2$ is an *isomorphism* if and only if there exists a morphism $h' : \gamma_2 \succ \rightarrow \gamma_1$ such that $h' \circ h = id_{\gamma_1}$. In this case, h' is the inverse of h and h is the inverse of h' .

If there exists an isomorphism between configurations γ_1 and γ_2 we say that they are isomorphic and write $\gamma_1 \cong \gamma_2$.

5.3.2 Allocational Büchi Automata

Allocational Büchi Automata are basically generalised Büchi automata where to each state a set of entities and a relation between entities are associated. These entities, in turn, serve as valuations of logical variables and the references as valuations for the navigation expressions.

Definition 5.3.10. An *Allocational Büchi Automaton* (ABA) \mathcal{A} is a tuple $\langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$, with

- $X \subseteq \text{LVAR}$ a finite set of logical variables;
- Q a (possibly infinite) set of states;
- $E : Q \rightarrow \text{CONF}$ a function yielding for each state q a configuration $\gamma_q = (E_q, \mu_q, \mathbf{1}_{E_q})$.
- $\rightarrow \subseteq Q \times Q$ a transition relation;
- $I : Q \rightarrow 2^{\text{Ent}} \times (X \rightarrow \text{Ent})$ a partial function yielding for every *initial state* $q \in \text{dom}(I)$ an *initial valuation* (N, θ) , where $N \subseteq E_q$ is a finite set of entities, and $\theta : X \rightarrow E_q$ is a partial valuation of the variables in X ;
- $\mathcal{F} \subseteq 2^Q$ a set of sets of accept states.

Notational conventions: we write $(q, N, \theta) \in I$ for $I(q) = (N, \theta)$ and $q \rightarrow q'$ for $(q, q') \in \rightarrow$ and γ_q for $E(q) = \gamma$. We adopt the generalised Büchi acceptance condition, i.e., $\rho = q_0 q_1 q_2 \dots$ is a *run* of ABA \mathcal{A} if $q_i \rightarrow q_{i+1}$ for all $i \in \mathbb{N}$ and $|\{i | q_i \in F\}| = \omega$ for all $F \in \mathcal{F}$. Let $\text{runs}(\mathcal{A})$ denote the set of runs of \mathcal{A} . Run $\rho = q_0 q_1 q_2 \dots$ is said to *accept* the (unfolded) allocation sequence $\sigma = (E_{q_0}, \mu_{q_0})(E_{q_1}, \mu_{q_1})(E_{q_2}, \mu_{q_2}) \dots$. Let

$$\mathcal{L}(\mathcal{A}) = \{(\sigma, N, \theta) | \exists \rho = q_0 q_1 q_2 \dots \in \text{runs}(\mathcal{A}) : \rho \text{ accepts } \sigma \text{ and } I(q_0) = (N, \theta)\}.$$

The essential difference w.r.t. ABA (without references) introduced in Definition 4.3.1 (cf. Section 4.3.1) is that states have associated a configuration instead of just a set of entities. Note that every configuration in an ABA has the unitary cardinality function. Thus, every entity in an ABA is concrete.

Notation on morphisms. In the previous definition, the set Q is introduced because in the definition of the model checking algorithm it is necessary to define the duplication of HABAs as this was done in Section 4.5.1 for HABA without references. Apart from this very technical reason, it could be possible to take as set of states a set of configurations dropping therefore the component E . The introduction of Q makes it convenient from now on — both for ABA as well as for HABA (cf, Definition 5.3.21) — to talk about morphisms on states. In that case, we intend to refer to morphisms actually defined on the configuration associated with those states. Therefore, in the rest of this chapter we will freely interchange between states and configurations writing then $h : q \succrightarrow q'$ for $h : \gamma_q \succrightarrow \gamma_{q'}$ and $q \succrightarrow q'$ for $\gamma_q \xrightarrow{h} \gamma_{q'}$.

5.3.3 Reallocations and HABA

The HABAs (with references) that we will define in this section are intended to be used as semantical models for programming languages. The execution of statements is reflected in these automata by some modification on the structure of the graph defined by a (configuration of a) state. Therefore, from state to state, the abstract shape of the graph is only preserved for those parts not involved in the execution of the statement. In particular, modifications in the structure arise because:

- i. entities may be allocated or deallocated (e.g., because of some creation/deletion mechanism of the language such as `new` in Java and `delete` in C++);
- ii. from state to state some references (pointers) between entities may change (e.g., because of assignments);
- iii. furthermore, multiple or unbounded entities may be disassembled into several entities, giving a more concrete structure.

The first two cases correspond to real changes in the structure of the configuration, whereas the third case corresponds to a change of representation of the configuration in terms of the level of abstraction.

Example 5.3.11. Consider configurations γ'_1 and γ'_2 in Figure 5.2. They resemble configurations γ_1 and γ_2 of Figure 5.1: in particular, γ'_1 is the same as configuration γ_1 , whereas γ'_2 is a slight modification of γ_2 . Configuration γ'_2 is obtained from γ'_1 by the following alterations:

- replacing the reference between e_0 and e_1 by the reference between e_0 and e_4 . This may reflect the execution of an assignment.

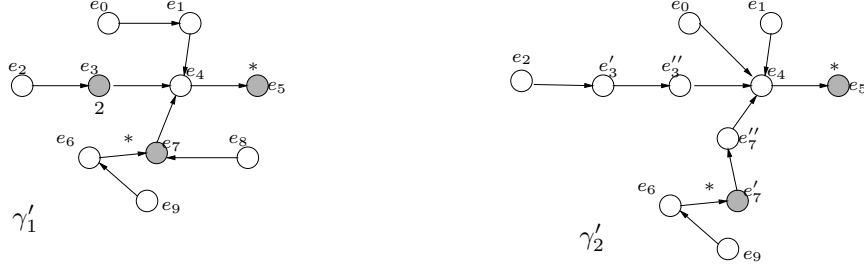


Figure 5.2: Two (configurations of) states not related by a morphism.

- Entity e_8 does not exist in γ'_2 . This may reflect execution of a `del` statement.
- Entities e'_3 and e''_3 are materialised from e_3 . This usually is not a direct consequence of the execution of a statement. However, it can be considered as a kind of rearrangement of the shape of the configuration that can be useful in particular situations (cf. the assignment rule of the symbolic semantics in Section 5.6.4).

Note that there does not exist a morphism between γ'_1 and γ'_2 because the two graphs do not represent the same pointer structure. \square

These considerations show that, between states related by a transition in the automaton, we need a *weaker* notion of correspondence than the one defined by a morphism. Namely, the correspondence must be only partial, in the sense that we might have only correspondence of a subgraph but not of the complete graph. Moreover, in order to faithfully model modification *iii*), for every entity in a state we need to associate possibly more than one entity of the other state. This weaker notion of correspondence is captured by the definition of reallocation that is introduced below.

Preliminary notation. Given two sets A_1, A_2 and a relation $\mathcal{R} \subseteq A_1 \times A_2$, if $a \in A_1$ we indicate by $\mathcal{R}(a) = \{a' \in A_2 \mid (a, a') \in \mathcal{R}\}$ and if $a \in A_2$ we write $\mathcal{R}^{-1}(a) = \{a' \in A_1 \mid (a', a) \in \mathcal{R}\}$. For a subset $A \subseteq A_1$, we write $\mathcal{R}(A) = \bigcup_{a \in A} \mathcal{R}(a)$.

A *multiset* \mathcal{M} of a given set A is a function $\mathcal{M} : A \rightarrow \mathbb{N}$. For $a \in A$, the image $\mathcal{M}(a)$ is called the *multiplicity* of a in \mathcal{M} . We write $a \in \mathcal{M}$ if $\mathcal{M}(a) \neq 0$ (and $a \notin \mathcal{M}$ if $\mathcal{M}(a) = 0$). The sum of two multisets \mathcal{M}_1 and \mathcal{M}_2 of A , denoted by $\mathcal{M}_1 + \mathcal{M}_2$, is defined as $(\mathcal{M}_1 + \mathcal{M}_2)(a) = \mathcal{M}_1(a) + \mathcal{M}_2(a)$ for all $a \in A$.

Moreover, given a configuration γ we extend to sets the relation \prec_γ as follows: let $E, E' \subseteq E_\gamma^\perp$ then

$$E \prec_\gamma E' \Leftrightarrow (E = E' = \{\perp\}) \vee (\forall e \in E : \exists e' \in E' \setminus \{\perp\} : e \prec_\gamma e') \quad (5.2)$$

We now introduce the concept of reallocation that will be essential throughout this chapter.

Definition 5.3.12. (reallocation) Let $(E_1, \prec_1, \mathcal{C}_1)$, $(E_2, \prec_2, \mathcal{C}_2)$ be two configurations. A *reallocation* λ is a function $\lambda : E_1^\perp \times E_2^\perp \rightarrow \mathbb{M}^*$ such that:

1. for all $e \in E_1 : \mathcal{C}_1(e) = \sum_{e' \in E_2^\perp} \lambda(e, e')$
2. for all $e \in E_2 : \mathcal{C}_2(e) = \sum_{e' \in E_1^\perp} \lambda(e', e)$
3. for all $e \in E_1 : |\{e' \in E_2 \mid \lambda(e, e') = *\}| \leq 1$ and $|\{e' \in E_2 \mid \lambda(\perp, e') = *\}| = 0$
4. for all $e \in E_1 : \{e' \in E_2^\perp \mid \lambda(e, e') \neq 0\}$ is a chain;
5. for all $e \in E_2 : \{e' \in E_1^\perp \mid \lambda(e', e) \neq 0\}$ is a chain.

We write $\lambda : (E_1, \prec_1, \mathcal{C}_1) \Rightarrow (E_2, \prec_2, \mathcal{C}_2)$ or $(E_1, \prec_1, \mathcal{C}_1) \xrightarrow{\lambda} (E_2, \prec_2, \mathcal{C}_2)$ if there exists a reallocation λ from $(E_1, \prec_1, \mathcal{C}_1)$ to $(E_2, \prec_2, \mathcal{C}_2)$.

A reallocation is a multiset³, with range \mathbb{M}^* . Accordingly, we write $(e, e') \in \lambda$ if $\lambda(e, e') \neq 0$ and $(e, e') \notin \lambda$ if $\lambda(e, e') = 0$. Furthermore, it is often convenient to treat λ as a relation over $Ent \times Ent \times \mathbb{M}^*$. In this case, we write $(e, e', n) \in \lambda$ if and only if $\lambda(e, e') = n$. Another notation we use (when convenient) is: $\lambda(e) = \{e' \mid (e, e') \in \lambda\}$ and $\lambda^{-1}(e) = \{e' \mid (e', e) \in \lambda\}$. Finally we write $\text{dom}(\lambda) = \{e \neq \perp \mid \exists e' \neq \perp : (e, e') \in \lambda\}$ and symmetrically $\text{cod}(\lambda) = \{e \neq \perp \mid \exists e' \neq \perp : (e', e) \in \lambda\}$.

The element \perp is used by the reallocations in order to model birth and death. More precisely, the birth of an entity $e \in E_2$ is modelled by relating it to \perp , i.e. $\lambda(\perp, e) \neq 0$. Symmetrically, λ models the death of an $e \in E_1$ by relating it to \perp , i.e., $\lambda(e, \perp) \neq 0$.

A reallocation allows to change dependencies between entities, in fact if $e_1 \prec_1 e_2$ it is not required that entities in $\lambda(e_1)$ precede entities in $\lambda(e_2)$ according to \prec_2 (as is enforced by the definition of morphism, cf. Definition 5.3.3). The above definition requires the cardinality of entities in the source state to be consistent with the multiplicity of the outgoing arcs (condition 1). Symmetrically, condition 2 forces consistency between the multiplicity of outgoing arcs and the cardinality of entities in the target state. Condition 3 is meant to restrict the amount of nondeterminism derived by unbounded entities and to avoid the creation of unbounded entities. The first part of this condition can be rephrased by saying that the unknown cardinality of an unbounded entity e is reallocated only in another unbounded entity e' whereas for every other entity in $\lambda(e) \setminus \{e'\}$ the precise weight is known. By conditions 4 and 5, the image as well as the inverse image of entities under λ are chains.

Example 5.3.13. Figure 5.3 shows two possible reallocations between (configurations of) states. Note that Definition 5.3.12 does not exclude the crossing

³Strictly speaking it is slightly different since it may give multiplicity $*$ to a pair (e, e') .

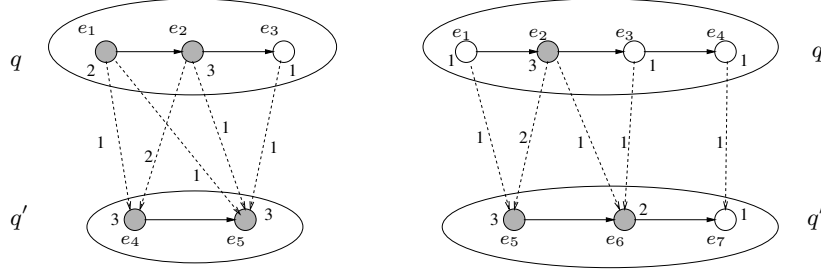


Figure 5.3: Reallocation examples.

of relations between entities from one state to the other. For example, in the reallocation depicted on the left of the figure part of e_2 is reallocated to e_4 while part of e_1 (that precedes e_2) is reallocated to e_5 (that is preceded by e_4). It is also not difficult to see that the (configuration of) states in Figure 5.2 are related by a reallocation. \square

Lemma 5.3.14 (impartiality). Let γ, γ' be two configurations. If $\gamma \xrightarrow{\lambda} \gamma'$ then:

- $\forall e \in E_{\gamma'} : (\lambda(\perp, e) \neq 0 \Leftrightarrow \forall e' \in E_{\gamma} : \lambda(e', e) = 0)$ (Common Birth)
- $\forall e \in E_{\gamma} : (\lambda(e, \perp) \neq 0 \Leftrightarrow (\forall e' \in E_{\gamma'} : \lambda(e, e') = 0))$. (Common Death)

Proof. By contradiction assume (Common Birth) does not hold, i.e., there exists $e \in E_{\gamma'}$ and $e' \in E_{\gamma}$ such that $\perp, e' \in \lambda^{-1}(e)$. By condition 5 of the definition of reallocation, $\lambda^{-1}(e)$ is a chain, but this is impossible since \perp cannot be related with entities to form a chain. Symmetrically, λ satisfies (Common Death) since for all $e \in E_{\gamma}$ by condition 4, $\lambda(e)$ is a chain and therefore cannot contain at the same \perp and some entities of $E_{\gamma'}$. \square

The previous lemma describe the impartiality property that a reallocation enjoys w.r.t. birth and death of entities (Common Birth) says that if an entity is born, it cannot be related with any entity except than \perp . Symmetrically, (Common Death) states that if an entity dies than it can be related only with \perp . The combination of condition 1 of the reallocation definition and (Common Death) (and symmetrically condition 2 and (Common Birth)) forces to transfer the complete cardinality of e by $\lambda(e, \perp)$ (and symmetrically $\lambda(\perp, e)$).

Example 5.3.15. The reallocation depicted on the left part of Figure 5.4 violates both (Common Birth) and (Common Death). In q , e_1 has two outgoing arcs. One would indicate that part of e_1 dies whereas the other arc reallocates part of e_1 onto e_4 . Similarly, the two arcs starting from e_2 and e_3 and reaching e_5 denote that the latter is old. However, the incoming arc from \perp indicates that e_5 represents also some new entities. (Common Birth) and (Common Death)

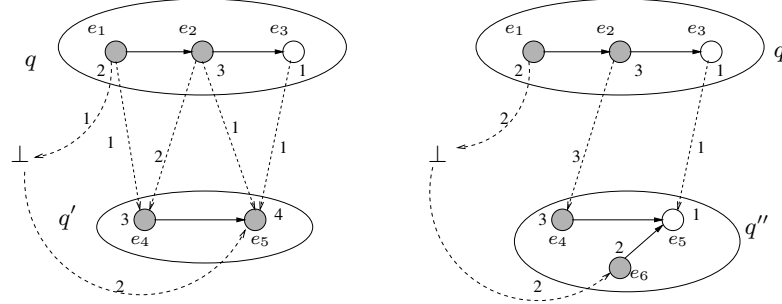


Figure 5.4: On the left a reallocation violating (Common Birth)/(Common Death); on the right a reallocation satisfying both these properties.

state that, by conditions 4 and 5, a reallocation rules out explicitly these kinds of ambiguities. The reallocation on the right part of the same figure satisfies (Common Birth) and (Common Death). e_1 dies by mapping the complete cardinality onto \perp . Similarly e_6 is new since its cardinality is all provided by \perp . \square

The relation between morphisms and reallocations is described by the following:

Proposition 5.3.16. Let γ_1 and γ_2 be two configurations. If $\gamma_1 \succ^h \gamma_2$ then $\gamma_1 \xrightarrow{\lambda} \gamma_2$ where let $e \in E_{\gamma_1}$ and $e' \in E_{\gamma_2}$:

$$\lambda(e, e') = \begin{cases} \mathcal{C}_{\gamma_1}(e) & \text{if } e' = h(e) \\ 0 & \text{otherwise.} \end{cases}$$

Proof. See Appendix B.1. \square

Definition 5.3.17. If $\gamma_1 \xrightarrow{\lambda} \gamma_2$ and $\gamma'_1 \xrightarrow{\lambda'} \gamma'_2$ with $\mathcal{C}_{\gamma'_1} = \mathcal{C}_{\gamma'_2} = \mathbf{1}$, we say that λ' is a *concretion* of λ , denoted $\lambda' \triangleright \lambda$, if and only if there exist h_1, h_2 such that:

1. $\gamma'_1 \xrightarrow{h_1} \gamma_1$ and $\gamma'_2 \xrightarrow{h_2} \gamma_2$
2. $\lambda : (e_1, e_2) = \sum_{(e'_1, e'_2) = (h_1(e'_1), h_2(e'_2))} \lambda'(e'_1, e'_2)$.
3. (No-Cross) $\forall e, e' \in E_{\gamma'_1}$:

$$h_1(e) = h_1(e') \vee h_2(\lambda'(e)) = h_2(\lambda'(e')) \Rightarrow (e \prec_{\gamma'_1} e' \Leftrightarrow \lambda'(e) \prec_{\gamma'_2} \lambda'(e')).$$
4. $\forall e \in E_{\gamma'_2} : (\mathcal{C}_{\gamma_2}(h_2(e)) = * \Rightarrow e \in \text{cod}(\lambda'))$;

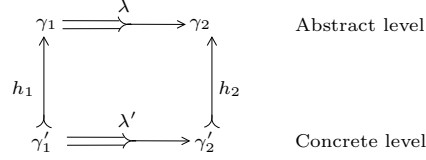


Figure 5.5: Commutative diagram for concretion of reallocations.

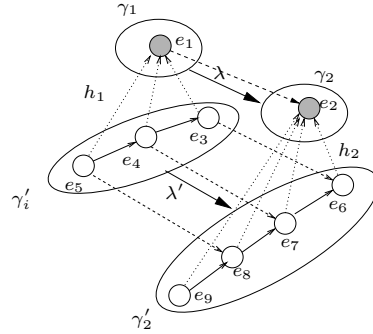


Figure 5.6: Incompatibility between new entities of the symbolic w.r.t. the concrete level.

A concretion λ' of λ via h_1 and h_2 is a reallocation that makes the diagram depicted in Figure 5.5 commute. Informally, λ' can be seen as a reallocation that agrees with λ , but that is defined on a concrete version of γ_1 and γ_2 , i.e., λ' is defined on configurations γ'_1 and γ'_2 related to γ_1 and γ_2 by morphisms. Condition 4 says that if an entity e is projected on an unbounded one then e must be old. This gives a correspondence between the number of new entities in γ_2 and γ'_2 . A reallocation and its concretions have the same behaviour in terms of fresh entities (cf. Lemma 5.3.19). Note that this condition is necessary. In fact, consider the reallocation represented in Figure 5.6 assuming that $M = 2$ and $\mathcal{C}_{\gamma_1}(e_1) = \mathcal{C}_{\gamma_2}(e_2) = *$. We have $\lambda' \triangleright \lambda$, and although $e_9 \in h_2^{-1}(\text{cod}(\lambda))$, it is new. This circumstance is ruled out by condition 4. Condition (No-Cross) prevents the concretion to change the order of entities that are mapped onto a multiple/unbounded entity or that have as image the same multiple/unbounded entity. In fact, since a multiple or an unbounded entity corresponds to a chain we want that the order of the concrete entities corresponding to this chain is not modified after the reallocation. For example, in Figure 5.7 (up), λ' is a concretion of the reallocation in the right part of Figure 5.3, whereas λ'' (down) is not since it violates (No-Cross). In particular in Figure 5.7 (down), entities e'_5, e''_5, e'''_5 are mapped on the same abstract entity e_5 . However, their order does not correspond to the order of the (concrete) inverse image e'_1, e'_2, e''_2 . Vice-versa, the order of the image of e'_2, e''_2 does not correspond to the order of their image e'''_5 and e'_5 . The same phenomenon occurs for e'_2, e'_3, e'_6 and e''_6 .

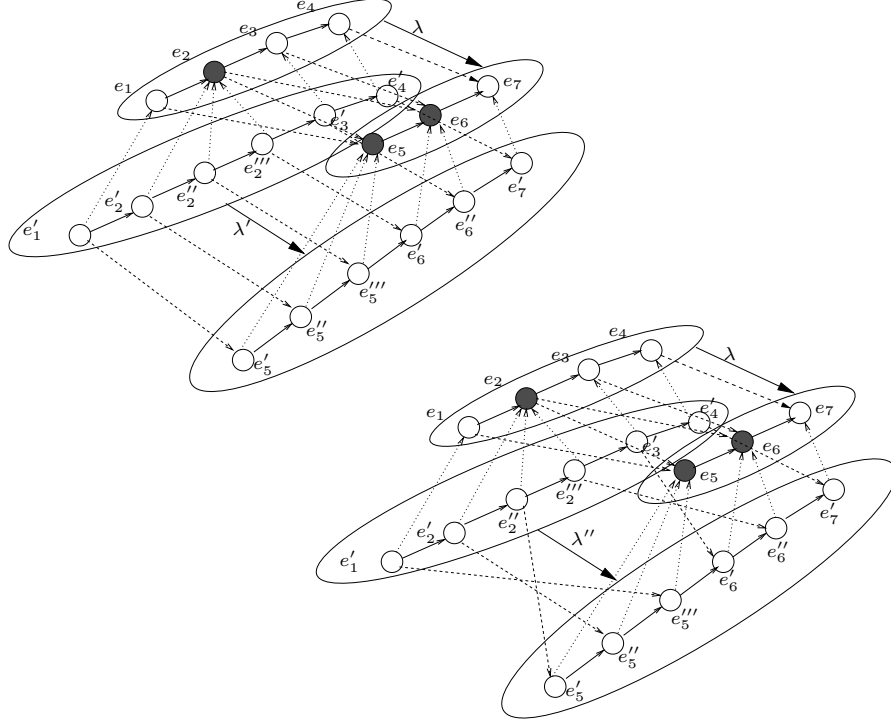


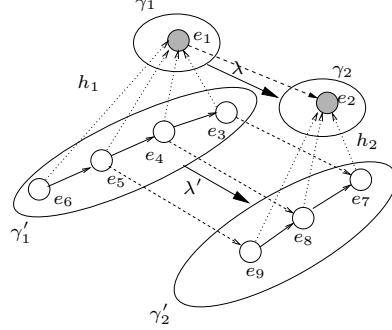
Figure 5.7: Concretions prevent reshuffling of entities in the concrete states.

A consequence of constraint (No-Cross) is that given an abstract entity $e \in E_{\gamma_1}$, its associated concrete entities — i.e., all the elements of the set $h_1^{-1}(e)$ — enjoy a *common fate*: either *all* of them survive the reallocation or *all* of them die. This fact is formalised in the following:

Lemma 5.3.18 (common fate). If $\gamma_1 \xrightarrow{\lambda} \gamma_2$ and $\gamma'_1 \xrightarrow{\lambda'} \gamma'_2$ and $\lambda' \triangleright \lambda$, then

$$\forall e \in E_{\gamma_1} : \forall e', e'' \in h_1^{-1}(e) : (\lambda'(e') = \{\perp\} \Leftrightarrow \lambda'(e'') = \{\perp\}).$$

Proof. By contradiction. Assume there exist $e', e'' \in h_1^{-1}(e)$ such that $\lambda'(e') = \{\perp\}$ and $\lambda'(e'') = \{\tilde{e}\} \neq \{\perp\}$. We can choose e', e'' such that $e' \prec_{\gamma'_1} e''$ or $e'' \prec_{\gamma'_1} e'$. Assume $e' \prec_{\gamma'_1} e''$. By (No-Cross), since $h_1(e') = h_1(e'') = e$ and $e' \prec_{\gamma'_1} e''$ it follows $\lambda'(e') \prec_{\gamma'_2} \lambda'(e'')$ which — by (5.2) — implies that $\perp \prec_{\gamma'_2} \tilde{e}$ that is impossible. On the other hand, assume $e'' \prec_{\gamma'_1} e'$ that by (No-Cross) implies $\lambda'(e'') \prec_{\gamma'_2} \lambda'(e')$. But then by (5.2) there must exist a $e''' \in E_{\gamma'_2} \cap \lambda'(e')$ such that $\tilde{e} \prec_{\gamma'_2} e'''$. But again this is a contradiction because $E_{\gamma'_2} \cap \lambda'(e') = \emptyset$. \square

Figure 5.8: λ' violates the common fate property.

Note that the property of common fate does not hold in absence of (No-Cross). The reason has to be sought among the large variety of consequences yielded by unbounded entities. Consider Figure 5.8, and assume $M = 2$. Since $\lambda(e_1, e_2) = *$, the second condition of \triangleright does not reveal that e_6 is deallocated because $\lambda(e_3, e_7) \oplus \lambda(e_4, e_8) \oplus \lambda(e_5, e_9) = *$. However, λ' does not satisfies (No-Cross) because $e_6 \prec_{\gamma'_1} e_5$ but $\lambda'(e_6) \not\prec_{\gamma'_2} \lambda'(e_5)$. Therefore λ and λ' do not have the same behaviour w.r.t. deallocation of entities related by morphisms.

Lemma 5.3.19. If $\gamma_1 \xrightarrow{\lambda} \gamma_2$ and $\gamma'_1 \xrightarrow{\lambda'} \gamma'_2$ and $\lambda' \triangleright \lambda$ via h_1 and h_2 then:

- $E_{\gamma'_2} \setminus \text{cod}(\lambda') = h_2^{-1}(E_{\gamma_2} \setminus \text{cod}(\lambda))$
- $E_{\gamma_2}^* \subseteq \text{cod}(\lambda)$.

Proof. See Appendix B.1. □

The relation \triangleright is neither reflexive nor symmetric, but, it is transitive:

Lemma 5.3.20 (\triangleright transitivity). Let $q_1 \xrightarrow{\lambda} q_2$, $q'_1 \xrightarrow{\lambda'} q'_2$ and $q''_1 \xrightarrow{\lambda''} q''_2$. Then:

$$(\lambda'' \triangleright \lambda' \wedge \lambda' \triangleright \lambda) \Rightarrow \lambda'' \triangleright \lambda.$$

Proof. See Appendix B.1 □

We are now in the position to define the enhancement of HABAs able to deal with pointer structures.

Definition 5.3.21. A *High-level ABA* (HABA) with references \mathcal{H} is a tuple $\langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ with X, Q, \mathcal{F} as in Def. 5.3.10, and

- $E : Q \rightarrow \text{CONF}$ a function that associates to each state $q \in Q$ a configuration $\gamma_q = (E_q, \mu_q, \mathcal{C}_{E_q})$.
- $\rightarrow \subseteq Q \times (\text{Ent} \times \text{Ent} \rightarrow \mathbb{M}^*) \times Q$, such that if $q \rightarrow_\lambda q'$ then $\gamma_q \xrightarrow{\lambda} \gamma_{q'}$.

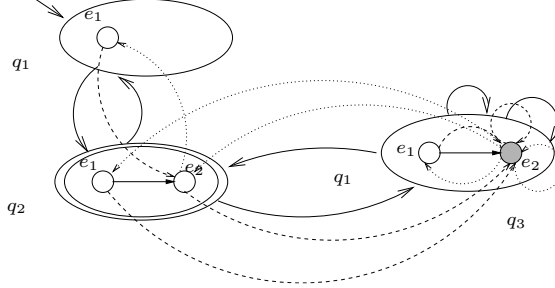


Figure 5.9: Example HABA with references modelling a stack.

- $I : Q \rightarrow 2^{Ent} \times (X \rightarrow Ent)$ a partial function yielding for every *initial state* $q \in \text{dom}(I)$ an *initial valuation* (N, θ) , where $N \subseteq (E_q \setminus E_q^*)$ is a finite set of (bounded) entities, and $\theta : X \rightarrow E_q$ is a partial valuation of the variables in X .

Whereas in an ABA state we only have concrete entities, this is no longer true in a HABA: the corresponding enforcing condition on the cardinality function of configurations (of an ABA) is now relaxed. The condition $N \subseteq (E_q \setminus E_q^*)$ on the initial state is needed since we want to keep track of the number of new entities in every state. For HABAs without references (cf. Definition 4.3.4) this was not necessary as the black-hole is not a proper entity.

Sometimes it is useful to specify, the precise range of the cardinalities used in a given HABA \mathcal{H} . We write $\mathcal{C}(\mathcal{H}) = M$ if M is the global upper bound on the cardinality of the entities, i.e., if for all $q \in Q_{\mathcal{H}}$, $\text{cod}(\mathcal{C}_q) \subseteq \mathbb{M}^*$.

Example 5.3.22. Figure 5.9 shows an example HABA with references. It models a system where at every step either an entity is created or deallocated. Creation/deletion follows a LIFO policy (i.e., the system modelled is a stack). From state to state there are two transitions: one with a dashed reallocation that models creation (and insertion in the stack), and a transition with a dotted reallocation that models the deletion (and extraction from the stack). For example, in state q_3 , entities that are created are accumulated in e_2 (following the dashed reallocations). The dotted reallocation extracts entities by splitting e_2 and remapping it on e_1 and e_2 respectively. Entity e_1 is not remapped and therefore deallocated. The system starts with only one entity in the initial state q_1 . The accept state is q_2 , therefore the automaton models a system in which every computation has infinitely many times a stack with two entities. \square

HABA with references are used to generate models for $\mathcal{M}allTL$. The mechanism exploited here is similar to the one used in Chapter 4 for $\mathcal{A}llTL$ and HABA without references.

Definition 5.3.23. A *folded allocation sequence* is an infinite alternating sequence

$$(E_0, \mu_0, \mathbf{1}_{E_0})\lambda_0(E_1, \mu_1, \mathbf{1}_{E_1})\lambda_1 \cdots$$

where for $i \geq 0$, $(E_i, \mu_i, \mathbf{1}_{E_i}) \xrightarrow{\lambda_i} (E_{i+1}, \mu_{i+1}, \mathbf{1}_{E_{i+1}})$.

Note that because of the unitary cardinality functions, in the previous definition, λ_i ($i \geq 0$) may associate at most one entity in E_{i+1} to an entity in E_i . Another consequence is that folded allocation sequences can be used together with to obtain an alternative semantics for $\mathcal{N}\acute{a}\ell\ell\text{TTL}$ denoted by \models_f . This is done by giving a new definition for N_i^σ and θ_i^σ (defined in page 130) as follows. For an allocation triple (σ, N, θ) let:

$$N_i^\sigma = \begin{cases} N & \text{if } i = 0 \\ E_i^\sigma \setminus \text{cod}(\lambda_{i-1}^\sigma) & \text{otherwise.} \end{cases} \quad (5.3)$$

$$\theta_i^\sigma = \begin{cases} \theta & \text{if } i = 0 \\ \lambda_{i-1}^\sigma \circ \theta_{i-1}^\sigma & \text{otherwise} \end{cases} \quad (5.4)$$

where the composition between a (concrete) reallocation

$$(E, \mu, \mathbf{1}_E) \xrightarrow{\lambda} (E', \mu', \mathbf{1}_{E'})$$

and a valuation of the logical variables $\hat{\theta}$ is given by:

$$\lambda \circ \hat{\theta}(x) = \begin{cases} e & \text{if } \hat{\theta}(x) \neq \perp \text{ and } \lambda(\hat{\theta}(x), e) = 1 \\ \perp & \text{otherwise.} \end{cases}$$

The semantics given by \models_f is equivalent to \models_u , hence folded and unfolded allocation sequences are equivalent models for $\mathcal{N}\acute{a}\ell\ell\text{TTL}$ (the reader is referred to Section 4.2.4 for details).

Runs of a HABA generate folded allocation sequences. The correspondence between allocation triples and runs of HABA is given in terms of morphisms in the following definition.

Definition 5.3.24 (generator). A run $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ of HABA $\mathcal{H} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ generates an allocation triple (σ, N, θ) , where $\sigma = \gamma_0^\sigma \lambda_0^\sigma \gamma_1^\sigma \dots$ is a folded allocation sequence, if there is a *generator*, i.e., a family of morphisms h_i from γ_i^σ to γ_{q_i} satisfying for all $i \geq 0$:

1. $\lambda_i^\sigma \triangleright \lambda_i$ via h_i and h_{i+1} ;
2. $I(q_0) = (N', h_0 \circ \theta)$ and $N = h_0^{-1}(N')$;

By condition 1, the morphism in the generator must preserve the reallocations in the run. In particular, λ_i^σ is a concretion of λ_i (for $i \geq 0$). Condition 2 imposes a correspondence between the set of initial entities and the interpretation of the logical variables in the state q_0 and the first state of the allocation sequence.

Runs of a HABA are defined in the same way as for ABA. Let $\text{runs}(\mathcal{H})$ denote the set of runs of \mathcal{H} and

$$\mathcal{L}(\mathcal{H}) = \{(\sigma, N, \theta) \mid \exists \rho \in \text{runs}(\mathcal{H}) : \rho \text{ generates } (\sigma, N, \theta)\}.$$

5.4 Relating HABA and ABA

In this section, we study how to relate ABAs and HABAs with references w.r.t. their generated languages and, therefore, the set of $\mathcal{N}\ell\ell\text{TTL}$ -formulae that they satisfy. First of all, we introduce a simulation preorder (denoted \sqsubseteq) between a HABA and a concrete one (i.e., a HABA where every entity is concrete). The intuition behind \sqsubseteq is that whenever $\mathcal{H} \sqsubseteq \mathcal{H}'$, then \mathcal{H}' can simulate all behaviours of \mathcal{H} , but the converse does not necessarily hold. That is, \mathcal{H}' may exhibit more behaviours than \mathcal{H} .

Definition 5.4.1. Let $\mathcal{H} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ such that $\mathcal{C}(\mathcal{H}) = 1$ and $\mathcal{H}' = \langle X, Q', E', \rightarrow', I', \mathcal{F}' \rangle$.

1. A binary relation $\sqsubseteq \subseteq Q \times (Ent \rightarrow Ent) \times Q'$ is a *simulation* if and only if for all $q_1 \sqsubseteq_{h_1} q'_1$ it holds:

- a) $q_1 \xrightarrow{h_1} q'_1$
- b) if $q_1 \rightarrow_\lambda q_2$ then $\exists \lambda', h_2$ such that
 - i) $q'_1 \rightarrow_{\lambda'} q'_2 \wedge q_2 \sqsubseteq_{h_2} q'_2$
 - ii) $\lambda \triangleright \lambda'$ via h_1, h_2

2. \mathcal{H}' *simulates* \mathcal{H} (written $\mathcal{H} \sqsubseteq \mathcal{H}'$) if and only if there exists a simulation $\sqsubseteq \subseteq Q \times (Ent \rightarrow Ent) \times Q'$ such that:

- a) for all $q \in I$ there exists $q' \in I'$ and h with $I'(q') = (N, h \circ \theta)$ such that:
 - $q \sqsubseteq_h q'$ and
 - $I(q) = (h^{-1}(N), \theta)$;
- b) there exists a bijective $\psi : \mathcal{F} \rightarrow \mathcal{F}'$ such that
$$\forall F \in \mathcal{F} : (\forall q \in F : (\exists q' \in \psi(F), \exists h : q \sqsubseteq_h q'))$$

Hence, a state q' simulates q if they represent the same pointer structure (condition 1.a). Every transition of q can be simulated by q' with a proper transition whose reallocation is a concretion of the reallocation of the transition of q . The target states simulate each other and (because of \triangleright) have a corresponding set of new entities (condition 1.b).

The second part of the previous definition specifies when a HABA simulates another one. Condition 2.a) requires that every initial state in \mathcal{H} has a corresponding (simulating) initial state in \mathcal{H}' and there is correspondence in the initial valuation. Condition 2.b) says that every accept state in $q \in F \in \mathcal{F}$ is simulated by an accept state $q' \in F' \in \mathcal{F}'$. The bijective function ψ enforces that every $F' \in \mathcal{F}'$ is considered so that the generalised Büchi acceptance condition of \mathcal{H} is ensured to exist in \mathcal{H}' .

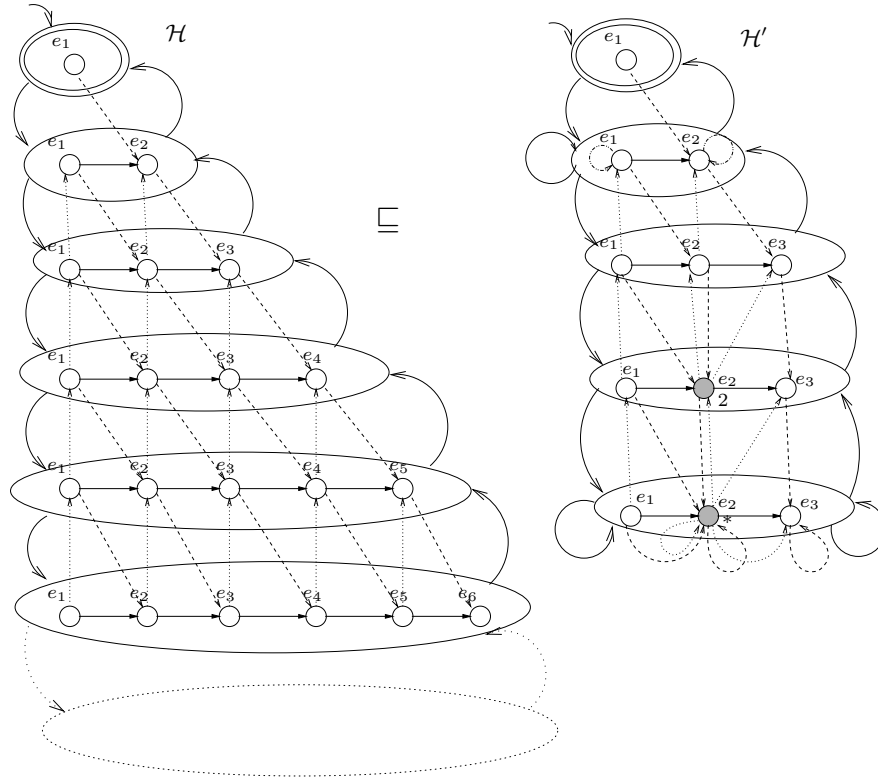


Figure 5.10: A HABA simulating the one in Figure 5.10

Example 5.4.2. The HABA \mathcal{H} depicted in Figure 5.10 (left) models a FIFO queue. In the initial state there is only one entity. At every step either a new entity is created and enqueued (transitions with dashed reallocations), or the system nondeterministically performs a step where the first entity of the queue is extracted and deallocated (transitions with dotted reallocations). \mathcal{H} is infinite-state since the queue can grow unboundedly. The HABA \mathcal{H}' (where $\mathcal{C}(\mathcal{H}') = 2$), depicted on the right side of the figure, simulates \mathcal{H} . \square

The comparison between HABAs and ABAs involves also a notion of isomorphic folded allocation sequences. The concept is introduced in the next definition.

Definition 5.4.3.

- Two folded allocation sequences σ_1, σ_2 are *isomorphic* (written $\sigma_1 \cong \sigma_2$) if there exists a family of isomorphisms $(h_i)_{i \in \mathbb{N}}$ such that for all $i \geq 0$,
 - $h_i : \sigma_1[i] \rightarrow \sigma_2[i]$

2. $\lambda_i^{\sigma_1} \triangleright \lambda_i^{\sigma_2}$ via h_i, h_{i+1} .

- Two allocation triples (σ, N, θ) and (σ', N', θ') are *isomorphic* (written $(\sigma, N, \theta) \cong (\sigma', N', \theta')$) if $\sigma \cong \sigma'$, $\text{dom}(\theta) = \text{dom}(\theta')$, $N' = h_0(N)$ and $\theta' = h_0 \circ \theta$.

Isomorphic allocation triples are the same up to renaming of entities. It is therefore not surprising that they satisfy the same set of $\mathcal{N}all\text{TL}$ formulae, as stated in the next result.

Proposition 5.4.4. For $\mathcal{N}all\text{TL}$ formula ϕ and folded allocation sequence σ, σ' :

$$(\sigma, N, \theta) \cong (\sigma', N', \theta') \Rightarrow (\sigma, N, \theta \models \phi \text{ iff } \sigma', N', \theta' \models \phi).$$

Proof. Straightforward by induction on the structure of ϕ . \square

The next important proposition states the relation between the language of HABAs related by a simulation relation.

Theorem 5.4.5. If $\mathcal{H} \sqsubseteq \mathcal{H}'$ then $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{H}')$.

Proof. See Appendix B.1. \square

In case of HABAs without references defined in Chapter 4 we had equality between the language of \mathcal{H} and $\text{Exp}(\mathcal{H})$. Here, the situation is more complex because of the abstraction by unbounded entities. \mathcal{H}' describes more behaviours of \mathcal{H} , or equivalently we can say that, \mathcal{H}' models spurious behaviours that are not actually present in the concrete system at hand. Nevertheless, the definition of simulation provides a means to compare different models w.r.t. the satisfiability of $\mathcal{N}all\text{TL}$ formulae.

Definition 5.4.6. Given a HABA \mathcal{H} and a $\mathcal{N}all\text{TL}$ -formula ϕ we say that:

- ϕ is \mathcal{H} -satisfiable if there exists $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$ such that $\sigma, N, \theta \models \phi$;
- ϕ is \mathcal{H} -valid if for all $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}) : \sigma, N, \theta \models \phi$.

The previous definition corresponds to the one given for $\mathcal{A}ll\text{TL}$ -formulae. It implies that ϕ is \mathcal{H} -valid (\mathcal{A} -valid) if and only if $\neg\phi$ is not \mathcal{H} -satisfiable (\mathcal{A} -satisfiable). For ABA \mathcal{A} the definition for \mathcal{A} -satisfiability and \mathcal{A} -validity can be given in precisely the same way.

The relation between models related by a simulation relation is given by the following result.

Proposition 5.4.7. For HABAs \mathcal{H} and \mathcal{H}' such that $\mathcal{H} \sqsubseteq \mathcal{H}'$ and $\mathcal{N}all\text{TL}$ -formula ϕ :

$$\phi \text{ is } \mathcal{H}'\text{-valid} \Rightarrow \phi \text{ is } \mathcal{H}\text{-valid}.$$

Proof. Straightforward from Theorem 5.4.5. \square

The converse does not hold as illustrated in the next example.

Example 5.4.8. Consider the following formula:

$$\phi \equiv \mathbf{G}[\mathbf{X}(\exists x : x \text{ new} \wedge x.a \neq \text{nil}) \vee (\exists y : y.a = \text{nil} \wedge \mathbf{X}y \text{ dead})] \quad (5.5)$$

expressing that in *every step* of the system either a new entity is created (which is enqueued) or an existing entity not pointing to any other one (i.e., the first of the queue) is deallocated. It is easy to see that ϕ holds in every allocation sequence generated by a run of \mathcal{H} in Figure 5.10, i.e., ϕ is \mathcal{H} -valid. However, ϕ is *not* \mathcal{H}' -valid because, in the second state of \mathcal{H}' there exists a self loop with a reallocation mapping every entity to itself. Note that \mathcal{H} does not have such transition. By taking this self-loop no entities are created or destroyed. Hence, ϕ does not hold in every allocation sequence generated by \mathcal{H}' . \square

Relating folded and unfolded allocation sequences. As for $\mathcal{A}\ell\text{TL}$, we can relate folded and unfolded allocation sequences with references in a straightforward manner. The next definition introduces a special kind of reallocation which is useful in order to establish this relation⁴.

Definition 5.4.9. For two configurations γ and γ' such that $\mathcal{C} \upharpoonright E_\gamma \cap E_{\gamma'} = \mathcal{C}' \upharpoonright E_\gamma \cap E_{\gamma'}$, a reallocation $\gamma \xrightarrow{\lambda} \gamma'$ is called *identity* reallocation if for all $e \in E_\gamma$ and $e' \in E_{\gamma'}$:

$$id(e, e') = \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e = e' \\ 0 & \text{otherwise.} \end{cases}$$

We indicate an identity reallocation by *id*.

Two configurations (and therefore states) related by the identity reallocation may differ only in the links between entities and because some entity has born or has died. The condition on E_γ and $E_{\gamma'}$ says that an entity not related to itself must be related to \perp (in which case is born or dies).

For an unfolded allocation sequence $\sigma = (E_0, \mu_0, \mathbf{1})(E_1, \mu_1, \mathbf{1})(E_2, \mu_2, \mathbf{1}) \cdots$ let $id(\sigma)$ be the folded allocation sequence

$$id(\sigma) = (E_0, \mu_0, \mathbf{1})id_0(E_1, \mu_1, \mathbf{1})id_1(E_2, \mu_2, \mathbf{1})id_2 \cdots$$

where $(E_i, \mu_i, \mathbf{1}) \xrightarrow{id_i} (E_{i+1}, \mu_{i+1}, \mathbf{1})$. The two sequences satisfy the same set of $\mathcal{N}\mathcal{A}\ell\text{TL}$ formulae: one on the unfolded semantics of $\mathcal{N}\mathcal{A}\ell\text{TL} \models_u$ and the other on the folded version, i.e., \models_f :

Proposition 5.4.10. For any $\mathcal{N}\mathcal{A}\ell\text{TL}$ formula ϕ we have $\sigma, N, \theta \models_u \phi$ iff $id(\sigma), N, \theta \models_f \phi$.

Proposition 5.4.11. For every ABA \mathcal{A} there exists a HABA \mathcal{H} such that given a $\mathcal{N}\mathcal{A}\ell\text{TL}$ formula ϕ , we have ϕ is \mathcal{A} -valid if and only if ϕ is \mathcal{H} -valid.

⁴It will also be useful for describing reallocations in the transitions of the symbolic operational semantics (cf. Section 5.6.4).

Proof. Let $\mathcal{A} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$, then we define $\mathcal{H} = \langle X, Q, E, \rightarrow', I, \mathcal{F} \rangle$ where \rightarrow' is such that

$$q \rightarrow q' \Leftrightarrow q \rightarrow_{id} q'$$

where $\gamma_q \xrightarrow{id} \gamma_{q'}$. Hence \mathcal{H} is defined as \mathcal{A} with an identity reallocation on the transitions. It is clear that

$$\mathcal{L}(\mathcal{H}) = \{(\sigma', N', \theta') \mid (\sigma', N', \theta') \cong (id(\sigma), N, \theta) \text{ and } (\sigma, N, \theta) \in \mathcal{L}(\mathcal{A})\}.$$

Therefore from Propositions 5.4.10 and 5.4.4 it follows that every \mathcal{A} -valid formula ϕ is also \mathcal{H} -valid and vice-versa. \square

We indicate the HABA \mathcal{H} defined in the proof of Proposition 5.4.11 by $id(\mathcal{A})$. By Propositions 5.4.7 and 5.4.11, in order to verify a $\mathcal{N}all\text{TL}$ -formula ϕ on an \mathcal{A} we can alternatively verify it on a HABA \mathcal{H} that simulates $id(\mathcal{A})$. This is particularly interesting when \mathcal{A} is an infinite model whereas \mathcal{H} is finite. In fact provided we have an effective method to check the validity of $\mathcal{N}all\text{TL}$ -formulae on finite HABA this can be used to extend the methodology to infinite-state systems.

Example 5.4.12. The formula $\phi \equiv \text{GF}(\forall x \forall y : x = y)$ stating that the queue will have infinitely often only one entity is \mathcal{H}' -valid because only \mathcal{H}' accept state has only one entity, and in an accepting run it is visited infinitely often. It follows that ϕ is also \mathcal{H} -valid. \square

5.5 A language for navigation

In this section we introduce a simple programming language, called \mathcal{L}_n , dealing with pointers. It is an extension of the language \mathcal{L} defined in Section 4.4. \mathcal{L}_n gives some insight into the kind of systems that can be modelled by HABA with references. Similar to \mathcal{L} , we will define the semantics of \mathcal{L}_n in terms of ABA and HABA. The first semantics is very concrete and intuitive, but infinite-state, whereas the second is symbolic and finite. In Section 5.6.6, we will study the relation between the two semantics.

5.5.1 Syntax

Let nil be a special syntactic constant. For PVAR a set of program variables with $v, v_i \in \text{PVAR}$, and such that $\text{PVAR} \cap \text{LVAR} = \emptyset$, the set of statements of the language \mathcal{L}_n is given by:

$$\begin{aligned} (p \in) \mathcal{L}_n & ::= \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k) \\ (s \in) \text{Stat} & ::= \text{new}(\alpha) \mid \text{del}(\alpha) \mid \alpha := \alpha \mid \text{skip} \mid s; s \\ & \quad \mid \text{if } b \text{ then } s \text{ else } s \text{ fi} \mid \text{while } b \text{ do } s \text{ od} \\ (\alpha \in) \text{Nexp} & ::= nil \mid v \mid \alpha.a \\ (b \in) \text{Bexp} & ::= \alpha = \alpha \mid b \vee b \mid \neg b \end{aligned}$$

A program p is thus a parallel composition of a finite number of statements preceded by the declaration of a finite number of global variables.

Informal semantics of \mathcal{L}_n . $\text{new}(\alpha)$ creates (i.e., allocates) a new entity that will be referred to by the expression α . If α is the only way to refer to entity e , say, then after the execution of $\text{new}(\alpha)$, e (being not reachable) is automatically garbage collected together with the entities reachable from e . $\text{del}(\alpha)$ destroys (i.e., deallocates) the entity associated to α , and makes α and every other pointer pointing to it undefined. The assignment $\alpha_1 := \alpha_2$ passes the reference held by α_2 to α_1 . Again, the entity α_1 was referring to might become unreferenced and therefore may be garbage collected. As assignments create aliases they introduce non-trivial side-effects. Sequential composition, while loop, skip, and conditional statement have the standard interpretation. Meaningless statements and expressions as $\text{new}(\text{nil})$, $\text{del}(\text{nil})$, $\text{nil}.a$ will be ruled out at the semantical level (cf. Section 5.6).

The differences between \mathcal{L}_n and the language \mathcal{L} (studied in Chapter 4) are essentially two:

- the possibility to use navigation expressions instead of simply program variables;
- the use of automatic garbage collection. This will become clear when we define the semantics.

Example 5.5.1. The following \mathcal{L}_n program is a classical example often reported in the literature (e.g., see [12, 94, 100]). It reverses a list originally pointed to by the variable v .

```

decl  $v, w, t$  :
 $w := \text{nil}$ ;
while  $v \neq \text{nil}$  do
   $t := w$ ;
   $w := v$ ;
   $v := v.a$ ;
   $w.a := t$ ;
od;
 $t := \text{nil}$ ;

```

□

5.5.2 Adding program variables to \mathcal{NallTL}

\mathcal{NallTL} expresses properties about logical variables LVAR. Given a program, $p \equiv \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$, in principle it is not possible to talk about v_i ($1 \leq i \leq n$) in \mathcal{NallTL} -formulae. For software verification this would represent a severe limitation. We would like to add this feature to \mathcal{NallTL} so that properties involving the (declared) program variables $\text{DeclPVar} = \{v_1, \dots, v_n\} \subseteq$

PVAR (with $v_i \neq v_j$ for $i \neq j$) of p can be naturally formulated. To model *DeclPVar*, we use a special set of entities $PV = \{e_{v_1}, \dots, e_{v_n}\} \subseteq Ent$ that are alive in every state of the allocation sequence. We assume the existence of a set of free logical variables *DeclLVar* = $\{x_{v_1}, \dots, x_{v_n}\}$ such that the interpretation function is:

$$\vartheta(x_{v_i}) = e_{v_i} \quad \forall 1 \leq i \leq n. \quad (5.6)$$

Notice that since entities in *PV* are alive in every state of the allocation sequence, the interpretation $\vartheta \upharpoonright PV$ will remain fixed.

The value of the logical variables x_{v_i} is not very interesting when writing properties about programs. Rather we are interested in the entity that a program variable v points to, that is, $\mu(\vartheta(x_{v_i}))$. Thus it is convenient to define

$$v_i \equiv x_{v_i}.a \quad \forall v_i \in DeclPVar \quad (5.7)$$

as syntactic sugar in the logic. This suffices us to express properties about program variables.

Furthermore, we would like to exclude *PV* from the set of entities considered in the domain of quantifiers. Again this is done by adding syntactic sugar. Throughout Sections 5.5 and 5.6, we consider the formula $\exists x : \phi$ as shorthand for

$$\exists x : (x \neq x_{v_1} \wedge \dots \wedge x \neq x_{v_n}) \Rightarrow \phi. \quad (5.8)$$

Example 5.5.2. The configurations (of the states) of the program in Example 5.5.1 — using entities $PV = \{e_t, e_v, e_w\}$ and the logical variables *DeclLVar* = $\{x_t, x_v, x_w\}$ to refer to them — are shown in Figure 5.11. It represents the execution of the loop manipulating a list with three elements.

The main property we want to verify for the program described in Example 5.5.1 is:

v 's list will be eventually reversed

that can be expressed in *NaℓℓTL* (as we have seen before) by:

$$\forall x : \forall y : (v \rightsquigarrow x \wedge x.a = y) \Rightarrow \text{FG}(y.a = x).$$

Moreover, another plausible property could be:

all the elements in v 's list will be eventually contained in w 's list

which is expressed by:

$$\forall x : (v \rightsquigarrow x) \Rightarrow \text{FG}(w \rightsquigarrow x).$$

□

5.6 Operational semantics

In this section, we describe how ABAs and HABAs can be used to give a semantics to programs in our example programming language \mathcal{L}_n .

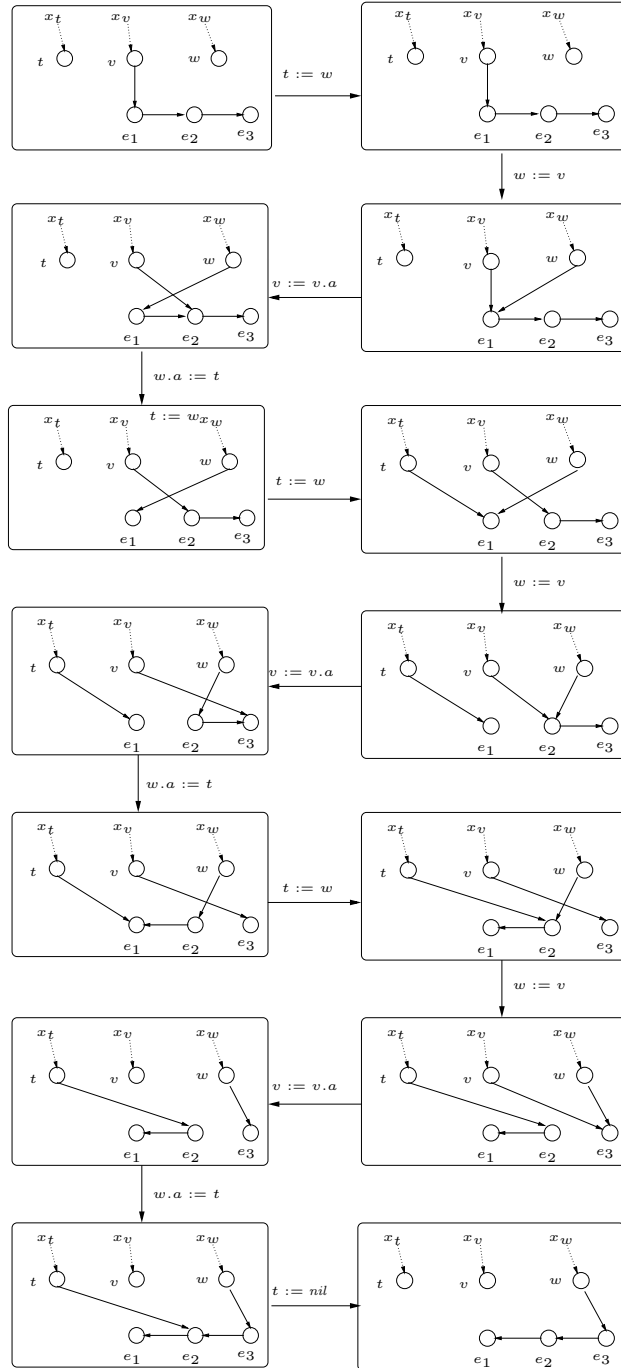


Figure 5.11: Execution of the program in Example 5.5.1

5.6.1 Preliminary terminology, assumptions and results

As discussed in Section 5.5.2, program variables can be modelled by a set of special entities that are alive in every state. Therefore, given a program $p \equiv \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$ declaring the set of program variables $\text{DeclPVar} = \{v_1, \dots, v_n\}$, from now let $\text{DeclLVar} = \{x_{v_1}, \dots, x_{v_n}\}$ and $PV = \{e_{v_1}, \dots, e_{v_n}\}$. Since p is globally given, also DeclLVar and PV will be globally given. The introduction of PV entails some constraints on some definitions already given. In particular, we give the following:

Definition 5.6.1 (well-formed configuration). A configuration (E, μ, \mathcal{C}) with $PV \subseteq E$ is *PV-well-formed* if

$$\text{cod}(\mu) \cap PV = \emptyset \quad (5.9)$$

$$\mathcal{C} \upharpoonright PV = \mathbf{1}_{PV}. \quad (5.10)$$

Entities representing program variables cannot be referred to, and are concrete. In the rest of this section we consider only well-formed configurations.

Assumptions. For two well-formed configurations γ_1 and γ_2 we consider throughout Section 5.6 only morphisms as well as reallocations that satisfy the following conditions:

$$\gamma_1 \xrightarrow{h} \gamma_2 \Rightarrow h \upharpoonright PV = \text{id}_{PV} \quad (5.11)$$

$$\gamma_1 \xrightarrow{\lambda} \gamma_2 \Rightarrow \forall e \in PV : \lambda(e, e) = 1 \quad (5.12)$$

Conditions (5.11) and (5.12) force the correspondence of the program variables in configurations related by morphisms or reallocations. These constraints seem to be rather natural because of the very special purpose assigned to PV (i.e., modelling p program variables). In fact, it is convenient — for example, from state to state of the computation — to have every program variable re-mapped on itself. Finally, we can see that (5.12) not only forces reallocations to map $e_v \in PV$ onto itself, but in combination with (5.10), ensures also that e_v is the *only* entity that can be reallocated to itself.

Preliminary notation. Let μ^* denote the reflexive and transitive closure of the relation $\{(e, \mu(e)) \mid e, \mu(e) \in \text{Ent}\}$ induced by the function μ . For configuration γ , and $e \in E_\gamma$, $\mu_\gamma^*(e)$ is the set of entities in the *remainder* of the chain starting with e . Note that $e \in \mu_\gamma^*(e)$. Moreover, let

$$\langle \gamma \rangle_{PV} = (\mu_\gamma^*(PV), \mu_\gamma \upharpoonright \mu_\gamma^*(PV), \mathcal{C}_\gamma \upharpoonright \mu_\gamma^*(PV))$$

be the configuration obtained from γ after garbage collection, i.e., having as set of entities those reachable via μ_γ by some program variable in PV . γ is called *PV-reachable* if $E_{\langle \gamma \rangle_{PV}} = E_\gamma$.

A special class of morphisms that will turn out to be useful later on are characterised by the following:

Definition 5.6.2.

- A morphism $h : \gamma \succrightarrow \gamma'$ is *contractive*, denoted $h \downarrow$, if $|h^{-1}(e)| > 1$ for some $e \in E_{\gamma'}$;
- the *shrink factor* of a morphism $h : \gamma \succrightarrow \gamma'$, is $\max \{|h^{-1}(e)| \mid e \in E_{\gamma'}\}$.

We write $h \downarrow^C$ if the shrink factor of h is at most C . Non-contractive morphisms have shrink factor 1. A contractive morphism abstracts a chain of entities into a multiple or an unbounded entity. Note that contractive morphisms correspond to non-injective morphisms. We introduce the term “contractive” because it closely resembles the idea that the morphism collapses several entities into one, thus providing a more compact view of the configuration. A straightforward fact is given by the following observation.

Proposition 5.6.3. Let $h : \gamma \succrightarrow \gamma'$. If h is non-contractive then h is an isomorphism.

Proof. It is straightforward to see that h is bijective. In fact, every morphism is surjective by definition and h is also injective because it is not contractive. Then, let $h' : \gamma' \succrightarrow \gamma$ such that $h'(e) = h^{-1}(e)$. h' is well-defined since h is bijective and it can be proved to be a morphism. Moreover, by construction we have $h' \circ h = id_{\gamma}$. Thus, by definition h is an isomorphism. \square

Definition 5.6.4. For a configuration (E, μ, \mathcal{C}) and entities $e, e' \in E$, if there exists $n \in \mathbb{N}$ such that $e' = \mu^n(e)$, the distance $d(e, e') = n$, otherwise $d(e, e') = \perp$.

Definition 5.6.5. (L -safety) Let $L > 0$. A configuration (E, μ, \mathcal{C}) is *L -safe* if

$$\forall e \in PV : (\forall e' : d(e, e') \leq L \Rightarrow \mathcal{C}(e') = 1).$$

If a configuration is not L -safe it is called *L -unsafe*.

Example 5.6.6. The configuration depicted in Figure 5.12 is 2-safe, since every entity e_{v_1}, \dots, e_{v_6} (representing a program variable) has a distance of at least two from the unbounded entity e . The configuration is not 3-safe, since $d(e_{v_6}, e) = 3$. \square

The following result says that isomorphic configurations have the same safety.

Proposition 5.6.7. For all $L > 0$: $\gamma_1 \cong \gamma_2 \Rightarrow (\gamma_1 \text{ } L\text{-safe} \Leftrightarrow \gamma_2 \text{ } L\text{-safe})$.

Proof. Straightforward since isomorphic configurations are equal up to renaming of entities. \square

Again we should pay some attention on the distinction between a HABA state q and its associated configuration γ when it comes to terminology and notation. In particular, the concepts just introduced for configurations, like

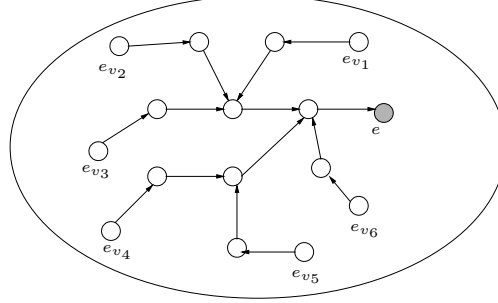


Figure 5.12: A 2-safe configuration.

well-formedness, *PV*-reachability and *L*-safeness are transferred to states in a straightforward manner, saying that a state q is respectively well-formed, *PV*-reachable, and *L*-safe if its configuration γ_q is so.

In the following, for the concrete and symbolic semantics we will guarantee that every state q is *L*-safe, and *PV*-reachable as well as well-formed.

5.6.2 Concrete semantics

A concrete semantics of \mathcal{L}_n is given in terms of ABA. Let Par denotes the set of compound statements, i.e., $r(\in Par) ::= s \mid r \parallel s$.

Definition 5.6.8 (Concrete automaton \mathcal{A}_p). The concrete semantics of $p = \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$ is the ABA $\mathcal{A}_p = \langle X_p, Q, E, \rightarrow, I, \mathcal{F} \rangle$ where

- $X_p = \{x_{v_1}, \dots, x_{v_n}\}$;
- $Q \subseteq (Par \times \text{CONF}) \cup \{\text{error}\}$, where for state $(r, \gamma) \in Q$, r is the compound statement to be executed, and γ is a *PV*-well-formed and *PV*-reachable configuration.
- $E(r, \gamma) = \gamma$ and $E(\text{error}) = (\emptyset, \emptyset, \emptyset)$;
- $\rightarrow \subseteq Q \times Q$ is the smallest relation satisfying the rules in Table 5.3;
- $\text{dom}(I) = \{(s_1 \parallel \dots \parallel s_k, PV, \emptyset, \mathbf{1}_{PV})\}$ and $I(s_1 \parallel \dots \parallel s_k, PV, \emptyset, \mathbf{1}_{PV}) = (\emptyset, \vartheta)$ where $\vartheta(x_{v_i}) = e_{v_i}$ ($1 \leq i \leq n$);
- let

$$\begin{aligned} \widehat{F}_i &= \{(s'_1 \parallel \dots \parallel s'_k, \gamma) \in Q \mid s'_i = \text{skip} \vee s'_i = \text{while } b \text{ do } s \text{ od}; s''\} \\ \widetilde{F}_i &= \{(s'_1 \parallel \dots \parallel s'_k, \gamma) \in Q \mid s'_i = \text{skip} \vee s'_i = s; \text{while } b \text{ do } s \text{ od}; s''\} \end{aligned}$$

then $\mathcal{F} = \{\widehat{F}_i \cup \{\text{error}\} \mid 0 < i \leq k\} \cup \{\widetilde{F}_i \cup \{\text{error}\} \mid 0 < i \leq k\}$.

A few remarks are in order. Every configuration γ in a state has 1_γ because \mathcal{A}_p is an ABA (cf. Definition 5.3.10). There is a special error state resulting from an attempt to evaluate an expression $\alpha.a$, where α does not denote any entity (illegal statement). The set of logical variables only contains those that are used to encode the program variables declared in the program p . \mathcal{A}_p has a single initial state $s_1 \parallel \dots \parallel s_k$. The entities that are initially alive are those used for modelling program variables. ϑ gives the standard interpretation for variables in X_p according to our initial assumptions (cf. Section 5.5.2). The set of accept states for the i -th sequential component consists of all states in which the component has either terminated ($s_i = \text{skip}$) or is processing a loop (which could be infinite) or is the error state.

Using the interpretation of navigation expressions defined in Section 5.2.1 and ϑ given by the initial state, we define the semantics of the boolean expressions used in conditional statements.

Definition 5.6.9. The semantics of boolean expressions is the function $\mathcal{V} : (\text{Bexp} \times \text{CONF}) \rightarrow \mathbb{B}$ given by (by definition $(\perp = \perp) = \text{tt}$)

$$\begin{aligned} \mathcal{V}(\alpha_1 = \alpha_2, \gamma) &= \begin{cases} \text{tt} & \text{if } \llbracket \alpha_1 \rrbracket_{\mu_\gamma, \vartheta} = \llbracket \alpha_2 \rrbracket_{\mu_\gamma, \vartheta} \\ \text{ff} & \text{otherwise} \end{cases} \\ \mathcal{V}(b_1 \vee b_2, \gamma) &= \mathcal{V}(b_1, \gamma) \vee \mathcal{V}(b_2, \gamma) \\ \mathcal{V}(\neg b, \gamma) &= \neg \mathcal{V}(b, \gamma). \end{aligned}$$

Manipulating configurations. For the definition of the concrete and symbolic operational rules we rely on three operations that are meant to perform an update on the configuration according to the statement (**new**, **del**, or **assignment**) that is executed by the rule.

The first operation, $\text{add}(\gamma, \alpha)$ adds to the configuration γ a fresh entity, and assigns the reference to the entity denoted by the expression α . We assume w.l.o.g. that the set Ent is totally ordered; this is convenient for selecting the fresh entity in a deterministic way, in fact we can take the first one not used in γ , i.e., $\min(\text{Ent} \setminus E_\gamma)$. The resulting configuration is composed only by PV -reachable entities, i.e., garbage collection is applied at this stage. Formally, the function $\text{add} : \text{CONF} \times \Pi \rightarrow \text{CONF}$ is given by:

$$\text{add}(\gamma, \alpha) = \langle E_\gamma \cup \{e\}, \mu_\gamma \{e / \llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta}\}, \mathcal{C}_\gamma \{1/e\} \rangle_{PV} \quad (5.13)$$

where $e = \min(\text{Ent} \setminus E_\gamma)$.

The operation $\text{cancel}(\gamma, \alpha)$ deletes from the configuration γ the entity denoted by α . $\text{cancel} : \text{CONF} \times \Pi \rightarrow \text{CONF}$ is given by

$$\text{cancel}(\gamma, \alpha) = \langle E_\gamma \setminus \{\llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta}\}, \mu_\gamma \circ \psi, \mathcal{C}_\gamma \upharpoonright (E_\gamma \setminus \{\llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta}\}) \rangle_{PV} \quad (5.14)$$

where $\psi : E_\gamma \rightarrow E_\gamma^\perp$ is defined as

$$\psi(e) = \begin{cases} \perp & \text{if } e \in \mu_\gamma^{-1}(\llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta}) \cup \llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta} \\ e & \text{otherwise} \end{cases}$$

In the resulting configuration, every pointer to $\llbracket \alpha \rrbracket_{\mu_\gamma, \vartheta}$ are set to \perp by ψ and the domain of the cardinality function is restricted to the remaining set of entities.

Finally, the last operation $modify(\gamma, \alpha_1, \alpha_2)$ performs an update on γ 's pointer structure so that the entity denoted by α_1 will point to $\llbracket \alpha_2 \rrbracket_{\mu_\gamma, \vartheta}$. This will be used in the assignment rules. As usual, the resulting configuration contains only reachable entities. The function $modify : \text{CONF} \times \Pi \rightarrow \text{CONF}$ is given by

$$modify(\gamma, \alpha_1, \alpha_2) = \langle E_\gamma, \mu_\gamma \{ \llbracket \alpha_2 \rrbracket_{\mu_\gamma, \vartheta} / \llbracket \alpha_1 \rrbracket_{\mu_\gamma, \vartheta} \}, \mathcal{C}_\gamma \rangle_{PV} \quad (5.15)$$

Concrete operational rules. Transitions for the basic statements follows the general pattern:

$$\frac{}{r, \gamma \rightarrow \text{skip}, \gamma'}$$

The configuration of the target state of a transition has a transformation of the configuration of the source state, and has only reachable entities. The transformation is carried out by one of the operations (5.13), (5.14), and (5.15) defined above. If garbage is produced by a transition it is immediately collected and removed by the application of the operation itself. Furthermore, recall that a program variable v is syntactic sugar for $x_v.a$. Therefore every navigation expression occurring in a statement is of the form $\alpha.a$ (or nil). Finally, we write $\llbracket \alpha \rrbracket$ as a shorthand for $\llbracket \alpha \rrbracket_{\mu, \vartheta}$. We briefly comment on the rules contained in Table 5.3.

- **Assignment.** Trying to perform an assignment $\alpha_1.a := \alpha_2$ results — by the application of rule (ASGN_{error-c}) — in a run-time error if α_1 is a null pointer. Otherwise, rule (ASGN-c) applies⁵. By the application of $modify(\gamma, \alpha_1, \alpha_2)$, the execution of the assignment corresponds to updating the outgoing reference $\mu(\llbracket \alpha_1 \rrbracket)$ to the entity denoted by α_2 (e.g. see a transition resulting from an assignment in Figure 5.11). After the execution, the assignment statement is replaced by **skip** that is either consumed in the context of a sequential composition rule or is blocked. This general pattern is also followed by rules (NEW-c) and (DEL-c).
- **Creation.** Unless α is an illegal expression — in which case (NEW_{error-c}) applies producing a run-time error — the reference of the first (fresh) entity e available from Ent according to the total order is assigned (by $add(\gamma, \alpha)$) to the outgoing reference of the entity denoted by α , cf. (5.13).
- **Deletion.** By (DEL-c), the entity denoted by $\alpha.a$ is deallocated and every reference to this entity is cancelled according to the definition of

⁵Note that if α_2 is of the form $\alpha_3.a$ where $\llbracket \alpha_3 \rrbracket_{\mu_\gamma, \vartheta} = \perp$ the assignment $\alpha_1.a := \alpha_2$ is treated by the current semantics as $\alpha_1.a := nil$. Another possible choice in the design of the semantics could be to treat such statement as “illegal”, and therefore, by the operational rule give a run-time error by making a transition to the **error** state. The conversion of the current approach to the other is straightforward.

(NEW _{error-c})	$\frac{\llbracket \alpha \rrbracket = \perp}{\text{new}(\alpha.a), \gamma \rightarrow \text{error}}$
(NEW-c)	$\frac{\llbracket \alpha \rrbracket \neq \perp}{\text{new}(\alpha.a), \gamma \rightarrow \text{skip}, \text{add}(\gamma, \alpha)}$
(DEL _{error-c})	$\frac{\llbracket \alpha \rrbracket = \perp}{\text{del}(\alpha.a), \gamma \rightarrow \text{error}}$
(DEL-c)	$\frac{\llbracket \alpha \rrbracket \neq \perp}{\text{del}(\alpha.a), \gamma \rightarrow \text{skip}, \text{cancel}(\gamma, \alpha.a)}$
(ASGN _{error-c})	$\frac{\llbracket \alpha_1 \rrbracket = \perp}{\alpha_1.a := \alpha_2, \gamma \rightarrow \text{error}}$
(ASGN-c)	$\frac{\llbracket \alpha_1 \rrbracket \neq \perp}{\alpha_1.a := \alpha_2, \gamma \rightarrow \text{skip}, \text{modify}(\gamma, \alpha_1, \alpha_2)}$
(IF _{1-c})	$\frac{\mathcal{V}(b, \gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, \gamma \rightarrow s_1, \gamma}$
(IF _{2-c})	$\frac{\neg \mathcal{V}(b, \gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, \gamma \rightarrow s_2, \gamma}$
(WHILE-c)	$\overline{\text{while } b \text{ do } s \text{ od}, \gamma \rightarrow \text{if } b \text{ then } s; \text{ while } b \text{ do } s \text{ od else skip fi}, \gamma}$
(SEQ _{error-c})	$\frac{s_1, \gamma \rightarrow \text{error}}{s_1; s_2, \gamma \rightarrow \text{error}}$
(SEQ _{1-c})	$\frac{s_1, \gamma \rightarrow s'_1, \gamma' \wedge s'_1 \neq \text{error}}{s_1; s_2, \gamma \rightarrow s'_1; s_2, \gamma'}$
(SEQ _{2-c})	$\overline{\text{skip}; s_2, \gamma \rightarrow s_2, \gamma}$
(PAR _{error-c})	$\frac{1 \leq j \leq k \wedge s_j, \gamma \rightarrow \text{error}}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, \gamma \rightarrow \text{error}}$
(PAR _{1-c})	$\frac{1 \leq j \leq k \wedge s_j, \gamma \rightarrow s'_j, \gamma' \wedge s'_j \neq \text{error}}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, \gamma \rightarrow s_1 \parallel \dots \parallel s'_j \parallel \dots \parallel s_k, \gamma'}$
(PAR _{2-c})	$\overline{\text{skip} \parallel \dots \parallel \text{skip}, \gamma \rightarrow \text{skip} \parallel \dots \parallel \text{skip}, \gamma}$
(ERROR)	$\overline{\text{error} \rightarrow \text{error}}$

Table 5.3: Operational rules for the concrete semantics of \mathcal{L}_n .

$cancel(\gamma, \alpha.a)$. Rule (DEL_{error-c}) only applies if α dereferences a null pointer.

- **Conditional and loop.** Rules (IF_{1-c})/(IF_{2-c})/(WHILE-c) are straight-forward⁶.
- **Sequential composition.** By rules (SEQ_{1-c})/(SEQ_{2-c}) when the first statement is reduced to a skip statement, it is consumed, if it is reduced to error, $s_1; s_2$ reduces to error.
- **Parallel composition.** By (PAR_{1-c}), if one of the components of the compound statement performs a step, the whole compound statement can do so, unless an error is produced in which case the whole compound statement reduces to error. As for the language defined in Chapter 4, by (PAR_{2-c}), a self-loop in an accept state with a terminated compound statement ensures that, in a run, it is visited infinitely many times.
- **Error.** A self-loop in an error state produces accepting runs that have terminated in an abnormal way because of run-time errors due to the dereferencing of null pointers.

Proposition 5.6.10. For any $L > 0$, if $q \in Q_{\mathcal{A}_p}$ then q is L -safe, PV -reachable and well-formed.

Proof. Straightforward. In fact, any $q \in Q_{\mathcal{A}_p}$ has the unitary cardinality function therefore it is safe by definition. Furthermore, states are PV -reachable by definition. Finally, q is well-formed; in fact, (5.10) is trivially satisfied and condition (5.9) it satisfied by the initial state. Moreover, modifications of pointers in the rules of Table 5.3 are performed only by (NEW-c) and (ASGN-c). But both rules assign entities which do not belong to PV by definition. In particular, for (ASGN-c) note that $\llbracket \alpha_2.a \rrbracket$ cannot be in PV because of the syntactic structure of $\alpha_2.a$. \square

5.6.3 Canonical form for HABA states

For the definition of the symbolic semantics it is convenient to define the concept of normal form which provides a standard representation (unique up to isomorphism) for a set of safe states that may be related by morphisms. To this end we first define a notion that it somehow complementary to safeness.

Definition 5.6.11 (L -compactness). A configuration (E, μ, \mathcal{C}) is L -compact if

$$\forall e \in E : (\text{indegree}(e) > 1 \vee \exists e' \in PV : d(e', e) \leq L + 1).$$

⁶As already observed for the assignment rule, note that, b may contain expressions like $\alpha_1.a = \alpha_2.a$ where either $\llbracket \alpha_1 \rrbracket_{\mu, \gamma, \vartheta} = \perp$ or $\llbracket \alpha_2 \rrbracket_{\mu, \gamma, \vartheta} = \perp$. The function $\mathcal{V}(b, \gamma)$ return a boolean value also in this case. As above, such expression are not treated as “illegal” but as *nil*. Again, it would not be problematic to adapt this semantics to the opposite approach.

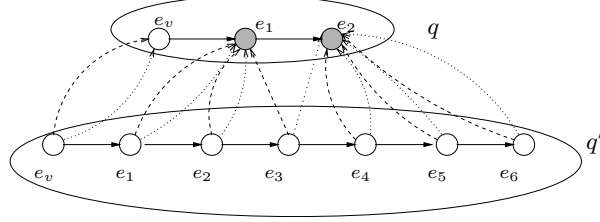


Figure 5.13: More than one morphism can relate a L -compact state to another.

As usual we extend this notion to states in the standard way: a state q is L -compact if its configuration γ_q is so.

Thus, a compact state is a state that can have pure chains only if they are distant at most $L + 1$ entities from a program variable. Given two states q and q' one of which is L -compact, in general there might exist more than one morphism relating them. This is normally the case if the state is not L -safe.

Example 5.6.12. Let $L = 2$ and the global constant $M = 1$. Let (cf. Figure 5.13):

$$\begin{aligned} q &= (\{e_v, e_1, e_2\}, \{(e_v, e_1), (e_1, e_2)\}, \{(e_v, 1), (e_1, *), (e_2, *)\}) \\ q' &= (\{e_v\} \cup \{e_i \mid 1 \leq i \leq 6\}, \\ &\quad \{(e_v, e_1)\} \cup \{(e_i, e_{i+1}) \mid 1 \leq i \leq 5\}, \\ &\quad \{(e_v, 1)\} \cup \{(e_i, 1) \mid 1 \leq i \leq 6\}). \end{aligned}$$

State q is L -compact (but not L -safe), however, there exists more than one morphism between q' and q . In fact let $h_1 : q' \rightarrow q$ and $h_2 : q' \rightarrow q$ defined as:

$$\begin{aligned} h_1(e_i) &= \begin{cases} e_1 & \text{if } 1 \leq i \leq 2 \\ e_2 & \text{otherwise} \end{cases} \\ h_2(e_i) &= \begin{cases} e_1 & \text{if } 1 \leq i < 4 \\ e_2 & \text{otherwise.} \end{cases} \end{aligned}$$

Hence, h_1 and h_2 distribute entities of q' onto entities of q in different ways. Figure 5.13 gives a pictorial representation of this situation. h_1 is represented by dashed lines whereas h_2 by dotted ones. \square

We now define the notion of canonical form.

Definition 5.6.13 (canonical form). A configuration γ is L -canonical (or in L -normal form) if

- γ is L -safe;
- γ is L -compact.

A configuration in canonical form enjoys several properties.

Proposition 5.6.14. If a configuration γ is L -canonical then:

- a) γ is PV -reachable;
- b) for every configuration γ' , if there exists a morphism $h : \gamma \succ \gamma'$ then either $\gamma \cong \gamma'$ or γ' is L -unsafe.

Proof. See Appendix B.2. □

The previous proposition shows that if a configuration is in canonical form then it is in the most compact (safe) form up to isomorphism. In fact, if compacted further, it would be unsafe.

Proposition 5.6.15. Let γ_1 and γ_2 be a PV -reachable and a L -canonical configurations, respectively. If $h_1 : \gamma_1 \succ \gamma_2$ and $h_2 : \gamma_1 \succ \gamma_2$ then $h_1 = h_2$.

Proof. See Appendix B.2. □

Theorem 5.6.16 (Existence of the canonical form). For every L -safe and PV -reachable configuration γ there exists an L -canonical configuration γ' and a unique morphism $h : \gamma \succ \gamma'$.

Proof. See Appendix B.2. □

Corollary 5.6.17. The canonical form of L -safe configuration γ is unique (up to isomorphism).

We call γ' (of the previous theorem) the canonical form of γ and indicate it by $\text{cf}(\gamma)$. We write $h_{\text{cf}}(\gamma)$ for the unique morphism h relating γ and $\text{cf}(\gamma)$.

Safe expansions. Theorem 5.6.16 ensures the existence of a unique canonical form for safe configuration. However, in the definition of the assignment rule we need to deal with unsafe configurations. We have also seen that an unsafe configuration can be related to a safe one by more than one morphism. The following notion defines a finite set of pairs (γ', h) where, the first component γ' is an L -safe configuration representing the same topological structure of a possibly unsafe configuration γ ; the second component h is the morphism relating γ' and γ .

Definition 5.6.18. The set of *safe expansions* of a configuration γ is

$$\text{SExp}(\gamma) = \{(\gamma', h) \mid \gamma' \text{ is } L\text{-safe and } h \downarrow^{L+1} : \gamma' \succ \gamma\}.$$

In $\text{SExp}(\gamma)$ all configurations that are included are related to γ by a contractive morphism (cf. Definition 5.6.2) with shrink factor at most $L + 1$. This bound ensures that $\text{SExp}(\gamma)$ is finite (up to isomorphism). Moreover, we will see that this is enough to include in $\text{SExp}(\gamma)$ all the necessary configurations needed for the definition of the assignment rule in Section 5.6.4.

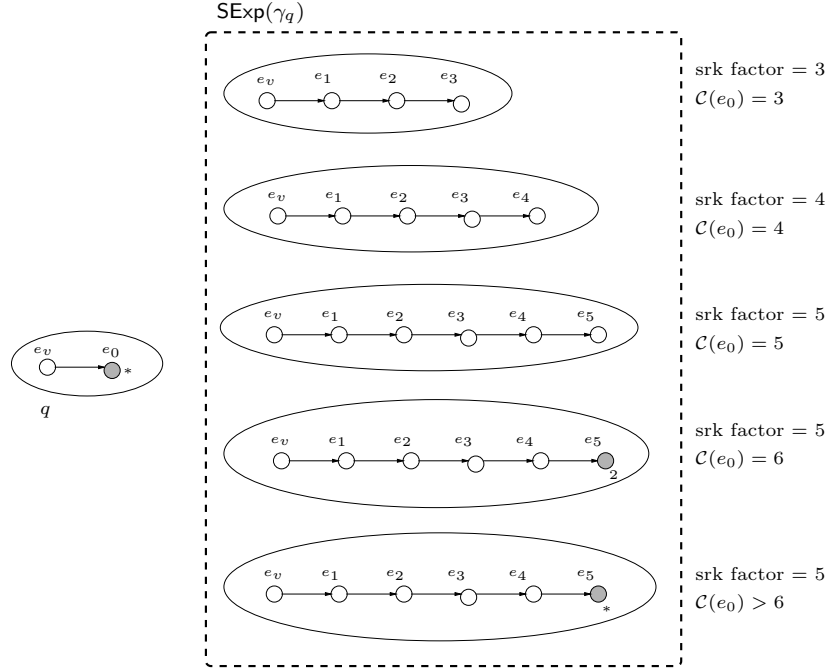


Figure 5.14: Example of safe expansions for an unsafe state (case $L = 4$, $M = 2$).

Example 5.6.19. Assume the global constant $M = 2$. Then configuration γ_q depicted in Figure 5.14 (left) is not 4-safe. The $\text{SExp}(\gamma_q)$ contains the configurations reported in the right part of figure enclosed in the dashed box. For each of them the shrink factor is indicated together with the corresponding cardinality of e_0 that the single (configuration of the) state represents. \square

Combining reallocations and morphisms. Combining reallocations with morphisms in general does not result neither in a morphism nor in a reallocation. However, in some special cases the combination of an *id* reallocation followed by morphisms defines a reallocation. For the definition of the symbolic operational rules we are interested in some of these special cases.

The following proposition proves that it is possible to complete the diagram reported in Figure 5.15 by a reallocation λ .

Proposition 5.6.20. Let γ and γ''' be two L -canonical states. If $\gamma \xrightarrow{id} \gamma' \xleftarrow{h_1} \gamma'' \xrightarrow{h_2} \gamma'''$ such that

- (a) $\forall e \in E_{\gamma''} : id^{-1}(h_1 \circ h_2^{-1}(e)) \subseteq E_{\gamma}^{\perp}$ is a chain and
- (b) $\mathcal{C}_{\gamma''}(e) = * \Rightarrow \perp \notin id^{-1}(h_1 \circ h_2^{-1}(e))$.

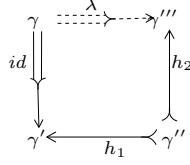


Figure 5.15: Diagram of Proposition 5.6.20

Then $\gamma \xrightarrow{\lambda} \gamma'''$ where:

$$\begin{aligned} \lambda = & \{(e_1, e_2, \mathcal{C}_{\gamma''}(h_1^{-1}(e_1) \cap h_2^{-1}(e_2))) \mid e_1 \in E_\gamma\} \cup \\ & \{(e, \perp, \mathcal{C}_\gamma(e)) \mid e \in E_\gamma \setminus E_{\gamma'}\} \cup \\ & \{(\perp, e, \mathcal{C}_{\gamma'''}(e)) \mid e \in E_{\gamma'''} \wedge h_1 \circ h_2^{-1}(e) \cap E_\gamma = \emptyset\}. \end{aligned}$$

Proof. See Appendix B.2. \square

We write $h_2 \circ h_1^{-1} \circ (\gamma \xrightarrow{id} \gamma')$ to indicate the reallocation λ defined in the previous proposition. In some cases we will use a simplified version of $h_2 \circ h_1^{-1} \circ (\gamma \xrightarrow{id} \gamma')$ that corresponds to the special case $\gamma \xrightarrow{id} \gamma' \xrightarrow{h} \gamma''$. In this case we define:

$$\begin{aligned} h \circ (\gamma \xrightarrow{id} \gamma') = & \{(e, h(e), id(e, e)) \mid e \in E_\gamma \cap E_{\gamma'}\} \cup \quad (5.16) \\ & \{(e, \perp, \mathcal{C}_\gamma(e)) \mid e \in E_\gamma \setminus E_{\gamma'}\} \cup \\ & \{(\perp, h(e), \mathcal{C}_{\gamma''}(e)) \mid e \in E_{\gamma'} \setminus E_\gamma\}. \end{aligned}$$

As $h_2 \circ h_1^{-1} \circ (\gamma \xrightarrow{id} \gamma')$ also $h \circ (\gamma \xrightarrow{id} \gamma')$ defines a reallocation provided that the same hypothesis of Proposition 5.6.20 are valid. This is stated in the next remark.

Corollary 5.6.21. Let γ, γ' be two L -canonical configurations. If $\gamma \xrightarrow{id} \gamma' \xrightarrow{h} \gamma''$ such that

- (a) $\forall e \in E_{\gamma''} : id^{-1}(h^{-1}(e)) \subseteq E_\gamma^\perp$ is a chain and
- (b) $\mathcal{C}_{\gamma''}(e) = * \Rightarrow \perp \notin id^{-1}(h^{-1}(e))$.

then $\gamma \xrightarrow{\lambda} \gamma''$ where $h \circ (\gamma \xrightarrow{id} \gamma')$.

Proof. Straightforward. In fact, by Proposition 5.6.20 we have $\gamma \xrightarrow{\lambda} \gamma''$ because $\gamma \xrightarrow{id} \gamma' \xrightarrow{id} \gamma' \xrightarrow{h} \gamma''$ where λ , defined in the proposition, is precisely $h \circ (\gamma \xrightarrow{id} \gamma')$. \square

The composition of *id* reallocations with morphisms is interesting because the operation *add*, *cancel*, *modify* — that we will use for the definition of the symbolic rules — transform a configuration γ into a configuration γ' which is related to the former by an *id* reallocation.

Lemma 5.6.22. Let γ be a configuration, and α, α' be navigation expressions. Then:

1. $\gamma \xrightarrow{id} add(\gamma, \alpha)$
2. $\gamma \xrightarrow{id} cancel(\gamma, \alpha)$
3. $\gamma \xrightarrow{id} modify(\gamma, \alpha, \alpha')$.

Proof. Straightforward. In fact, for the three operation it is easy to define the corresponding identity reallocations.

1. For $e \in E_\gamma, e' \in E_{add(\gamma, \alpha)}$, let $\lambda : \gamma \Rightarrow E_{add(\gamma, \alpha)}$ defined as:

$$\begin{aligned} \lambda(e, e') &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e = e' \\ 0 & \text{otherwise.} \end{cases} \\ \lambda(\perp, e') &= \begin{cases} 1 & \text{if } e' = \min(Ent \setminus E_\gamma) \\ 0 & \text{otherwise.} \end{cases} \\ \lambda(e, \perp) &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } E_\gamma \setminus E_{add(\gamma, \alpha)} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

It is possible to verify that λ defines a reallocation according to Definition 5.3.12 and moreover it satisfies the Definition 5.4.9, therefore it is an identity reallocation.

2. Similar to the previous case. For $e \in E_\gamma, e' \in E_{cancel(\gamma, \alpha)}$, let $\lambda' : \gamma \Rightarrow E_{cancel(\gamma, \alpha)}$ defined as:

$$\begin{aligned} \lambda'(e, e') &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e = e' \\ 0 & \text{otherwise.} \end{cases} \\ \lambda'(e, \perp) &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } E_\gamma \setminus E_{cancel(\gamma, \alpha)} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Like the *add* case, it can be verified that λ' is an identity reallocation.

3. Similar to the *cancel* case. For $e \in E_\gamma, e' \in E_{modify(\gamma, \alpha, \alpha')}$, let $\lambda'' : \gamma \Rightarrow E_{modify(\gamma, \alpha, \alpha')}$ defined as:

$$\begin{aligned} \lambda''(e, e') &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e = e' \\ 0 & \text{otherwise.} \end{cases} \\ \lambda''(e, \perp) &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } E_\gamma \setminus E_{modify(\gamma, \alpha, \alpha')} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Again, like the *add* case, it can be verified that λ'' is an identity reallocation.

□

5.6.4 Symbolic semantics

The concrete semantics of \mathcal{L}_n defined in the previous section is rather intuitive, but results in an infinite state space. For example, this can happen, in our case, because during the computation the mechanism of entity creation is invoked infinitely often without equally many deletions. The infinite-state explosion should not be a surprise since it is a typical phenomenon of concrete semantics of many formalisms meant to model complex systems. Recall, for instance, that the language \mathcal{L} studied in Section 4.4 has an infinite state space even for very simple programs. In this section — in a similar way as in Section 4.4.3 — we define a symbolic semantics of \mathcal{L}_n in terms of HABA with references. We will exploit the notion of canonical form for states introduced in Section 5.6.3 and we will observe that this helps to achieve a finite-state semantics for every program of the language.

Assumptions. In the definition of the symbolic semantics, we assume to know the longest navigation expression occurring in the program p . This number, denoted by L_p , can be determined statically by a syntactic analysis over p . In any state of the program, L_p provides us with an upper bound (in terms of distance from a program variable) to the most distant entity accessed by a statement of p . For example, if $x.a^4$ is the longest occurring reference expression in p then $L_p = 5$. Thus we define:

$$L_p = \max \{n \mid v.a^n \text{ occurs in } p\} + 1. \quad (5.17)$$

Informal idea of the symbolic model. In the symbolic semantics, we exploit unbounded entities in order to keep the model finite⁷. The price to pay for the resulting finitary treatment of the semantics is an increase in the complexity of the machinery needed for the definition of the transition system. In particular, the difficulties inherent to the employment of unbounded entities are two:

- It may be unclear which entities are involved in the execution of a statement.
- it may be unclear which is the state resulting after the execution of a statement.

The direct consequence is the unavoidable introduction of *non-determinism* in the model. We would like to exploit as much information as possible in order to minimise the amount of this nondeterminism. For this reason, the symbolic semantics applies the following strategy:

whenever an unbounded entity appears in a state we make sure that this is preceded by a chain of length at least L_p of concrete entities.

⁷Here unbounded entities generalise the concept of black-hole used in Chapter 4.

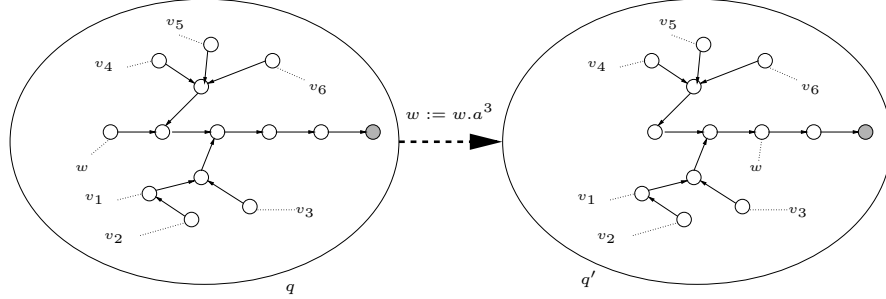


Figure 5.16: From the 3-safe state q the assignment $w := w.a^3$ produces the 3-unsafe state q' .

In other words, any state of the symbolic semantics is enforced to be L_p -safe (cf. Definition 5.6.5). This implies, by assumption on L_p (see (5.17)), that in every state we can precisely determine the *concrete* entity denoted by any navigation expression occurring in `new` or `del` statements. The major benefit is that for these statements the operational rules are deterministic and easy to define. The only source of *nondeterminism* in the symbolic semantics may be the assignment statement. In fact, although the entities denoted by both the expressions on the left-hand side and on the right-hand side of the assignment are uniquely identified (by L_p -safeness assumption), the effect of an assignment may result in an unsafe state. In general, this happens when variables change their reference to some entities that are closer to an unbounded entity. For example, the state q depicted in Figure 5.16 is 3-safe. However, the assignment $w := w.a.a.a$ produces the 3-unsafe state q' . Hence, since we admit only safe states, it is not possible to take as a result of an assignment q' that merely results from the manipulation of pointers dictated by the assignment. We need to consider “safe versions” of q' , i.e., more concrete states that represent the same pointer structure. In terms of Definition 5.6.18, this means to consider states in $\text{SExp}(q')$. In general there can be more than one possibility and therefore a nondeterministic step is unavoidable.

Definition 5.6.23 (Symbolic automaton \mathcal{H}_p). The symbolic semantics of $p = \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k)$ is the HABA $\mathcal{H}_p = \langle X_p, Q, E, \rightarrow, I, \mathcal{F} \rangle$ where

- $Q \subseteq (\text{Par} \times \text{CONF}) \cup \{\text{error}\}$, i.e., a state (r, γ) consists of a compound statement r , and a PV -well-formed and L_p -canonical configuration γ .
- $\rightarrow \subseteq Q \times (\text{Ent} \times \text{Ent} \rightarrow \mathbb{M}^*) \times Q$, is the smallest relation defined by the rules in Table 5.6.4; see Section 5.6.5 for concepts and notation.

and X_p , E , I and \mathcal{F} are defined in the same way as Definition 5.6.8.

The definition of \mathcal{H}_p resembles the one given for \mathcal{A}_p in many aspects, in particular concerning the initial state and the accept state, and the reader is referred to Definition 5.6.8 for comments on these components.

5.6.5 Symbolic operational rules

The rules of the symbolic operational semantics are heavily based on the central notion of canonical form defined in Section 5.6.3. They follow the general pattern:

$$\frac{}{r, \gamma \rightarrow_\lambda \text{skip}, \text{cf}(\gamma')}$$

The idea is that the target state is the canonical form of a certain manipulation of the source state performed — as for the concrete semantics — by one of the operation (5.13), (5.14) or (5.15). The canonical form must be enforced since manipulating pointers in the source state (in particular, during an assignment) may indeed result in a target state that is not L_p -canonical (see above). Observe that since the target state is PV -reachable⁸, garbage collection is applied at every transition. Hence, apart from the canonical form, any symbolic rule resembles the corresponding concrete rule. The only exception is (ASGN-s) (that we comment below). There exists in general a relation between the reallocation λ and the morphism $h_{\text{cf}}(\gamma')$, which we abbreviate with h_{cf} , as can be observed in the rules.

In the symbolic operational rules the evaluation of a navigation expression α is done by the definition of semantics $\llbracket \alpha \rrbracket$ given in Section 5.2.1 and already used for concrete semantics. This is possible because \mathcal{H}_p has only L_p -canonical states (that are L_p -safe by definition) and therefore, the scope of any expression α is within the part of the configuration containing only concrete entities. Hence, for every state in a configuration γ_q of a state $q \in Q_{\mathcal{H}_p}$, the expression $\llbracket \alpha \rrbracket_{\mu_{\gamma_q}, \vartheta}$ is well-defined.

- **Creation.** Performing $\text{new}(\alpha.a)$ results in the canonical form of the configuration obtained by $\text{add}(\gamma, \alpha)$. Corollary 5.6.21 and Lemma 5.6.22 ensure that this is a reallocation. Performing a $\text{new}(\alpha.a)$ statement with α undefined results in a run-time error. As in the concrete semantics, this is modelled by the special state error (cf. ($\text{NEW}_{\text{error-s}}$)rule).
- **Deletion.** The rule (DEL-s) deletes the entity denoted by $\alpha.a$ according to $\text{cancel}(\gamma, \alpha.a)$.
- **Assignment** As usual, if a null pointer is dereferenced, a run-time error is produced, cf. ($\text{ASGN}_{\text{error-s}}$) rule⁹.

⁸Because of the application of the operations that manipulate the configurations.

⁹Concerning the possibility to detect a run-time error in case α_2 is an illegal expression, the same observation done for the concrete semantics holds also for the symbolic one (cf. foot note page 159).

(NEW _{error-s})	$\frac{\llbracket \alpha \rrbracket = \perp}{\text{new}(\alpha.a), \gamma \rightarrow_{\emptyset} \text{error}}$
(NEW-s)	$\frac{\llbracket \alpha \rrbracket \neq \perp}{\text{new}(\alpha.a), \gamma \rightarrow_{\lambda} \text{skip}, \text{cf}(\text{add}(\gamma, \alpha))} \quad \lambda = h_{\text{cf}} \circ (\gamma \xrightarrow{\text{id}} \text{add}(\gamma, \alpha))$
(DEL _{error-s})	$\frac{\llbracket \alpha \rrbracket = \perp}{\text{del}(\alpha.a), \gamma \rightarrow_{\emptyset} \text{error}}$
(DEL-s)	$\frac{\llbracket \alpha \rrbracket \neq \perp}{\text{del}(\alpha.a), \gamma \rightarrow_{\lambda} \text{skip}, \text{cf}(\text{cancel}(\gamma, \alpha.a))}$ where $\lambda = h_{\text{cf}} \circ (\gamma \xrightarrow{\text{id}} \text{cancel}(\gamma, \alpha.a))$
(ASGN _{error-s})	$\frac{\llbracket \alpha_1 \rrbracket = \perp}{\alpha_1.a := \alpha_2, \gamma \rightarrow_{\emptyset} \text{error}}$
(ASGN-s)	$\frac{\llbracket \alpha_1 \rrbracket \neq \perp}{\alpha_1.a := \alpha_2, \gamma \rightarrow_{\lambda} \text{skip}, \text{cf}(\gamma'')}$ where $\begin{cases} (\gamma'', h) \in \text{SExp}(\text{modify}(\gamma, \alpha_1, \alpha_2)) \\ \lambda = h_{\text{cf}} \circ h^{-1} \circ (\gamma \xrightarrow{\text{id}} \text{modify}(\gamma, \alpha_1, \alpha_2)) \end{cases}$
(IF _{1-s})	$\frac{\mathcal{V}(b, \gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, \gamma \rightarrow_{\text{id}} s_1, \gamma}$
(IF _{2-s})	$\frac{\neg \mathcal{V}(b, \gamma)}{\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ fi}, \gamma \rightarrow_{\text{id}} s_2, \gamma}$
(WHILE-s)	$\overline{\text{while } b \text{ do } s \text{ od}, \gamma \rightarrow_{\text{id}} \text{if } b \text{ then } s; \text{while } b \text{ do } s \text{ od else skip fi}, \gamma}$
(ERROR)	$\frac{}{\text{error} \rightarrow_{\emptyset} \text{error}} \quad (\text{SEQ}_{\text{error-s}}) \quad \frac{s_1, \gamma \rightarrow \text{error}}{s_1; s_2, \gamma \rightarrow_{\emptyset} \text{error}}$
(SEQ _{1-s})	$\frac{s_1, \gamma \rightarrow_{\lambda} s'_1, \gamma' \wedge s'_1 \neq \text{error}}{s_1; s_2, \gamma \rightarrow_{\lambda} s'_1; s_2, \gamma'}$
(SEQ _{2-s})	$\overline{\text{skip}; s_2, \gamma \rightarrow_{\text{id}} s_2, \gamma}$
(PAR _{error-s})	$\frac{1 \leq j \leq k \wedge s_j, \gamma \rightarrow \text{error}}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, \gamma \rightarrow_{\emptyset} \text{error}}$
(PAR _{1-s})	$\frac{1 \leq j \leq k \wedge s_j, \gamma \rightarrow_{\lambda} s'_j, \gamma' \wedge s'_j \neq \text{error}}{s_1 \parallel \dots \parallel s_j \parallel \dots \parallel s_k, \gamma \rightarrow_{\lambda} s_1 \parallel \dots \parallel s'_j \parallel \dots \parallel s_k, \gamma'}$
(PAR _{2-s})	$\overline{\text{skip} \parallel \dots \parallel \text{skip}, \gamma \rightarrow_{\text{id}} \text{skip} \parallel \dots \parallel \text{skip}, \gamma}$

Table 5.4: Operational rules for the symbolic semantics of \mathcal{L}_n .

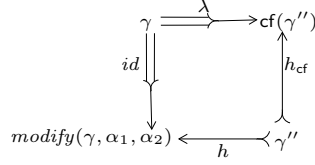


Figure 5.17: Correspondence between the source and the target state in the assignment rule.

Otherwise, (ASGN-s) applies. Informally speaking, it employs the following strategy:

1. First of all, the manipulation of pointers dictated by the assignment takes place according to $modify(\gamma, \alpha_1, \alpha_2)$.
2. If the configuration $modify(\gamma, \alpha_1, \alpha_2)$ is unsafe, we consider its safe expansion.
3. Every state having as a configuration the canonical form of a $\gamma'' \in \mathbf{SExp}(modify(\gamma, \alpha_1, \alpha_2))$ is a target state of the assignment rule.

Safety can be lost only if in the remainder of the expression considered by the assignment there exists some unbounded entity. For example, consider the case where a variable w is assigned with some entity down in the same list pointed to by w (cf. Figure 5.16).

If in the remainder of the expression we want to assign there are no unbounded (or multiple) entities, readjusting the pointers performed by $modify(\gamma, \alpha_1, \alpha_2)$ according to the assignment and taking the canonical form suffices, i.e., step 2 is not necessary.

Rule (ASGN-s) is *non-deterministic* if the set $\mathbf{SExp}(modify(\gamma, \alpha_1, \alpha_2))$ contains more than one configuration and they do not have the same canonical form. The existence of the canonical form $cf(\gamma'')$ is guaranteed by Theorem 5.6.16. The definition of the reallocation λ can be understood by Figure 5.17. Configurations γ and $modify(\gamma, \alpha_1, \alpha_2)$ are related by an identity reallocation as stated by Lemma 5.6.22. λ re-allocates entities $e \in E_\gamma$, $e' \in E_{cf(\gamma'')}$ related by the morphisms h in $\mathbf{SExp}(modify(\gamma, \alpha_1, \alpha_2))$ and h_{cf} .

Rules for conditional, while loop, sequential composition, parallel composition, and run-time error are similar to those in Table 5.3 and are listed here for completeness. For an explanation, we refer the reader to Section 5.6.2.

Example 5.6.24. Let $L = 3$ and $M = 2$. The actual transitions resulting from the assignment $w := w.a^3$ considered in Figure 5.16 are shown in Figure 5.18. $\gamma_{q_1}, \gamma_{q_2}, \gamma_{q_3}$ are the canonical form of the states in $\mathbf{SExp}(\gamma_{q'})$.

In Figure 5.19, we focus on the steps corresponding to the diagram in Figure 5.17 taken to obtain the transition $q \rightarrow_\lambda q_1$. Only the mapping of the

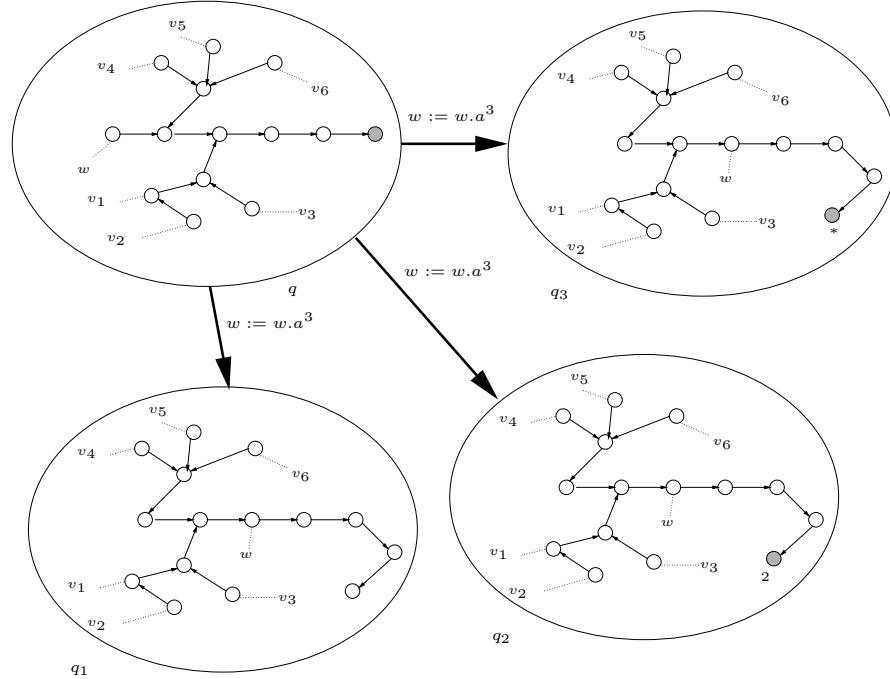


Figure 5.18: Non-deterministic step given by an unsafe assignment.

unbounded entity is drawn. The mappings for the other entities are identities. State q'' is obtained by the morphism $h \downarrow^4$. The canonical form then reduces the chain of four elements in one with three elements. \square

The next lemma states a corollary of Proposition 5.6.20.

Lemma 5.6.25. λ defined in rule (ASGN-s) is a reallocation.

Proof. See Appendix B.2. \square

Proposition 5.6.26. If $q \in Q_{\mathcal{H}_p}$ then q is L -safe, PV -reachable and well-formed.

Proof. Straightforward by the definition of $Q_{\mathcal{H}_p}$. \square

5.6.6 Relating the concrete and symbolic semantics

In this section, we study the relation between the two semantics we have defined for \mathcal{L}_n . For the language \mathcal{L} defined in Chapter 4 we found that the concrete and the symbolic semantics are actually equivalent. For the special case of \mathcal{L}_n , the correspondence between \mathcal{A}_p and \mathcal{H}_p is stated by the following:

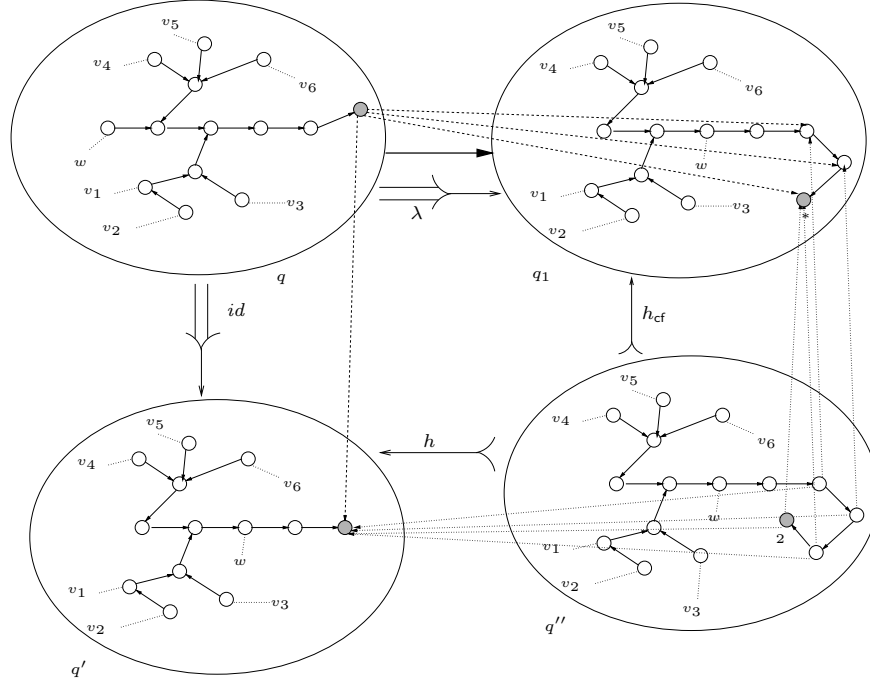


Figure 5.19: Zoom of the diagram in Fig. 5.17 applied to the assignment of Fig. 5.18.

Theorem 5.6.27. For all programs p : $id(\mathcal{A}_p) \sqsubseteq \mathcal{H}_p$.

Proof. See Appendix B.2. □

The symbolic automaton simulates the concrete one, therefore the theory and results developed in Section 5.4 can be applied here to \mathcal{A}_p and \mathcal{H}_p . In particular concerning the verification of $\mathcal{N}all$ TL properties of the program p whose semantics is given by \mathcal{A}_p and \mathcal{H}_p . In fact, as a straightforward consequence of the previous theorem we have:

Corollary 5.6.28. For every program p and every $\mathcal{N}all$ TL-formula ϕ :

$$\phi \text{ is } \mathcal{H}_p\text{-valid} \Rightarrow \phi \text{ is } \mathcal{A}_p\text{-valid.}$$

Proof. Straightforward application of Theorem 5.6.27 and Propositions 5.4.7 and 5.4.11. □

Finally, the next result gives another important step towards the development of techniques for exhaustive state space verification for HABA.

Theorem 5.6.29. For all programs p , \mathcal{H}_p is finite-state.

Proof. See Appendix B.2. □

On the refining of the model. The construction of the model \mathcal{H}_p depends on the canonical form that in turn is parametric to the number L_p . The precision of the model may be increased by tuning opportunely L_p : in fact, by increasing L_p , the corresponding canonical form of the states will be more concrete. A second parameter that can be used to change the precision of the model is M . The higher is $\mathcal{C}(\mathcal{H}_p)$ the more concrete is the model. Hence, a tool that extracts a model from p should be designed to provide the user with the capability to input L_p and M .

5.7 Model checking $\mathcal{N}allTL$

In this section, we define an algorithm for model-checking $\mathcal{N}allTL$ formulae against a HABA with references. The algorithm is based on the one defined in Section 4.5 that, in turn, extends the tableau-based method for LTL [77].

As we have done in Section 4.5, we will evaluate $\mathcal{N}allTL$ -formulae on states of a HABA by mapping the free variables of the formula to entities of the state. Again, these mappings are used to resolve all basic propositions like: the freshness predicate $x \text{ new}$; the entity equation $x.a^n = y.a^m$; and the (new) leads-to propositions $x.a^n \rightsquigarrow y.a^m$. The same obstacles encountered for $\mathcal{A}llTL$ need to be addressed here, together with new difficulties proper of $\mathcal{N}allTL$. These can be summarised as follow.

- It is not always uniquely determined whether or not an entity is fresh in a state. Arriving in the states from different transitions an entity can be new or old depending whether or not it is in the codomain of the reallocation attached to that transition.

As for $\mathcal{A}llTL$, this obstacle is dealt with by the *duplication* of states defined in Section 4.5.1.

- For variables (of the formula we want to model-check) that are mapped onto unbounded entities, propositions like entity equation or leads-to cannot be decided since it is not clear in which instances of the unbounded entity the variables are interpreted.

To deal with this difficulty, we extend the notion of valuation used for $\mathcal{A}llTL$ (cf. Definition 4.5.4). For $\mathcal{N}allTL$, we define a notion of *distance* between the interpretation of the free variables of a formula. Since variables, say x and y , can be mapped onto the same unbounded entity e , the distance needs to be sensible to the level of the “instances” of e where x and y are precisely interpreted¹⁰.

- For the formula ϕ we want to model-check, the HABA can be too abstract. In particular, the global constant M up-to which we have precise

¹⁰In this context, instances of e are synonym of the concrete entities that e abstractly represents.

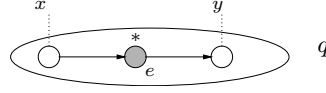


Figure 5.20: In q is the truth value of $x.a^5 = y$ ambiguous.

knowledge of the number of concrete entities an multiple entity represents can be too small with respect to the navigation expressions occurring in ϕ . When this is the case, it is not possible to decide equality propositions and the leads-to predicate.

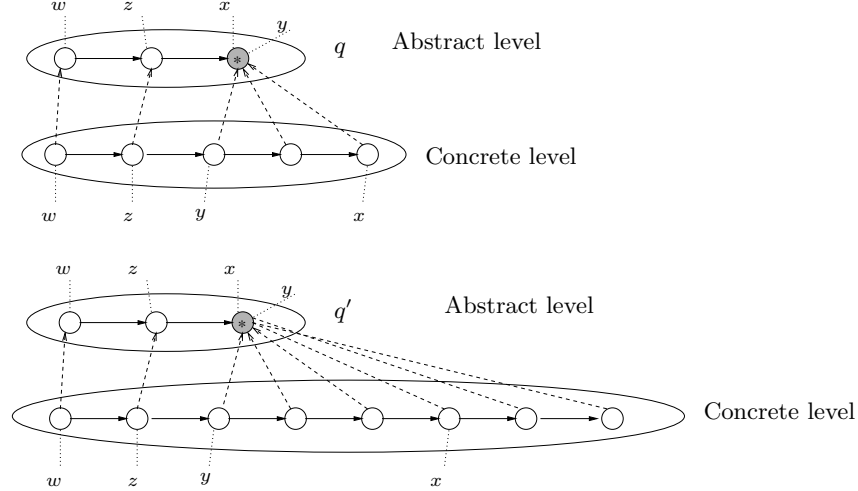
To overcome this problem typical of HABA with references and \mathcal{NallTL} , we transform the model in an equivalent one where the global constant M is raised up to a suitable upper bound (dependent from ϕ) that provides correct information for the atomic propositions in ϕ . This transformation process is called *stretching*.

5.7.1 Stretching HABA

In a HABA \mathcal{H} an entity can have precise cardinality *only* up to the global constant M . In the process of model checking a formula ϕ , the abstraction imposed by M can result to be too strong. In fact, it is necessary to find suitable assignments for ϕ 's variables in order to decide whether it cannot be satisfiable in \mathcal{H} . Thus there exists a dependency between the atomic propositions in ϕ and the constant M . In case M is too small, and therefore the representation of \mathcal{H} is too abstract to support the definition of valuations (cf. Section 5.7.6), \mathcal{H} must be unfolded, or as we say, stretched. The stretching process increases the precision of \mathcal{H} states.

Example 5.7.1. Assume $M = 2$ and consider the state q depicted in Figure 5.20. Moreover, assume we want to decide if the formula $\phi \equiv x.a^5 = y$ holds in q . The truth value of ϕ changes depending on the number of entities actually represented by the unbounded entity e . Nevertheless, since $M = 2$, e can represents any number of instances strictly greater than 2. In particular, ϕ is true only in the case, e represents precisely 4 instances, and it is false in every other case. Hence, in q , ϕ can be both true or false. By stretching the model to a sufficient extent, we can avoid this kind of ambiguities (at the cost of nondeterminism). \square

Stretching means augmenting the precision of the states by increasing the value of M . This yields for every state with an abstract configuration a set of states associated to more concrete configurations and represents the same original pointer structure (i.e., these configurations are related by morphisms). The first obvious question is, of course, how much concrete the resulting stretched HABA should be. Clearly, the risk we run is that the model must be magnified so much to become infinite which, in the context of model checking, would

Figure 5.21: Example of suitable $K(\phi)$.

mean to neutralise the complete system of abstraction built on unbounded entities. Fortunately, for a given formula ϕ we need only a bounded stretching, since after a certain point, more concrete HABAs would not provide any further useful information. For a formula ϕ this bound, written $K(\phi)$, is given by:

$$K(\phi) = \max(M + 1, \sum_{x \in fbv(\phi)} \max \{n + 1 \mid x.a^n \text{ occurs in } \phi\}). \quad (5.18)$$

where $fbv(\phi)$ is the set of free and bound variables of ϕ .

Example 5.7.2. Consider the formula $\phi_1 \equiv \exists x : \exists y : (z = x.a^3 \wedge w.a^2 = y)$, and the HABA state q depicted in Figure 5.21 (top) where a possible given interpretation of the variables in ϕ_1 is considered. In order to satisfy the formula, x and y must be interpreted as part of the unbounded entity. It is clear from the picture that we must consider the unbounded entity to represent *at least* 3 entities in order to find a suitable assignment for x and y that makes the formula true in the state. $K(\phi_1) = 9 (= 4 + 3 + 1 + 1)$ is obviously enough to decide the validity of ϕ_1 .

Now consider $\phi_2 \equiv \exists x : \exists y : (z.a^5 = x.a^2 \wedge w = y.a^2)$ and state q' (cf. Figure 5.21, bottom part). In this case because of the given interpretation of the free variable z and the offset $.a^5$, it results that the unbounded entity must represent at least 6 entities in order to find a suitable assignment for the variable that makes ϕ_2 valid in the state. This explains why we need to sum up the expression related to free variables together with those related to bounded variables. Note that $K(\phi_2) = 13 (= 6 + 3 + 3 + 1)$. \square

Definition 5.7.3 (HABA stretching). Let $\mathcal{H} = \langle X, Q, E, \rightarrow, I, \mathcal{F} \rangle$ be a HABA such that $\mathcal{C}(\mathcal{H}) = M$. The *stretching of \mathcal{H} up to \hat{M}* (with $M < \hat{M}$) is the HABA $\mathcal{H} \uparrow \hat{M} = \langle X, Q', E', \rightarrow', I', \mathcal{F}' \rangle$ where for $q \in Q$ let $S_q = \{(q, E_q, \mu_q, \mathcal{C}) \mid \text{cod}(\mathcal{C}) \subseteq \hat{M}^*, \mathcal{C}_q = \lceil \mathcal{C} \rceil_M\}$ then

- $Q' = \bigcup_{q \in Q} S_q$;
- $E'(q, \gamma) = \gamma$
- \rightarrow' is the smallest relation such that:

$$\frac{q_1 \rightarrow_\lambda q_2 \wedge \gamma_1 \xrightarrow{\lambda_s} \gamma_2}{(q_1, \gamma_1) \rightarrow'_{\lambda_s} (q_2, \gamma_2)} \quad \forall \lambda_s : \lambda = \lceil \lambda_s \rceil_M$$

- $I' = \bigcup_{q \in I} S_q$;
- $\mathcal{F}' = \{\bigcup_{q \in F} S_q \mid F \in \mathcal{F}\}$.

The automaton $\mathcal{H} \uparrow \hat{M}$ includes for each state q of \mathcal{H} the set S_q containing all the states obtained by assigning to entities with cardinality $*$, every cardinality in $\{M+1, \dots, \hat{M}\} \cup \{*\}$. The transition relation \rightarrow' is obtained from the original one by adding for every transition $q \rightarrow q'$, all possible transitions from states in S_q to state $S_{q'}$ taking care that there exists a reallocation λ_s between the configurations of the modified states. This constraint concerns the compatibility of the new cardinalities of their entities. The initial and accept states of $\mathcal{H} \uparrow \hat{M}$ are those corresponding to initial and accept states of \mathcal{H} .

Example 5.7.4. Consider the HABA depicted in Figure 5.22, such that $\mathcal{C}(\mathcal{H}) = 2$. The stretching up to 4 is shown in Figure 5.23. In this example we have:

$$\begin{aligned} S_q &= \{q_1, q_2, q_3\} \\ S_{q'} &= \{q'_1, q'_2, q'_3\}. \end{aligned}$$

Between q_1 and q'_1 there are no transitions since $\lambda(e_2, e_4) = *$, therefore, every suitable $\lambda_s(e_2, e_4) \in \{3, 4, *\}$, but since $\mathcal{C}_{q_1}(e_2) = 3$ and $\lambda(e_2, e_5) \neq 0$ (i.e., $\mathcal{C}_{q_1}(e_2)$ is redistributed between e_4 and e_5) it is clear that there cannot be a reallocation $q_1 \xrightarrow{\lambda_s} q'_1$. On the contrary, q_3 and q'_3 have the configurations as q and q' , respectively. However, between q_3 and q'_3 there exist two transitions corresponding to the reallocations that assign to the pair (e_2, e_4) the multiplicity 4 and $*$.

Consider now, the formula $\phi \equiv x.a^5 = y$ where x and y in q have the following interpretation: $\theta(x) = e_1$ and $\theta(y) = e_3$. In Example 5.7.2 we have seen that in q (for $M = 2$) ϕ can be either true or false. In $\mathcal{H} \uparrow 4$, the ambiguity is resolved. ϕ holds only in q_2 and it is false in q_1 and q_3 since in the latter e_2 represents at least 5 entities. \square

Theorem 5.7.5. For all HABA \mathcal{H} such that $\mathcal{C}(\mathcal{H}) = M < \hat{M}$: $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{H} \uparrow \hat{M})$.

Proof. See Appendix B.3. \square

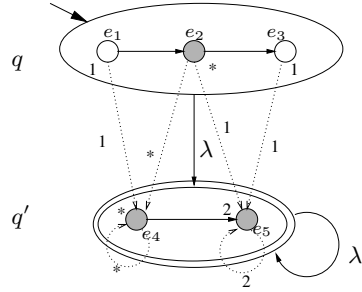


Figure 5.22: The HABA \mathcal{H} with $\mathcal{C}(\mathcal{H}) = 2$.

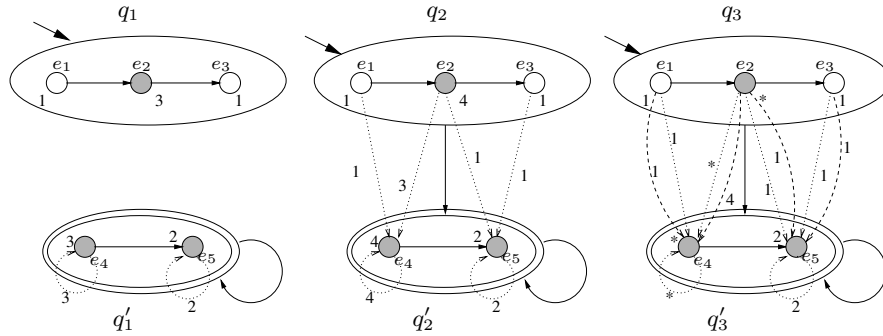


Figure 5.23: The HABA $\mathcal{H} \uparrow 4$.

Assumptions. In the remainder of this section, we assume that the necessary duplication as well as stretching have been carried out already: that is, we will assume that a state $q \in Q$ has an extra component $N_q \subseteq E_q$ that contains the entities that are *new in* q , and that the global constant M equals $K(\phi) - 1$. As for $\mathcal{A}llTL$ model-checking (cf. Section 4.5), other assumptions needed below are that every quantified variable is different (i.e., the formula has been α -converted) and every quantified variable actually appears free in the sub-formula; that is, we only consider formulae $\exists x.\phi$ for which $x \in fv(\phi)$. Note that this imposes no real restriction, since $\exists x.\phi$ is equivalent to $\exists x.(x \text{ alive} \wedge \phi)$.

5.7.2 Valuations

As for the $\mathcal{A}llTL$ model-checking algorithm defined in Chapter 4, in a given state, a valuation of a $\mathcal{N}allTL$ formula is an interpretation of its free logical variables as entities of the state. Such an interpretation is meant to establish the validity of the atomic propositions within the formula, which in $\mathcal{N}allTL$ have the form $\alpha_1 = \alpha_2$, α new, and $\alpha_1 \rightsquigarrow \alpha_2$. Because we allow unbounded and multiple entities as interpretations of logical variables, a simple mapping

from variables to entities would not be enough to decide these atomic propositions. Here we face a generalisation of the problem encountered for $\mathcal{A}llTL$ where we permitted the interpretation of variables onto the black hole. The interpretation of the valuations should be able to express not only whether two variables mapped onto an unbounded entity refer to *the same instance* or not. In fact, in the latter case, it should provide us with information about the *distance* between the two instances (within the unbounded/multiple entity) where the variables are supposed to be interpreted. Such information is necessary in order to decide equality propositions (e.g. $x.a^2 = y.a^5$) as well as leads-to propositions (e.g. $x.a^2 \rightsquigarrow y.a^5$).

For a set of entities E let $E^-, E^+ \subseteq LVAR$ be two special sets of logical variables defined as:

$$\begin{aligned} E^- &= \{e^- \mid e \in E\} \\ E^+ &= \{e^+ \mid e \in E\}. \end{aligned}$$

In the following let $E^\pm = E^- \cup E^+$. For these special variables, we will have always a fixed interpretation: variable e^- is interpreted in the first instance of the entities represented by e , and e^+ is interpreted in the last instance of e .

Definition 5.7.6 (Valuations). Let γ be a configuration. A γ -valuation is a tuple (ψ, Θ, δ) where

- ψ is a $\mathcal{N}allTL$ -formula;
- $\Theta: fv(\psi) \cup E_\gamma^\pm \rightarrow E_\gamma$ a partial function mapping every free variable to an entity such that $\forall e \in E_\gamma^\pm : \Theta(e^-) = \Theta(e^+) = e$.
- $\delta: (fv(\psi) \cup E_\gamma^\pm) \times (fv(\psi) \cup E_\gamma^\pm) \rightarrow \mathbb{M}^*$, a partial function that satisfies the conditions in Table 5.5.

We denote by V_γ the set of *all* γ -valuations ranged over by v and we write V_q for V_{γ_q} .

The function δ is the notion of distance that is used in a valuation together with the interpretation Θ to decide the equality and leads-to propositions. Some comments on the conditions reported in Table 5.5 are now in order. Condition (DELTA GAMMA 1) enforces the consistency between δ and the cardinality function \mathcal{C}_γ of the configuration. In particular it implies that $\{(e^-, e^+) \mid e \in E^\pm\} \subseteq \text{dom}(\delta)$. Condition (DELTA GAMMA 2) gives consistency between δ and μ_γ . Condition (DELTA THETA 1) and (DELTA THETA 2) relate Θ and δ . (DELTA THETA 1) can be rephrased saying that the distance between two variables is defined only if the variables are interpreted in reachable entities. The other direction is also a reasonable property to require. However, it is not necessary to impose it since it can be derived by the other properties as stated by the following result.

Proposition 5.7.7. For all γ -valuations (ψ, Θ, δ) :

(DELTA GAMMA 1)	$\forall e \in E_\gamma : \delta(e^-, e^+) \oplus 1 = \mathcal{C}_\gamma(e)$
(DELTA GAMMA 2)	$\forall e_1, e_2 \in E_\gamma : (e_1 \prec_\gamma e_2 \Leftrightarrow \delta(e_1^+, e_2^-) = 1)$
(DELTA THETA 1)	$\forall x, y \in fv(\psi) :$ $(x, y) \in \text{dom}(\delta) \Rightarrow x, y \in \text{dom}(\Theta) \wedge \Theta(x) \preceq_\gamma^* \Theta(y)$
(DELTA THETA 2)	$\Theta(x) = e \Leftrightarrow \delta(e^-, x) \geq 0 \wedge \delta(x, e^+) \geq 0$
(DELTA MET 1)	$(x, x) \in \text{dom}(\delta) \Rightarrow \delta(x, x) = 0$
(DELTA MET 2)	$(x, y), (y, z) \in \text{dom}(\delta) \Rightarrow (x, z) \in \text{dom}(\delta)$
(DELTA MET 3)	$(x, y), (x, z) \in \text{dom}(\delta) \Rightarrow (\delta(x, y) \oplus \delta(y, z) = \delta(x, z)) \vee$ $(\delta(x, z) \oplus \delta(z, y) = \delta(x, y))$

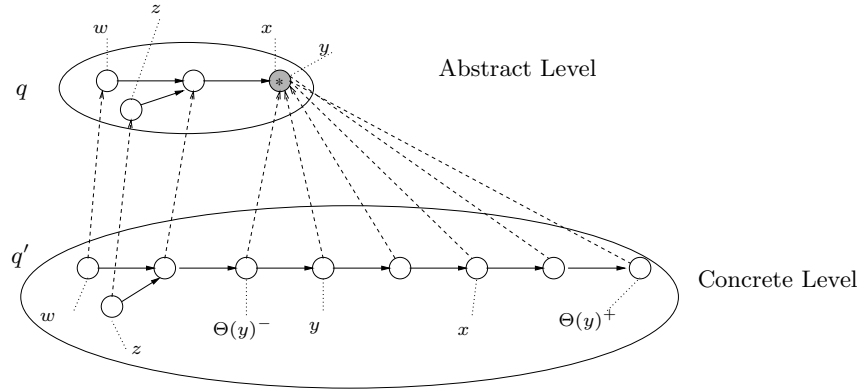
Table 5.5: Conditions on a valuation (ψ, Θ, δ) .

Figure 5.24: An interpretation of the variables corresponding to a valuation.

- (a) $(\exists j > 0 : \mu^j \circ \Theta(x) = \Theta(y) \neq \perp) \Rightarrow (x, y) \in \text{dom}(\delta)$
 (b) $\Theta(x) = \Theta(y) \neq \perp \Rightarrow (x, y) \in \text{dom}(\delta) \vee (y, x) \in \text{dom}(\delta)$

Proof. See Appendix B.3. □

Corollary 5.7.8. For all γ -valuations $(\psi, \Theta, \delta) : x \in \text{dom}(\Theta) \Rightarrow (x, x) \in \text{dom}(\delta)$.

Proof. Straightforward from Proposition 5.7.7. □

Example 5.7.9. Consider Figure 5.24. Assume that in a valuation, Θ and δ are defined as follows:

$$\begin{array}{ll}
 \delta(\Theta(y)^-, y) = 1 & \delta(\Theta(y)^-, x) = 3 \\
 \delta(y, \Theta(y)^+) = * & \delta(x, \Theta(y)^+) = 2 \\
 \delta(y, x) = 2 & \delta(x, y) = \perp \\
 \delta(w, \Theta(y)^-) = 2 & \delta(z, \Theta(y)^-) = 2
 \end{array}$$

And $\delta(w, x) = \delta(z, x) = \delta(w, \Theta(y)^+) = \delta(z, \Theta(y)^+) = *$. Note that, since $\Theta(w)$ and $\Theta(z)$ are unreachable we have $\delta(w, z) = \delta(z, w) = \perp$. On the concrete level, for a state where the unbounded entity is instantiated by six concrete entities, the interpretation given by the valuation corresponds to evaluate the free variables as depicted in the concrete state q' . \square

Metrics versus δ . Since δ expresses our notion of distance, it is somehow natural as well as interesting to relate this concept to metrics in the context of metric spaces. For, let us deviate a bit from the exposition of the model checking algorithm and make a short comparison.

A *metric space* is a set S with an associated function $m : S \times S \rightarrow \mathbb{R}_{\geq 0}$ (called metric) satisfying the following properties:

1. $\forall x \in S : m(x, x) = 0$
2. $\forall x, y \in S : m(x, y) = 0 \Rightarrow x = y$
3. $\forall x, y \in S : m(x, y) = m(y, x)$
4. $\forall x, y, z \in S : m(x, z) \leq m(x, y) + m(y, z)$

By condition 2, the distance between two points is zero only if these two points are actually the same. Condition 3 is the symmetric law and 4 is the well-known triangle inequality. Different choices of metric on a given set give rise to different metric spaces. If condition 2 is dropped, a *pseudo metric* space is obtained, whereas without condition 3, we would obtain a so-called *quasi metric* space [104]. As defined in Definition 5.7.6, the partial function δ clearly satisfies postulate 1 of a metric by (DELTA MET 1). Moreover, from condition (DELTA MET 1) – (DELTA MET 3) it is possible to derive postulate 4 as stated by:

Proposition 5.7.10 (triangle inequality).

$$(x, y), (y, z) \in \text{dom}(\delta) \Rightarrow \delta(x, z) \leq \delta(x, y) \oplus \delta(y, z).$$

Proof. $(x, y), (y, z) \in \text{dom}(\delta)$ implies by (DELTA MET 2) $(x, z) \in \text{dom}(\delta)$. Then, by (DELTA MET 3) we have

$$\delta(x, y) \oplus \delta(y, z) = \delta(x, z) \quad \text{or} \quad (5.19)$$

$$\delta(x, z) \oplus \delta(z, y) = \delta(x, y) \quad (5.20)$$

If (5.19) holds then the statement of the proposition holds as well. If (5.20) holds then we have

$$\delta(x, y) \oplus \delta(y, z) = \delta(x, z) \oplus \delta(z, y) \oplus \delta(y, z) \geq \delta(x, z)$$

that is what we wanted to prove. \square

Hence, δ , where defined, satisfies condition 1 and 4. By the consideration described above we could then classify δ as a “partial pseudo-quasi metric”.

Abstract semantics and distance for navigation expressions. We can adapt the definition of $\llbracket \alpha \rrbracket$ given in Section 5.2 in order to exploit the information given by a valuation. First of all let us consider an example that shows some difficulties towards such definition.

Example 5.7.11. Consider the configuration γ represented in Figure 5.25 and a γ -valuation v were $\Theta_v(x) = e_1$ and $\Theta_v(y) = e_{10}$ and δ_v is given by:

$$\begin{array}{ll} \delta_v(x, e_1^-) = 0 & \delta_v(x, e_1^+) = 0 \\ \delta_v(x, e_2^-) = 1 & \delta_v(x, e_2^+) = 1 \\ \delta_v(x, e_3^-) = 2 & \delta_v(x, e_3^+) = 4 \\ \delta_v(x, e_4^-) = 5 & \delta_v(x, e_4^+) = 6 \\ \delta_v(x, e_5^-) = 7 & \delta_v(x, e_5^+) = 7 \\ \delta_v(x, e_6^-) = 8 & \delta_v(x, e_6^+) = 9 \\ \delta_v(x, e_7^-) = 10 & \delta_v(x, e_7^+) = 10. \end{array}$$

The natural way to define the entity corresponding to expression $x.a^n$ would be to exploit δ_v by taking the entity e such that $\delta_v(x, e^-) \leq n \leq \delta_v(x, e^+)$. Hence, for example, the interpretation of $x.a^6$ should be e_4 whereas $x.a^8$ should be interpreted onto e_6 . However, this is too naive. In fact, consider the expressions $x.a^{14}$ or $x.a^{25}$, it is clear that the approach described before does not work since there are no entities with a distance more than 10 from x , and nevertheless looking at the configuration, both expressions must point to an existing entity. \square

It is not difficult to see that the problem described in the previous example stems from the existence of a cycle. In order to deal with it we introduce the following notion. For a configuration $\gamma = (E, \mu, \mathcal{C})$ and a γ -valuation v let $\mathcal{C}_v : (\text{LVAR} \times \mathbb{N}) \rightarrow \mathbb{M}^*$ given by:

$$\mathcal{C}_v(x, k) = \delta_v(x, \Theta_v(x)^+) \oplus \sum_{1 \leq i \leq k} \mathcal{C}(\mu^i(\Theta_v(x))).$$

Informally, $\mathcal{C}_v(x, k)$ returns the cumulative cardinality obtained performing k steps from the interpretation of x in v , i.e., $\Theta_v(x)$ up to $\mu^k(\Theta_v(x))$ and summing up all the cardinalities of the entities encountered in between. Note that if there exists a cycle some cardinality may be considered more than once.

Example 5.7.12. Consider again the configuration γ in Figure 5.25. We have:

$$\begin{array}{lll} \mathcal{C}_v(x, 1) = 1 & \mathcal{C}_v(x, 2) = 4 & \mathcal{C}_v(x, 3) = 6 \\ \mathcal{C}_v(x, 4) = 7 & \mathcal{C}_v(x, 5) = 9 & \mathcal{C}_v(x, 6) = 10 \\ \mathcal{C}_v(x, 7) = 13 & \mathcal{C}_v(x, 8) = 15 & \mathcal{C}_v(x, 9) = 16 \dots \end{array}$$

Looking at the previous values we can see that performing 7 steps from $\Theta(x)$, at the concrete level, we traverse 13 entities, whereas after 8 steps at the concrete level we visit 15 entities. Therefore, at this point it should be obvious that the interpretation of $x.a^{14}$, should be the entity $\mu^8(\Theta(x))$. \square

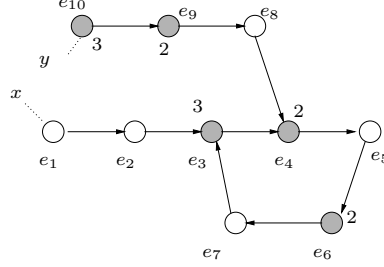


Figure 5.25: A cycle induces a difference between distance and cumulative cardinality.

Definition 5.7.13. Let $\gamma = (E, \mu, \mathcal{C})$ be a configuration and $v = (\phi, \Theta, \delta) \in V_\gamma$. The abstract semantics of the navigation expression is the function $\llbracket \cdot \rrbracket_{\gamma, v} : \text{Nav} \rightarrow (\text{Ent}^\perp \times (\mathbb{M}^* \cup \{0\}))$ given by:

$$\llbracket \text{nil} \rrbracket_{\gamma, v} = (\perp, 0)$$

$$\llbracket x.a^n \rrbracket_{\gamma, v} = \begin{cases} (\Theta(x), \delta(\Theta(x)^-, x) \oplus n) & \text{if } \delta(x, \Theta(x)^+) \geq n \\ (\mu^j \circ \Theta(x), n - \mathcal{C}_v(x, j-1) - 1) & \text{if } \delta(x, \Theta(x)^+) < n \text{ and} \\ & j = \min\{k > 0 \mid \mathcal{C}_v(x, k) \geq n\} \\ (\perp, 0) & \text{otherwise} \end{cases}$$

The abstract semantics of the navigation expression $x.a^n$ is defined as a pair (e, k) where e is either the (abstract) entity in which $x.a^n$ is actually interpreted or \perp in case the $x.a^n$ dereferences a null pointer. The second component $k \in \{0, \dots, M, *\}$ represents the offset of the instance corresponding to $x.a^n$ with respect to the first instance in e . If $e = \perp$ then k is set to 0. Note that the value $n - \mathcal{C}(x, j-1) - 1$ can be computer since $n < K(\phi)$ and it is not $*$. As expected the semantics of nil does not denote any entity and therefore is $(\perp, 0)$. As a notation we write $\llbracket \alpha \rrbracket^1$ and $\llbracket \alpha \rrbracket^2$ for the first and the second component of $\llbracket \alpha \rrbracket$, respectively. Moreover, $\llbracket \alpha_1 \rrbracket = \llbracket \alpha_2 \rrbracket$ stands for $\llbracket \alpha_1 \rrbracket^1 = \llbracket \alpha_2 \rrbracket^1 \wedge \llbracket \alpha_1 \rrbracket^2 = \llbracket \alpha_2 \rrbracket^2$. Finally, we write $\llbracket \alpha \rrbracket = \perp$ and $\llbracket \alpha \rrbracket \neq \perp$ for $\llbracket \alpha \rrbracket^1 = \perp$ and $\llbracket \alpha \rrbracket^1 \neq \perp$ respectively.

Example 5.7.14. Using the cumulative cardinality computed in the previous example, we have $\llbracket x.a^3 \rrbracket_{\gamma, v} = (\mu^2 \circ \Theta(x), 3 - \mathcal{C}_v(x, 1) - 1) = (e_3, 1)$. \square

In the definition of atomic valuations (see Definition 5.7.15) it is useful to refer to a notion of distance of two navigation expressions α_1 and α_2 in the context of some valuation. To be more specific we do not need the precise distance between α_1 and α_2 , but it is enough to know whether the distance is undefined (i.e., α_2 is unreachable from α_1), zero (i.e., α_1 and α_2 denotes the same instance of the same entity), or positive (i.e., from α_1 it is possible to reach α_2). These values are precisely what is needed to decide atomic propositions such as equations like $\alpha_1 = \alpha_2$ and leads-to propositions like $\alpha_1 \rightsquigarrow \alpha_2$. Thus, we

define a three-valued function Δ that given two navigation expressions returns undefined (\perp), zero, or strictly positive (\top) depending on the distance between the two expressions.

For a configuration γ and a $v \in V_\gamma$, the function $\Delta_{\gamma,v} : Nav \times Nav \rightarrow \{\perp, 0, \top\}$ is given by:

$$\begin{aligned} \Delta_{\gamma,v}(nil, nil) &= 0 \\ \Delta_{\gamma,v}(x.a^n, nil) &= \begin{cases} 0 & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \perp \\ \top & \text{if } \exists j > 0 : \mu^j(\llbracket x.a^n \rrbracket_{\gamma,v}^1) = \perp \\ \perp & \text{otherwise} \end{cases} \\ \Delta_{\gamma,v}(nil, x.a^n) &= \begin{cases} 0 & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \perp \\ \perp & \text{otherwise} \end{cases} \\ \Delta_{\gamma,v}(x.a^n, y.a^m) &= \begin{cases} 0 & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 = \perp \\ 0 & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp \text{ and } \llbracket x.a^n \rrbracket_{\gamma,v}^2 \neq * \\ 0 & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp \text{ and } \llbracket x.a^n \rrbracket_{\gamma,v}^2 = * \\ & \text{and } (\delta(x, y) \oplus m = n \vee \delta(y, x) \oplus n = m) \\ \top & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 \neq \llbracket y.a^m \rrbracket_{\gamma,v}^1 \\ & \text{and } \exists j > 0 : \mu^j(\llbracket x.a^n \rrbracket_{\gamma,v}^1) = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \\ \top & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp \\ & \text{and } \llbracket x.a^n \rrbracket_{\gamma,v}^2 < \llbracket y.a^m \rrbracket_{\gamma,v}^2 \\ \top & \text{if } \llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp \\ & \text{and } \llbracket x.a^n \rrbracket_{\gamma,v}^2 = \llbracket y.a^m \rrbracket_{\gamma,v}^2 = * \\ & \text{and } (\delta(x, y) \oplus m > n \vee \delta(y, x) \oplus n < m) \\ \perp & \text{otherwise} \end{cases} \end{aligned}$$

The distance between nil and itself is 0 by definition. The distance between an expression $x.a^n$ and nil is 0 if $x.a^n$ refers to a null pointer and \top if $x.a^n$ reaches a null pointer. The symmetric case $\Delta_{\gamma,v}(nil, x.a^n)$ is defined only if $x.a^n$ denotes a null pointer and undefined otherwise. This is because nil cannot lead to any special entity. The more complex case is $\Delta_{\gamma,v}(x.a^n, y.a^m)$. In particular by definition the distance between $x.a^n$ and $y.a^m$ is 0 if either both expressions dereference a null pointer, or they have the same semantics (defined in both components). However, in this last case, special attention must be devoted to the case $\llbracket x.a^n \rrbracket_{\gamma,v}^2 = *$. In fact, this means that both $x.a^n$ and $y.a^m$ denote something beyond our range of precision, therefore it is not clear whether the expressions denote the same instance or not. By definition of $K(\phi)$, $\llbracket x.a^n \rrbracket_{\gamma,v}^2 = *$ can only happen when x and y are interpreted in the same entity as $x.a^n$ and $y.a^m$ (recall that $n, m < K(\phi)$). Thus, to solve this ambiguity we exploit the distance $\delta(x, y)$. It is straightforward to see that $x.a^n$ and $y.a^m$ point to the same instance (of the entity) if $\delta(x, y) \oplus m = n$ or $\delta(y, x) \oplus n = m$

depending whether the interpretation of x precedes the interpretation of y or vice-versa. Finally, the distance $\Delta_{\gamma,v}(x.a^n, y.a^m)$ is positive, either when the entity where $x.a^n$ is interpreted reaches the interpretation of $y.a^m$ after $j > 0$ steps (provided $\llbracket x.a^n \rrbracket_{\gamma,v}^1 \neq \llbracket y.a^m \rrbracket_{\gamma,v}^1$). Otherwise, when $x.a^n$ and $y.a^m$ are interpreted in the same entity but the offset (from the first instance of the interpretation) of $x.a^n$ is smaller than the offset of $y.a^m$. Again, as discussed above, if both offsets are $*$ then a case distinction is needed according whether (x, y) or (y, x) belongs to $\text{dom}(\delta)$.

Having at our disposal the functions $\llbracket \cdot \rrbracket_{\gamma,v}$ and $\Delta_{\gamma,v}$, we can introduce the *atomic proposition valuations* of a state q that are those q -valuations of basic propositions of $\mathcal{N}\ell\ell\text{TL}$ (i.e., freshness predicates, entity equations and leads-to predicates) that make the corresponding propositions true.

Definition 5.7.15. Let \mathcal{H} be a HABA and let $q \in Q_{\mathcal{H}}$. The *atomic proposition valuations* of q are defined by the set $AV_q \subseteq V_q$ of all $v = (\phi, \Theta, \delta)$ such that:

- $\phi = \text{tt}$;
- $\phi = (\alpha \text{ new})$ and $\llbracket \alpha \rrbracket_{q,v}^1 \in N_q$;
- $\phi = (\alpha_1 = \alpha_2)$, and $\Delta_{q,v}(\alpha_1, \alpha_2) = 0$.
- $\phi = \alpha_1 \rightsquigarrow \alpha_2$, and $(\Delta_{q,v}(\alpha_1, \alpha_2) = 0 \text{ or } \Delta_{q,v}(\alpha_1, \alpha_2) = \top)$

An atomic valuation with $\phi = \alpha \text{ new}$ must interpret α among the set of new entities in the state, i.e., N_q . For the equality proposition $\alpha_1 = \alpha_2$, the distance between α_1 and α_2 must be 0. For the leads-to proposition $\alpha_1 \rightsquigarrow \alpha_2$ the distance between α_1 and α_2 must be either 0 or positive (\top).

Model-checking \mathcal{L}_n models. When we want to model check the HABA \mathcal{H}_p representing the semantics of a program $p \equiv \text{decl } v_1, \dots, v_n : (s_1 \parallel \dots \parallel s_k) \in \mathcal{L}_n$, we should make sure that the interpretation Θ in the valuations maps the special (free) logical variables $\text{DeclLVar} = \{x_{v_1}, \dots, x_{v_n}\}$ onto the special entities representing program variables, i.e., $\text{DeclPVar} = \{e_{v_1}, \dots, e_{v_n}\}$ according to the strategy described in Section 5.5.2. That is:

$$\Theta \upharpoonright \text{DeclLVar} = \vartheta$$

where ϑ is the fixed special interpretation that links DeclLVar to DeclPVar defined by (5.6).

The next is the standard definition of the closure of a formula ϕ as already defined for the $\mathcal{A}\ell\ell\text{TL}$ model checking algorithm (cf. Section 4.5); we repeat it for completeness reasons.

Definition 5.7.16. Let ϕ be an $\mathcal{N}\ell\ell\text{TL}$ -formula. The *closure* of ϕ , $CL(\phi)$, is the smallest set of formulae (identifying $\neg\neg\psi$ with ψ) such that:

- $\phi, \text{tt}, \text{ff} \in CL(\phi)$;

- $\neg\psi \in CL(\phi)$ iff $\psi \in CL(\phi)$;
- if $\psi_1 \vee \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2 \in CL(\phi)$;
- if $\exists x.\psi \in CL(\phi)$ then $\psi \in CL(\phi)$;
- if $X\psi \in CL(\phi)$ then $\psi \in CL(\phi)$;
- if $\neg X\psi \in CL(\phi)$ then $X\neg\psi \in CL(\phi)$;
- if $\psi_1 \cup \psi_2 \in CL(\phi)$ then $\psi_1, \psi_2, X(\psi_1 \cup \psi_2) \in CL(\phi)$.

5.7.3 Tableau-graph for $\mathcal{N}allTL$

In the rest of the chapter, for $\psi \in CL(\phi)$ let

$$\begin{aligned}\Theta \upharpoonright \psi &= \Theta \upharpoonright fv(\psi) \\ \delta \upharpoonright \psi &= \delta \upharpoonright (fv(\psi) \cup E^\pm \times fv(\psi) \cup E^\pm).\end{aligned}$$

We now construct a graph that will be the basis of the model-checking algorithm. The nodes of this graph are called atoms and are built from states of a HABA and valuations of formulae from the closure.

Definition 5.7.17 (atom). Given a HABA \mathcal{H} and an $\mathcal{N}allTL$ -formula ϕ , an *atom* is a pair (q, D) where $q \in Q_{\mathcal{H}}$, $D \subseteq \{(\psi, \Theta, \delta) \in V_q \mid \psi \in CL(\phi)\}$ such that for all $v = (\psi, \Theta, \delta) \in V_q$ with $\psi \in CL(\phi)$:

- $AV_q \subseteq D$;
- if $\psi = \neg\psi'$, then $v \in D$ iff $(\psi', \Theta, \delta) \notin D$;
- if $\psi = \psi_1 \vee \psi_2$, then $v \in D$ iff $(\psi_i, \Theta \upharpoonright \psi_i, \delta \upharpoonright \psi_i) \in D$ for $i = 1$ or $i = 2$;
- if $\psi = \exists x.\psi'$, then $v \in D$ iff there exists a $(\psi', \Theta', \delta') \in D$ such that $\Theta = \Theta' \upharpoonright \psi$, $\delta = \delta' \upharpoonright \psi$, and $\Theta'(x) \neq \perp$;
- if $\psi = \neg X\psi'$, then $v \in D$ iff $(X\neg\psi', \Theta, \delta) \in D$;
- if $\psi = \psi_1 \cup \psi_2$, then $v \in D$ iff either $(\psi_2, \Theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2) \in D$, or both $(\psi_1, \Theta \upharpoonright \psi_1, \delta \upharpoonright \psi_1) \in D$ and $(X\psi, \Theta, \delta) \in D$.

The set of all atoms for a given formula ϕ constructed for HABA \mathcal{H} is denoted $A_{\mathcal{H}}(\phi)$, ranged over by A, B . We denote the components of an atom A by (q_A, D_A) .

In order to define the transitions between atoms in $A_{\mathcal{H}}(\phi)$, we need to define a notion of correspondence between some components of valuations of the source atom and valuations of the target atom. This correspondence must be such that the resulting graph is *sound* with respect to the semantics of the formulae and the semantics of the underlying HABA. First we need some intermediate definitions.

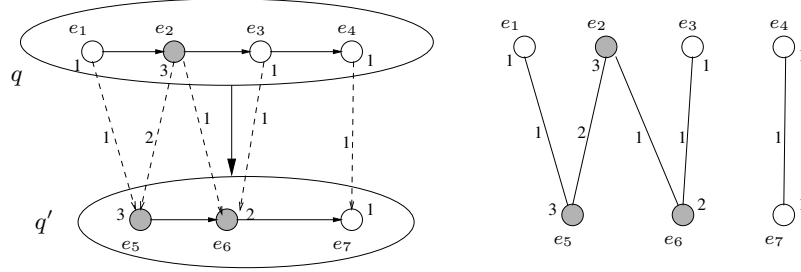


Figure 5.26: Weakly connected graphs defined by a reallocation.

Auxiliary notation. For sets S_1 and S_2 the *disjoint union* is $S_1 \uplus S_2 = (S_1 \times \{1\}) \cup (S_2 \times \{2\})$. A relation \mathcal{R} on sets S_1 and S_2 can be seen as a directed graph $G_{\mathcal{R}} = (S_1 \uplus S_2, \mathcal{R})$ with $S_1 \uplus S_2$ as a set of nodes and \mathcal{R} as a set of arcs. For such a graph there are two injective functions $\iota_1 : 2^{(S_1 \uplus S_2)} \rightarrow 2^{S_1}$ and $\iota_2 : 2^{(S_1 \uplus S_2)} \rightarrow 2^{S_2}$ that project a subset of nodes $S \subseteq S_1 \uplus S_2$, onto S_1 and S_2 :

$$\begin{aligned}\iota_1(S) &= \{a \mid (a, 1) \in S\} \\ \iota_2(S) &= \{a \mid (a, 2) \in S\}.\end{aligned}$$

An undirected graph is called *connected* if each node is reachable from any other node. A directed graph is called *weakly connected* if neglecting the direction of the arcs (i.e., treating the graph as an undirected one) it is connected. A connected subgraph is called *maximal* if it is not a proper subgraph of any other connected subgraph.

Example 5.7.18. Consider the transitions $q \rightarrow_{\lambda} q'$ in Figure 5.26 (left). Consider the graph depicted in the right part of the same figure. It is obtained from the reallocation λ as relation between the sets of entities E_q and $E_{q'}$. In this case we have: sets $\{e_1, e_5\}$, $\{e_2, e_5\}$, $\{e_2, e_6\}$, $\{e_3, e_6\}$, $\{e_1, e_2, e_5\}$, $\{e_2, e_3, e_6\}$ are (weakly) connected. Sets $\{e_4, e_7\}$ and $\{e_1, e_2, e_3, e_5, e_6\}$ are (weakly) maximal connected. An example of non-connected set is $\{e_1, e_3, e_5, e_6\}$. \square

Proposition 5.7.19. For $q_1 \xrightarrow{\lambda} q_2$, if E is a maximal weakly connected subgraph of $(E_{q_1} \uplus E_{q_2}, \lambda)$ such that $\perp \notin E$, then $\mathcal{C}_{q_1}(\iota_1(E)) = \mathcal{C}_{q_2}(\iota_2(E))$.

Proof. See Appendix B.3. \square

This proposition states that in a reallocation λ , the consistency on cardinalities¹¹ is not limited to an entity e and its images (via λ): it naturally extends to the level of weakly maximal connected subgraphs derived by λ . For example, in Figure 5.26, there exists a correspondence between the global weight

¹¹ \perp is excluded since it is not a proper entity and it does not have a cardinality. Therefore it is meaningless to require any kind of consistency.

of $\{e_1, e_2, e_3\}$ and $\{e_5, e_6\}$ that are projections of a maximal weakly connected subgraph. The same consistency criterion holds for $\{e_4\}$ and $\{e_7\}$ since $\{e_4, e_7\}$ is maximal connected.

Atom reallocations. We need now to define the notion of arcs between atoms. Similar to the graph we have constructed for the model-checking algorithm for $\mathcal{A}\ell\ell\text{TL}$ (cf. Definition 4.5.11), it should not be a surprise that these arcs will be of the form:

$$(q, D) \rightarrow_\lambda (q', D'). \quad (5.21)$$

In defining this transition relation, however, few aspects deserve special attention. More specifically, there exist some dependencies between D and D' in case the former contains valuations with formulae involving a next operator, i.e., of the form $X\psi$. In fact, such formulae express properties that are not confined to the current atom A but, on the contrary, refer to every atom connected to A by a transition. Therefore, if $(X\psi, \Theta, \delta) \in D$ and $(\psi, \Theta', \delta') \in D'$ we have the following dependencies:

- the interpretation given by Θ' must agree with the interpretation Θ according to the reallocation λ ;
- the distance given by δ' must agree to the distance δ at least for those entities that are “invariant” under the transition.

The correspondence between these components is not trivial. For example, it is clear that we would like to have $\Theta'(x) = \lambda \circ \Theta(x)$. However, for HABA with references, we can very well have that $|\lambda(\Theta(x))| > 1$ therefore, $\Theta'(x)$ can be interpreted onto an element of $\lambda(\Theta(x))$, say e' . Nevertheless, since $\Theta'(x) = e'$ imposes some restrictions on δ' (according to the definition of valuation) then it could be that these restrictions are incompatible with the dependencies that must exist between δ' and δ . The risk we run here is to set up a transition between atoms that would be unsound w.r.t. the semantics of $\mathcal{N}\ell\ell\text{TL}$ and the behaviour of the HABA. These considerations lead us to define, given a valuation v which is the set of sound valuations w.r.t. v after a reallocation λ . We use this set in order to rule out “bad” valuations in the target atom.

Definition 5.7.20. For states q and q' , the *valuation reallocator from q to q'* is the function $[- \circ -] : (E_q \times E_q \rightarrow \mathbb{M}) \times V_q \rightarrow 2^{V_{q'}}$ that, given a q -valuation v and a reallocation $\lambda : q \Rightarrow q'$, returns a set of q' -valuations compatible w.r.t. v and λ . It is defined as:

$$[\lambda \circ (\psi, \Theta, \delta)] = \{(\psi, \Theta', \delta') \in V_{q'} \mid \text{condition 1 and 2 hold}\}$$

1. $\forall x \in fv(\psi) : \Theta'(x) \in \lambda \circ \Theta(x)$
2. for all weakly maximal connected subgraph E_{mcs} of $(E_q \uplus E_{q'}, \lambda)$ and for all $x \in fv(\psi)$ such that $\Theta(x) \in \iota_1(E_{mcs})$:

- (a) $\delta(\text{first}(\iota_1(E_{mcs}))^-, x) = \delta'(\text{first}(\iota_2(E_{mcs}))^-, x)$
- (b) $\delta(x, \text{last}(\iota_1(E_{mcs}))^+) = \delta'(x, \text{last}(\iota_2(E_{mcs}))^+)$
- (c) $\forall y \in \text{fv}(\psi) : \Theta(y) \in \iota_1(E_{mcs}) \Rightarrow \delta(x, y) = \delta'(x, y).$

Condition 1 states that for free variable x , the interpretation $\Theta'(x)$ must be obtained by applying λ to $\Theta(x)$. Condition 2(c) enforces the correspondence between the distance of variables x, y interpreted on E_{mcs} . Moreover, the distance between a variable x w.r.t. the beginning and the end of the maximal connected graph (where x is interpreted), before and after the transition must be preserved (conditions 2(a) and 2(b)). The other distances between x and the entities within E_{mcs} can be derived from those distances imposed by condition 2. Note that since Θ' and δ' are in the context of a q' -valuation, they satisfy, by definition, the conditions of Table 5.5. The choice of constraints imposed in the previous definition derives from the compatibility we want to have between the graph $G_{\mathcal{H}}$ and the allocation sequences in the language of \mathcal{H} . It becomes clear in the next example.

Example 5.7.21. Consider the reallocation λ on top of Figure 5.27 where $K(\phi) = 4$ and:

$$\begin{aligned}
\delta(e_1^-, y) &= 0 & \delta(y, e_1^+) &= 1 \\
\delta(y, e_2^-) &= 2 & \delta(y, e_2^+) &= * \\
\delta(y, x) &= * \\
\delta(e_1^-, x) &= * & \delta(x, e_1^+) &= * \\
\delta(x, e_2^-) &= * & \delta(x, e_2^+) &= *
\end{aligned}$$

The maximal weakly connected subgraph we are concerned with in this example is given by $E = (\{e_1, e_2, e_3, e_4\}, \lambda)$. According to Definition 5.7.20, valuation (ψ, Θ', δ') in $[\lambda \circ (\psi, \Theta, \delta)]$ has at least $\Theta'(x) \in \{e_3, e_4\}$ because of condition 1. Furthermore, by condition 2 the following distances must be preserved:

$$\begin{aligned}
\delta'(e_3^-, y) &= \delta(e_1^-, y) &= 0 \\
\delta'(y, e_4^+) &= \delta(y, e_2^+) &= * \\
\delta'(y, x) &= \delta(y, x) &= * \\
\delta'(e_3^-, x) &= \delta(e_1^-, x) &= * \\
\delta'(x, e_4^+) &= \delta(x, e_2^+) &= *
\end{aligned}$$

From these distances it is possible to derive the others. We start by computing the values for y . Since $\delta'(e_3^-, y) = 0$ and by condition (DELTA GAMMA 1) of Table 5.5 it follows $\delta'(e_3^-, e_3^+) \oplus 1 = *$, then we have:

$$\delta(y, e_3^+) \oplus 1 = *$$

which has as solution $\delta(y, e_3^+) \in \{3, *\}$. Moreover, from these values we can deduce $\delta(y, e_4^-) = \delta(y, e_3^+) \oplus 1 = *$.

Concerning x we can immediately exclude $\Theta'(x) = e_4$ otherwise $\delta'(x, e_4^+) \neq *$ contradicting condition 2 of Definition 5.7.20. Hence, $\Theta'(x) = e_3$. Because of condition (DELTA MET 3) of Table 5.5,

$$* = \delta(e_3^-, e_3^+) \oplus 1 = \delta'(e_3^-, x) \oplus \delta'(x, e_3^+) \oplus 1$$

and since $\delta'(e_3^-, x) = *$ the solution of the previous equation is $\delta'(x, e_3^+) \in \{0, 1, 2, 3, *\}$. Moreover, since $\mathcal{C}(e_4) = 1$ then $\delta'(x, e_4^+) = \delta'(x, e_4^-) = 0$. Summarising we have the following possibilities for the other values of δ' :

$$\begin{aligned} \delta'(y, e_3^+) &\in \{3, *\} \\ \delta'(y, e_4^-) &= * \\ \delta'(x, e_3^+) &\in \{0, 1, 2, 3, *\} \\ \delta'(x, e_4^-) &= * \end{aligned}$$

Some of these solution are impossible, and the informations we have to our disposal allow us to be more precise and by narrowing the set of possible δ' . We have not used so far that $\delta'(y, x) = *$. In fact this implies $\delta'(y, e_3^+) = *$ excluding therefore the value 3 in the solutions previously computed. Moreover, we can use that $\delta'(x, e_4^+) = *$ and $\delta(e_4^-, e_4^+) = 0$ to figure out that actually the range of solutions for $\delta'(x, e_3^+)$ is much smaller than $\{0, 1, 2, 3, *\}$. In fact,

$$* = \delta'(x, e_4^+) = \delta'(x, e_4^-) = \delta'(x, e_3^+) \oplus \delta'(e_3^+, e_4^-) = \delta'(x, e_3^+) \oplus 1.$$

The solution of the last equation is $\delta'(x, e_3^+) \in \{3, *\}$.

We may wonder why $[\lambda \circ (\psi, \Theta, \delta)]$ contains sensible options for the valuations (ψ, Θ', δ') that must be contained in an atom in order to yield an edge in the graph. In order to give some insights, let us consider how this abstract transition is reflected at the concrete level. In the bottom of the Figure 5.27, two possible concrete states are depicted. They are generated from q and q' , and connected by a reallocation λ' that is a concretion of λ . This is the particular case where e_2 is instantiated with 11 concrete entities: e_1^2, \dots, e_{11}^2 and e_3 is instantiated with 12 entities, namely e_1^3, \dots, e_{12}^3 . The shadow boxes depict the generator. The symbolic interpretation given in the valuation by Θ and δ , is projected at the concrete level to several choices for a concrete interpretation θ . Indeed, for x we can have: either $\theta(x) = e_5^2$, or $\theta(x) = e_6^2$ or $\theta(x) = e_7^2$. Fixing one of these possibilities, by λ' we obtain $\theta'(x)$. Thus after the transition, we get the following possibilities:

$$\begin{aligned} \theta(x) = e_5^2 &\Rightarrow \Theta'(x) = e_3 & \delta'(e_3^-, x) = * & \delta'(x, e_3^+) = * \\ \theta(x) = e_6^2 &\Rightarrow \Theta'(x) = e_3 & \delta'(e_3^-, x) = * & \delta'(x, e_3^+) = * \\ \theta(x) = e_7^2 &\Rightarrow \Theta'(x) = e_3 & \delta'(e_3^-, x) = * & \delta'(x, e_3^+) = 3. \end{aligned}$$

Note that the first two options are actually the same and this is according to the solutions we have found before. \square

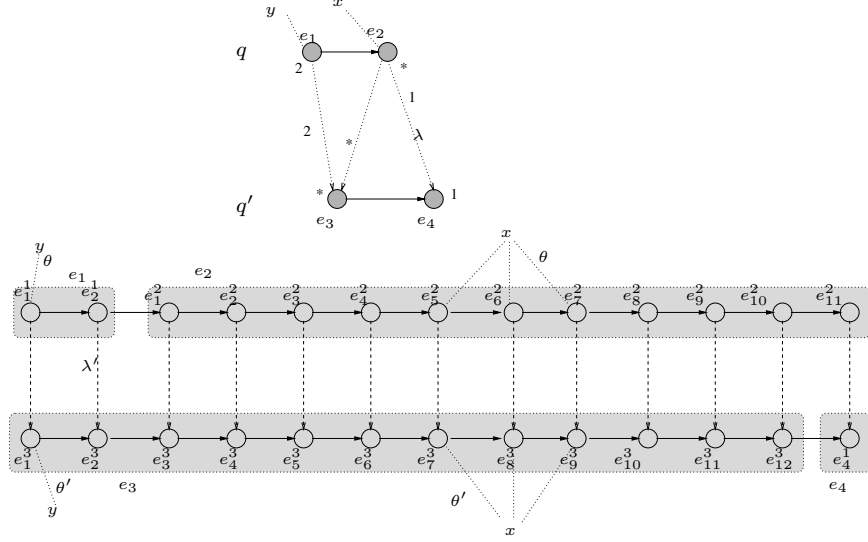


Figure 5.27: Dependencies (at the abstract and concrete level) between distances of free variables in a weakly maximal connected subgraph.

The definition of valuation reallocator is extended point-wise to a set $V \subseteq V_q$:

$$[\lambda \circ V] = \bigcup_{v \in V} [\lambda \circ v]$$

we can then define the composition for a chain of reallocations $\lambda_j, \dots, \lambda_0$ (with $j \geq 0$) of q -valuation:

$$[\lambda_j \circ \dots \circ \lambda_0 \circ v] = [\lambda_j \circ [\lambda_{j-1} \circ \dots \circ \lambda_0 \circ v]]$$

Definition 5.7.22 (tableau graph). The *tableau graph* for a HABA \mathcal{H} and an $\mathcal{N}allTL$ -formula ϕ , denoted $G_{\mathcal{H}}(\phi)$, consists of vertexes $A_{\mathcal{H}}(\phi)$ and edges $\rightarrow \subseteq A_{\mathcal{H}}(\phi) \times (Ent \times Ent \rightarrow \mathbb{M}^*) \times A_{\mathcal{H}}(\phi)$ such that $(q, D) \rightarrow_{\lambda} (q', D')$ iff

- $q \rightarrow_{\lambda} q'$
- for all $X\psi \in CL(\phi)$: $(X\psi, \Theta, \delta) \in D \Leftrightarrow \exists(\psi, \Theta', \delta') \in [\lambda \circ (\psi, \Theta, \delta)] \cap D'$.

The reallocation λ attached on the edge is the same as the reallocation of the corresponding transition in the HABA. Furthermore, in case of a formula $X\psi$ in the source atom, there is a transition if we have a valuation on the target state that contains ψ where the components Θ' , and δ' are compatible (in the sense of Definition 5.7.20) with the reallocation λ and the corresponding components of the source state. Since the set $[\lambda \circ (\Theta, \delta)]$ is not always a singleton, there can be more than one transition to different atoms, concerning the same $X\psi$ (as we have seen in Example 5.7.21).

5.7.4 Paths

Definition 5.7.23. An (*allocation*) *path* in $G_{\mathcal{H}}(\phi)$ is an infinite sequence $\pi = (q_0, D_0) \lambda_0 (q_1, D_1) \lambda_1 \cdots$ such that:

1. $q_0 \lambda_0 q_1 \lambda_1 \cdots \in \text{runs}(\mathcal{H})$;
2. for all $i \geq 0$, $(q_i, D_i) \rightarrow_{\lambda_i} (q_{i+1}, D_{i+1})$;
3. for all $i \geq 0$ and all $(\psi_1 \cup \psi_2, \Theta, \delta) \in D_i$, there exists a $j \geq i$ such that $\exists(\psi_2, \Theta', \delta') \in D_j \cap [\lambda_{j-1} \circ \cdots \circ \lambda_i \circ (\psi_2, \Theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2)]$.

The next is the important concept of fulfilling path.

Definition 5.7.24 (fulfilling path). Given an allocation path π in $G_{\mathcal{H}}(\phi)$, we say that π *fulfils* ϕ if the underlying run generates an allocation triple (σ, N, θ) such that $\sigma, N, \theta \models \phi$.

As usual, if ϕ is clear from the context, we call π a *fulfilling path*. Furthermore, if there exists $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$ such that $\sigma, N, \theta \models \phi$ we say that ϕ is \mathcal{H} -satisfiable. In contrast with the definition of HABA given in Chapter 4, a run ρ of the HABAs we consider in this paper does not always generates triples (σ, N, θ) in the sense of Definition 5.3.24. We write $Gen(\rho)$ for the set of allocation triples generated by ρ , and for a path π we write $Gen(\pi)$ for $Gen(\rho)$ where ρ is the underlying run of π .

There is a correspondence between the satisfiability of a formula in the HABA and the existence of a fulfilling path in the tableau graph.

Proposition 5.7.25. ϕ is \mathcal{H} -satisfiable if and only if there exists a path in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ .

Proof. See Appendix B.3. □

Definition 5.7.26. A subgraph $G' \subseteq G_{\mathcal{H}}(\phi)$ is *self-fulfilling* if every node A in G' has at least an outgoing edge and for every $(\psi_1 \cup \psi_2, \Theta, \delta) \in D_A$ there exists a node $B \in G'$ such that

- $A = A_0 \rightarrow_{\lambda_0} A_1 \rightarrow_{\lambda_1} \cdots \rightarrow_{\lambda_{i-2}} A_{i-1} \rightarrow_{\lambda_{i-1}} A_i = B$
- $\exists(\psi_2, \Theta_B, \delta_B) \in D_B \cap [\lambda_{i-1} \circ \cdots \circ \lambda_0 \circ (\psi_2, \Theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2)]$.

A *prefix* in $G_{\mathcal{H}}(\phi)$ is a sequence $A_0 \rightarrow_{\lambda_0} A_1 \rightarrow_{\lambda_1} \cdots \rightarrow_{\lambda_{i-2}} A_{i-1} \rightarrow_{\lambda_{i-1}} A_i$ such that A_0 is an initial atom (i.e., $q_{A_0} \in I_{\mathcal{H}}$) and A_i is in a self-fulfilling subgraph. Let $Inf(\pi)$ denote the set of nodes that appear infinitely often in the path π . $Inf(\pi)$ is a strongly connected subgraph (SCS). We can prove the following implication:

Proposition 5.7.27. π is a fulfilling path in $G_{\mathcal{H}}(\phi) \Rightarrow Inf(\pi)$ is a self-fulfilling SCS of $G_{\mathcal{H}}(\phi)$.

Proof. See Appendix B.3. □

Finally, we can collect all the previous results into the following theorem:

Theorem 5.7.28. For any HABA \mathcal{H} and formula ϕ , it is possible to verify mechanically whether $\mathcal{H} \not\models \phi$.

Proof. By Proposition 5.7.25, in order to prove that $\mathcal{H} \not\models \phi$, it is sufficient to check that in the graph $G_{\mathcal{H}}(\phi)$ there does not exist a fulfilling path π . By Proposition 5.7.27, for a path π to be fulfilling it is necessary to have as a $Inf(\pi)$ a self-fulfilling SCS. Thus, in order to check that ϕ is not satisfiable in \mathcal{H} , it is sufficient to check that every self-fulfilling SCS is not the Inf of any path. That is, if Π is the set of *all* paths in $G_{\mathcal{H}}(\phi)$ and

$$\begin{aligned}\Pi_{ful} &= \{\pi \in \Pi \mid \pi \text{ is a fulfilling}\} \\ \Pi_1 &= \{\pi \in \Pi \mid Inf(\pi) \text{ is a self-fulfilling SCS}\}\end{aligned}$$

in order to prove $\Pi_{ful} = \emptyset$ it suffices to show that $\Pi_1 = \emptyset$ (since $\Pi_{ful} \subseteq \Pi_1$). But this is the case (see below) if the following set Π_{SCS} is empty.

$$\Pi_{SCS} = \{G' \subseteq G_{\mathcal{H}}(\phi) \mid G' \text{ is a self-fulfilling SCS such that a) and b) hold}\}$$

where

- a) there exists a fulfilling prefix of G' ;
- b) for all $F \in \mathcal{F}_{\mathcal{H}} : F \cap \{q_B \mid B \in G'\} \neq \emptyset$.

We prove that

$$\Pi_{SCS} = \emptyset \Rightarrow \Pi_1 = \emptyset.$$

By contradiction, assume $\Pi_1 \neq \emptyset$ (and $\Pi_{SCS} = \emptyset$). Take $\pi = A_0 \lambda_0 A_1 \lambda_1 \dots \in \Pi_1$. Since π is a path then $\rho_{\pi} = q_{A_0} \lambda_0 q_{A_1} \lambda_1 \dots$ is an accepting run of \mathcal{H} (condition 1 of Definition 5.7.23). However, this implies that condition b) of the definition of Π_{SCS} is satisfied. Furthermore, since π is a path there exists a prefix for $Inf(\pi)$. Hence, it must be $Inf(\pi) \in \Pi_{SCS}$ which contradicts $\Pi_{SCS} = \emptyset$. Since SCS are finite and there are only a finite number of them, it is possible to verify the emptiness of Π_{SCS} . \square

The proof of Theorem 5.7.28 suggests us a procedure described by Algorithm 3 that can be used to verify whether $\mathcal{H} \not\models \phi$.

5.7.5 Discussion and future work: the HABA emptiness problem.

By Algorithm 3 we can only verify whether ϕ is *not* \mathcal{H} -satisfiable but not the reverse. In other words, the model checking algorithm described here provides us with a *partial* decidability result. The obvious manifestation of the semi-decidability is the introduction, in some cases, of *false negatives* that are originated from the abstraction applied by unbounded entities. This does not immediately imply that the model checking problem for \mathcal{NallTL} is partially

Algorithm 3 Procedure for non-satisfiability of ϕ .

```

procedure NonSatisfiable( $\mathcal{H}, \phi$ ) do
  Construct  $G_{\mathcal{H}}(\phi)$ ;
  Construct the set of self-fulfilling SCS  $\Pi_{SCS}$  having a prefixes and satisfying the accept condition on  $\mathcal{F}_{\mathcal{H}}$ ;
  if  $\Pi_{SCS} = \emptyset$  then
    Output: “ $\mathcal{H}$  does not satisfy  $\phi$ ”;
  else
    return  $G' \in \Pi_{SCS}$  and its prefix as a (possible) counterexample;
  end if
end procedure

```

decidable in itself. Some speculations on investigations we have lately undertaken suggest that, by imposing some extra constraints on the notion of fulfilling path, it could be possible to reduce the decidability of the $\mathcal{N}\hat{\mathcal{A}}\ell\ell\text{TL}$ model checking problem to deciding whether the run of the fulfilling path has a non-empty language.

Let us be more precise. We have seen in the Appendix B.3 for the proofs of the auxiliary lemmas of Proposition 5.7.25, that when comparing a path $\pi = A_0\lambda_0A_i\lambda_1\cdots$ w.r.t. an allocation sequence σ , we have a notion of consistency between the interpretation of the variables given in a valuations $v \in A_i$ and the concrete interpretation θ_i^σ on the concrete level¹². This notion of consistency given by Definition B.3.1 is local in a state. However, it can be strengthened so that a path π tightly mimics the behaviour of an allocation sequence σ . This strengthening should be done on two sides:

- requiring for every valuation of a formula $X\psi$ in A_i ($i \geq 0$) the existence of a valuation for ψ in A_{i+1} with a consistent δ_{i+1} w.r.t. the concrete interpretation in the allocation sequence θ_{i+1}^σ ;
- for every valuation of a formula $\exists x:\psi$ in atom A_i , and all the possible

¹²This definition is given in Appendix B.3 because it is only used in the proofs. However, for the sake of discussion we briefly report it here as well. Let γ' and γ be a concrete and an abstract configuration, respectively. Let $\theta : fv(\psi) \rightarrow E_{\gamma'}$ be an interpretation for the logical variables over γ' . We extend θ to the set of special logical variables $E_{\gamma'}^\pm$ as follows. Let $h : \gamma' \succrightarrow \gamma$ and $e \in E_\gamma$, then:

$$\theta^\pm(x) = \begin{cases} \theta(x) & \text{if } x \in fv(\psi) \\ \text{first}(h^{-1}(e)) & \text{if } x = e^- \\ \text{last}(h^{-1}(e)) & \text{if } x = e^+ \end{cases}$$

Definition B.3.1 Let $h : \gamma' \succrightarrow \gamma$ be a morphism where $\mathcal{C}_{\gamma'} = \mathbf{1}$, (ψ, Θ, δ) a γ -valuation and $\theta : \text{LVAR} \rightarrow E_{\gamma'}$. Then, (θ, h) is consistent with (Θ, δ) (written $(\theta, h) \simeq (\Theta, \delta)$) if

- $h \circ \theta = \Theta$
- $\forall x, y \in fv(\psi) \cup E_{\gamma'}^\pm : \delta(x, y) = \min \{ [n] \mid \mu_{\gamma'}^n \circ \theta^\pm(x) = \theta^\pm(y) \}$ where $\min \emptyset = \perp$.

concrete assignments of x given by θ_i^σ there should be a valuation for ψ in A_i with a consistent δ_i .

This strong form of consistency may be called *global consistency*. Global consistency can be used to obtain a much stronger notion of fulfilling path. The latter would probably allow us to reestablish the following fact that holds for $AllTL$ (cf. Proposition 4.5.16) but not for $\mathcal{N}allTL$ in the current setting:

$$\pi \text{ fulfils } \phi \Leftrightarrow \exists(\phi, \Theta, \delta) \in D_0 \quad (5.22)$$

where, π is a path in $G_{\mathcal{H}}(\phi)$ such that:

$$Gen(\pi) \neq \emptyset. \quad (5.23)$$

Having (5.22) to our disposal would mean being able to decide whether or not ϕ is satisfiable in \mathcal{H} under the condition however that we are able to decide (5.23). So far this question seems most likely to be undecidable¹³. Certainly more investigation have be done in order to verify this last conjecture. Accordingly the decidability of the *HABA emptiness problem*:

“given a HABA with references \mathcal{H} and a $\rho \in \text{runs}(\mathcal{H})$ is $Gen(\rho) \neq \emptyset$?”

is an important open question.

5.8 Related work

3-valued logic. In [100, 101], a framework for the generation of a family of shape analysis algorithms is presented. The framework allows to reason about pointer structures in case of destructive updates. This methodology can be instantiated in different ways to handle different kind of data structures at different levels of precision and efficiency. The 3-valued logic methodology and ours appear complementary in several aspects. The major differences with our approach can be summarised as follows. In [101], states are represented by predicates whereas our approach uses automata. Moreover, $\mathcal{N}allTL$ provides some second order capabilities given by the predicate leads-to. Both methodologies produce only safe approximations of the concrete system that is modelled, false negatives may be returned as result of the analysis. This phenomenon requires some means to tune the abstraction, so that a more precise heap is obtained. To this end, 3-valued logic technology uses *instrumentation predicates*. Every predicate modifies the model (e.g., the operational semantics of the system) and imposes the duty to prove the correctness of the new system with respect to the original one. Our approach is instead parametric in the global constant M . For the programming language it also depends on the constant L given to define L -canonicity of the state. In order to increase the precision of the heap

¹³Indeed, if the problem is undecidable, there would be no real advantages of this new setting w.r.t the current one.

representation we only need to increase the two constants. The new model obtained corresponds automatically to the original one thanks to the machinery provided by morphisms and cardinalities. There is no need to establish the equivalence of the two models. States in [101] are very abstract because of the use of a single summary node and the approach is very general since no restriction on the type of graph is imposed. We take the opposite point of view. In our approach, we try to be more concrete by morphisms that exploit information on the cardinality for entities and by keeping explicit singularities of the heap. This should provide more precision in the analysis (i.e., less spurious counterexamples) since the amount of nondeterminism is reduced. The price we pay for such precision is a less general framework. It would be interesting to study an integration of the two strategies.

[112] presents a model and an algorithm to prove safety properties of Java objects and threads based on 3-valued logic. In this context, however, entities of different states cannot be related. This problem is circumvented in the recent paper [113] (again based on 3-valued logic) that uses reallocations similar to those employed by HD-automata and by HABA without references defined in Chapter 4. The paper proposes a first-order modal (temporal) logic for allocation and deallocation of objects and threads as well as for the specification of properties related to the evolution of the heap. The properties can be verified by an abstract-interpretation algorithm, sound but not complete, that is also defined in the paper.

Others. An intuitionistic extension of Hoare logic, called *Separation Logic* for reasoning about shared mutable data structure is presented by Reynolds in [94]. The logic introduces a special (conjunction) operator that allows to describe the separation of storage into disjoint parts. It is possible therefore to extend a local specification, involving only some variables and parts of the heap, to a global specification involving also other parts of the heap. The main strength of this approach is the capability to reason in a local fashion. The logic can address several kinds of data structures like lists and trees. [67] provides a classical model for the approach introduced by Reynolds, and the relation w.r.t. the intuitionistic is investigated.

In [12] a store-less formalism for describing properties of linked data structures is defined. Moreover the paper introduces a logic, called *Alias Logic*, for reasoning about destructive update performed on data structures. For it a Hoare logic-like proof system is defined.

A spatial logic for reasoning about directed graphs is studied in [18]. The logic is used for the analysis of the manipulation of such graphs that are described by constructs of process algebra. An interesting feature of this logic is the possibility to reason locally about disjoint subgraphs.

6

An application: analysis of Mobile Ambients

6.1 Introduction

The calculus of Mobile Ambients (MA) is a calculus meant to model *wide area* computations. Introduced firstly in [19], the ambient calculus has as a main characteristic to allow active processes to move between different sites. This notion of mobility that extends the one found in Java, where only passive code can move, makes this formalism very appealing for modelling and studying mobile computations.

A wide range of work has been recently carried out on the analysis of mobile ambients [14, 40, 57, 75, 86], mostly based on static-analysis techniques and abstract interpretation [37, 36]. The analysis defined in these papers provided results on subsets of the ambient calculus with different levels of precision (for example on processes with replication, see next section).

Taking inspiration from the aforementioned papers, in this chapter we intend to apply the techniques developed in the previous chapters to analyse mobile ambients by model checking. In fact, since the natural models of ambients are trees, HABA with references, $\mathcal{N}allTL$ and its model checking algorithm defined in Chapter 5 may become suitable tools — alternative to static analysis — for the verification of mobile ambients. In particular, HABA define suitable *finite* abstractions of mobile ambients processes; $\mathcal{N}allTL$ can express interesting security properties; and the model checking algorithm can be used to check

if the properties can be satisfied. In some cases, the accuracy of the analysis may be tuned exploiting the machinery induced by unbounded entities.

This chapter is organised as follows: Section 6.2 provides some background information on the ambient calculus, presenting its syntax and semantics. Section 6.3 defines an operational semantics for MA using HABA with references. Finally, in Section 6.4 we give an overview on other techniques known in the literature used for the analysis of mobile ambients.

6.2 An Overview of Mobile Ambients

6.2.1 Syntax

We consider the pure *Mobile Ambients* calculus [19] without communication primitives.

Definition 6.2.1. Let \mathcal{N} be a denumerable set of *names* (ranged over by a, b, n, m). The set of processes over \mathcal{N} is defined according to the following grammar:

$N ::=$	(capabilities)	$P, Q ::=$	(processes)
$\text{in } n$	enter n	$\mathbf{0}$	inactivity
$\text{out } n$	exit n	$(\nu n)P$	restriction
$\text{open } n$	open n	$P \mid Q$	parallel composition
		$!P$	replication
		$n[P]$	ambient
		$N.P$	prefix.

For a process P we write $n(P)$, $fn(P)$, $bn(P)$ for the set of names, free names and bound names, respectively. In a process there can be multiple ambients with the same name. The restriction $(\nu n)P$ creates a new name called n that is private in the scope of P . $P \mid Q$ is the standard parallel composition of processes P and Q . Replication $!P$ represents an arbitrary number of copies of P and it is used to introduce recursion as well as iteration. $n[P]$ represents an ambient with name n enclosing a running process P . Ambients can be arbitrarily nested, and a graphical representation of an ambient n enclosing ambients m_1, \dots, m_i and executing process $P_1 \mid \dots \mid P_j$ is depicted in Figure 6.1(a). Capabilities provide ambients with the possibility to *interact* with other ambients. In particular, $\text{in } n$ has the effect to move the ambient that performs $\text{in } n$ into a sibling ambient called n (if there exists one). Figure 6.1(b) gives a pictorial representation of the execution of in . Symmetrically, by $\text{out } n$ an ambient nested inside n , moves outside (cf. Figure 6.1(c)). Finally, $\text{open } n$ dissolves an ambient n nested inside the one performing this capability (cf. Figure 6.1(d)).

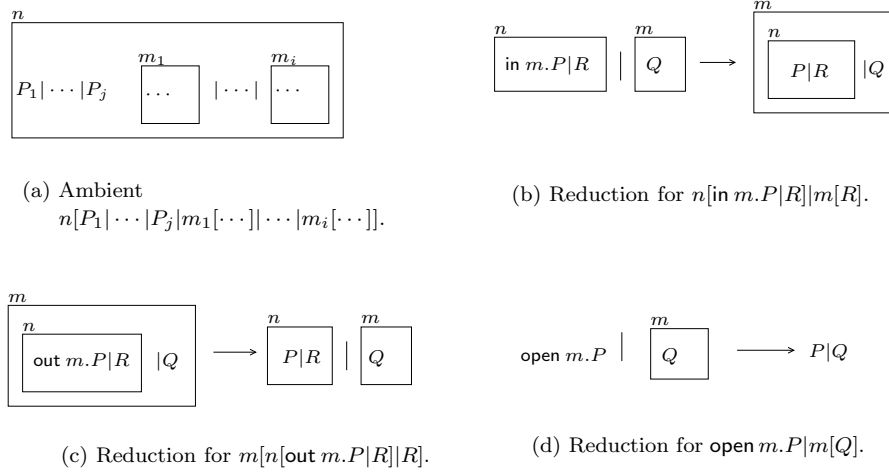


Figure 6.1: Graphical representation of nested ambients and reduction rules for capabilities.

6.2.2 Operational semantics

The standard semantics of Mobile Ambients is given in [19] on the basis of a structural congruence between processes (denoted by \equiv) and a reduction relation. Structural congruence provides us with a partitioning of processes into equivalence classes (of structurally congruent processes) within which processes are equivalent up to syntactic restructuring. Table 6.2.2 defines the structural congruence. Moreover, processes are identified up to α -conversion, i.e.,

$$(\nu n)P = (\nu m)P\{m/n\} \quad \text{if } m \notin \text{fn}(P) \quad (6.1)$$

The previous two processes are considered to be identical. This allows us to choose different appropriate representations of the same process.

It is worth to note that:

$$n[P]|n[Q] \not\equiv n[P|Q] \quad (6.2)$$

that is, multiple copies of an ambient n have distinct identities. Moreover,

$$!(\nu n)P \not\equiv (\nu n)!P \quad (6.3)$$

that is, the replication operator combined with restriction creates an infinite number of new names. This is exemplified as follows.

Example 6.2.2. Let $P_1 = !(\nu n)(n[\text{in } n])$ and $P_2 = (\nu n)!(n[\text{in } n])$. Process P_1 creates an unbounded number of new local names, i.e., it expands as:

$$n'[\text{in } n']|n''[\text{in } n'']|n'''[\text{in } n''']|\dots$$

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q \wedge Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P R \equiv Q R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Amb)
$P Q \equiv Q P$	(Struct Par Comm)
$(P Q) R \equiv P (Q R)$	(Struct Par Assoc)
$!P \equiv P !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)P Q \equiv P (\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)m[P] \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)

Table 6.1: Structural Congruence for Mobile ambients.

where n', n'', n''', \dots cannot interact with each other since these names are local. Hence, P_1 cannot perform any action. On the contrary, P_2 expands as

$$n[\text{in } n]|n[\text{in } n]|n[\text{in } n]|\dots$$

that is, an infinite number of copies of the same ambient n is created. Every copy can interact with any other one. Hence, in P_2 the instances of n can move, producing thus, every kind of possible nesting. \square

The analysis later in this chapter adopts some simplifications on processes, in particular on those of the form $(\nu n)P$.

The reduction relation \rightarrow is defined by the rules listed in Table 6.2.2. The first three rules define the effect of capabilities in one step-reductions. The reduction is then propagated within name restriction, ambient nesting, and parallel composition by the next three rules that, for this reason, are called *structural rules*. The last rule allows the use of structural congruence during reduction. As usual, \rightarrow^* stands for the reflexive and transitive closure of \rightarrow .

6.3 An analysis oriented semantics with HABA

We give an abstract semantics for the calculus introduced in Definition 6.2.1 that captures essential information useful for proving security properties. The

$n[\text{in } m.P Q] m[R] \rightarrow m[n[P Q] R]$	(Red In)
$m[n[\text{out } m.P Q] R] \rightarrow n[P Q] m[R]$	(Red Out)
$\text{open } n.P n[Q] \rightarrow P Q$	(Red Open)
$\frac{P \rightarrow Q}{(\nu n)P \rightarrow (\nu n)Q}$	(Red Res)
$\frac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$	(Red Amb)
$\frac{P \rightarrow Q}{P R \rightarrow Q R}$	(Red Par)
$\frac{P' \equiv P \quad P \rightarrow Q \quad Q \equiv Q'}{P' \rightarrow Q'}$	(Red \equiv)

Table 6.2: Reduction rules for Mobile ambients.

information we try to retrieve are along the line of [40, 57, 75, 86]. We start with some motivating examples.

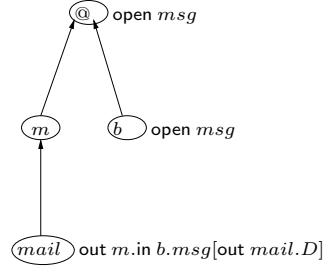
6.3.1 Motivating examples

Mobile ambients are used to model wide area systems. In such systems, security is an important issue since trustworthy ambients may operate inside and together with untrustworthy ambients. Malicious ambients can acquire information contained in other ambients by opening them. Properties such as *secrecy* of data are preserved if no untrustworthy ambient can ever open a trustworthy one.

Example 6.3.1. In [40] the following system is considered. Ambient m wants to send a message to ambient b . Messages are delivered by enclosing them in a wrapper ambient that moves inside the receiver which acquires the information by opening it. For secret messages we want to be sure that they can be opened only by the receiver b .

$$SYS_1 = m[\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]] \mid b[\text{open } \text{msg}] \mid \text{open } \text{msg}.$$

Figure 6.2 shows a pictorial view of the initial configuration of SYS_1 . Data D is secret, mail is the pilot ambient that goes out of m to reach b . The outer-most ambient, which we denote by $@$, attempts to access the secret by an $\text{open } \text{msg}$. Once inside b , the wrapper mail is opened and b reads the secret D . For the process SYS_1 we want to guarantee that

Figure 6.2: Initial configuration of SYS_1 .

(UA) no untrusted ambients can access D .

□

Example 6.3.2. The following example was presented in [14] where multi-level security for Mobile Ambients is investigated. *Boundary* ambients are introduced to protect high-level information. The restriction is that high-level data must be contained either in boundary ambients or in low level ambients not escaping boundaries. The authors consider the following system:

$$SYS_2 = m[send[out m.in b | hdata[in filter]]] | b[open send] | filter[in send] | open filter.$$

Boundary ambients are b and $send$. The security property we want this system to satisfy is:

(BA) $hdata$ is always within boundary ambients or if $hdata$ is within the low level ambient $filter$ then $filter$ is in a boundary ambient.

□

In the following, we want to develop finite-state models for mobile ambients in order to verify properties such as (UA) and (BA) by model checking.

6.3.2 HABA modelling approach

Modelling issues. In modelling mobile ambients by HABA (with references) we want to exploit cardinalities of entities in order to code multiple instances of the same ambient. However, there are some issues related with the use of unbounded entities and with the possibility to express properties about systems.

- Since unbounded entities impose restrictions on reallocations from state to state (see Definition 5.3.12), we should make sure that our model conforms with the definition of reallocation. For example, unbounded entities cannot be fresh, therefore they must always be in the codomain of the reallocation.

- Due to replication, any ambient can have an unbounded number of copies. In $\mathcal{Nall}TL$ we need some mechanism to distinguish among those different instances so that given a certain entity it is clear which copy of the ambient it stand for.

As a possible solution we propose the following model developed in this section.

Basic idea of the model. Along the line of [40, 57, 75, 86], the essential information we want to retrieve from a mobile ambient process P is:

which ambients may end up in which other ambient

To model the structure of a process P we introduce a classification among the entities in use. For any ambient a occurring in P we have:

- a special entity $\text{ho}(a)$ (called a 's *host*) that is used to record, at any point in the computation, the ambients (hosted) inside *any* copy of ambient a . It is fixed, i.e., during the computation its position within the topology of the process does not change.
- A concrete entity, say e , distinct from $\text{ho}(a)$, that is used to represent an instance of the ambient a that executes capabilities. e moves according to the capabilities of that particular copy of a . The association between e and a will be clarified below. If there exist several instances of a , some of them may be represented by multiple/unbounded entities (again distinct from $\text{ho}(a)$). Before one single instance among those represented by multiple/unbounded entity performs a capability it is extracted. This is somehow similar to the semantics of $!Q$ which cannot be reduced until it has been expanded to $Q \mid !Q$.

Example 6.3.3. Figure 6.3(a) shows a possible state of a process. Figure 6.3(b) depicts how this process is represented in our model. In this case we have three different ambients a , b and n . Outgoing references define the son/father relation μ . Notation $e:n$ says that e denotes an ambient with name n . The host $\text{ho}(a)$ keeps track of the ambients contained in *any* copy of an ambient a . Thus we have a copy of n contained in a because e_1 is concrete and more than M copies of the ambient b because e_2 is unbounded¹. Ambient b does not contain any other ambients since $\text{ho}(b)$ does not have incoming references; and n contains another copy of b . Hosts entities are depicted as squares in order to distinguish them from entities that can move around during the computation (depicted as circles). Figure 6.4 shows the state resulting after e_1 has executed the capability in b . This configuration detects that n is contained in a copy of ambient b although we do not distinguish which one. \square

¹Here, the parameter M has the same as in Chapter 5.

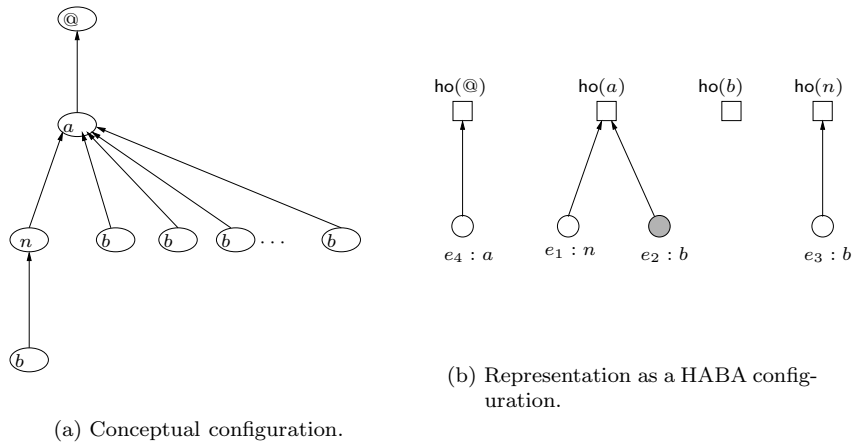


Figure 6.3: Example of ambient process and its coding.

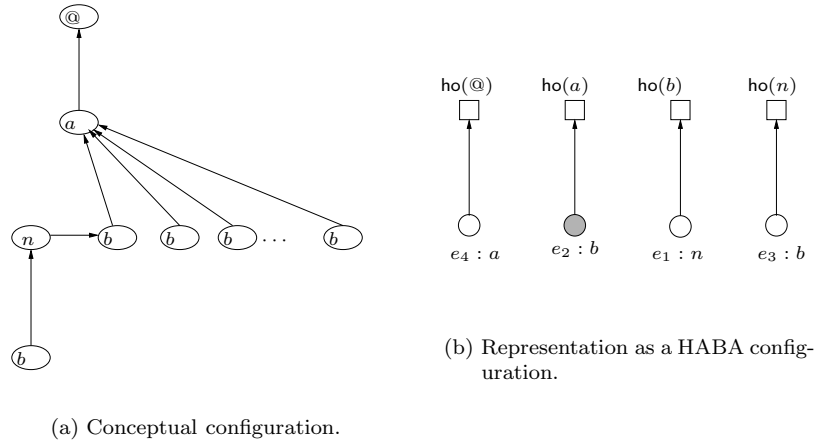


Figure 6.4: The model of Fig 6.3 after $e_1:n$ has executed in b .

6.3.3 Process indexing

First of all we assume that the names occurring bound inside restriction are all distinct from each other and from the free names. This can always be done by α -conversion.

In order to achieve a more precise analysis we label the ambients occurring in the process P by distinguished indexes. Let J be a countable set of indexes and $\tilde{\mathcal{N}} = \{n_j \mid n \in \mathcal{N}, j \in J\}$ be the set of indexed names. A *indexed process* \tilde{P} is a process such that if $n[Q]$ occurs in \tilde{P} then $n \in \tilde{\mathcal{N}}$. Let $noi(P) : \mathbf{Proc} \rightarrow \mathbf{Proc}$ be function that given an indexed process \tilde{P} , returns the (non-indexed) process P obtained from \tilde{P} by stripping out every index from the ambient names occurring in \tilde{P} . The process \tilde{P} is an indexed version of P , if $noi(\tilde{P}) = P$. Moreover, $idx(\tilde{P}) = \{j \in J \mid n_j \text{ occurs in } \tilde{P}\}$ is the set of indexes occurring in \tilde{P} . We need a notion of well-indexing for processes that is introduced by the following definition.

Definition 6.3.4. Process \tilde{P} is *well-indexed* if and only if

- $P = \mathbf{0}$;
- $P = n_i[Q]$ and Q is well-indexed and $i \notin idx(Q)$;
- $P = Q \mid Q'$, and Q, Q' are well-indexed and $idx(Q) \cap idx(Q') = \emptyset$;
- $P = !Q$ and Q is well-indexed;
- $P = M.Q$ and Q is well-indexed;
- $P = (\nu n)Q$ and Q is well-indexed.

Note that indexed names neither occur in capabilities, nor in name restriction.

Example 6.3.5. Let

$$P = a[\text{in } b.a[\text{out } b]] \mid b[\mathbf{0}].$$

A well-indexed instance of P is

$$\tilde{P} = a_1[\text{in } b.a_2[\text{out } b]] \mid b_3[\mathbf{0}].$$

We use indexing in order to distinguish between copies of identically named ambients that have different behaviour as for example the two instances of a . For simplicity (without loss of generality) we assume that process indexing is done always after α -conversion.

In the following we assume that every process has been well-indexed. This can be considered as a preprocessing step before the extraction of the model.

6.3.4 Preliminary notation

We assume the existence of a global function that associate to every entities e a name of the ambients in well-indexed process \tilde{P} represented by e .

$$A : Ent \rightarrow n(\tilde{P}) \quad (6.4)$$

The function A provides a partitioning of Ent . For $e \in Ent$, we write $e:n$ as a shorthand for $A(e) = n$. In the following we often make use of this notation in some contexts if it is necessary to know the ambient of an entity. For example, in a function with parameter e , we write $f(e:a)$ if it is essentially to know that e represents a copy of the ambient a . Moreover, we write $e:a \in E$ is a shorthand for $e \in E \wedge A(e) = a$.

We consider two special sets of entities $E_{n(P)}^{is} \cap E_{n(P)}^{ho} = \emptyset$; where:

- entities $E_{n(P)}^{is} = \{is(n) \in Ent \mid n \in n(P)\}$ are used for two purposes:
 - first of all to model ambient names. They are the corresponding of the special entities PV used for program variables in Chapter 5. In $\mathcal{M}allTL$ we will have special variables x_n whose interpretation is $is(n)$ and therefore we can easily refer to ambient n by $x_n.a$.
 - Secondly, for any ambient $n \in n(P)$, $is(n)$ is the *inactive site* of n (from which the name is), i.e., the repository where the copies of n are placed when this ambient is inactive. Informally speaking, inactive means that n cannot execute any action and it is not yet visible to other ambients (a complete exposition of this concept is postponed till Section 6.3.7).

An entity $is(n)$ does not move and we assume $A(is(n)) = n$.

- $E_{n(P)}^{ho} = \{ho(n) \mid is(n) \in n(P)\}$, is a set of special host entities. We assume $A(ho(n)) = n$. In every state of the model and for every ambient n we have $is(n)$ point to $ho(n)$.

Note that for a indexed process \tilde{P} , in $E_{n(\tilde{P})}^{is}$ there exist distinct instances of $is(n_i)$, $is(n_j)$ and $ho(n_i)$, $ho(n_j)$ where $i \neq j$ for different occurrences of the ambient n in P . Every HABA state considered in this chapter for modelling mobile ambients is of the form:

$$q = \langle \gamma, P \rangle.$$

The first component $\gamma = (E, \mu, \mathcal{C}) \in \text{CONF}$ is the standard configuration of HABA states (with references) as defined in Definition 5.3.21. This implies that much of the notation used in Chapter 5 is reused here, as for example the constant M denoting the upper bound on the precision of the cardinality function \mathcal{C} . For γ we write E^{fix} for its set of fixed entities $E^{\text{fix}} = E \cap (E_{n(P)}^{ho} \cup$

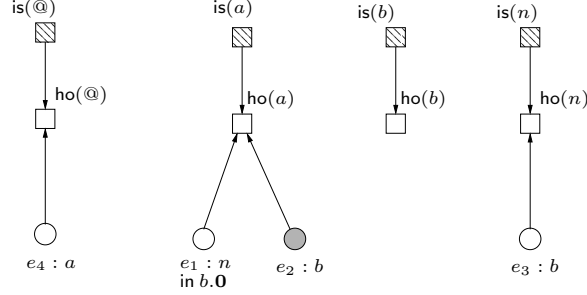


Figure 6.5: HABA state representation for the model of Fig 6.3(b) with entities modelling inactive sites processes to be executed.

$E_{n(P)}^{\text{is}}$) and E^c for its set of entities representing the copies of ambients that can move, i.e., $E^c = E \setminus E^{\text{fix}}$. The second component P has type

$$P : Ent \rightarrow 2^{\text{Proc}}$$

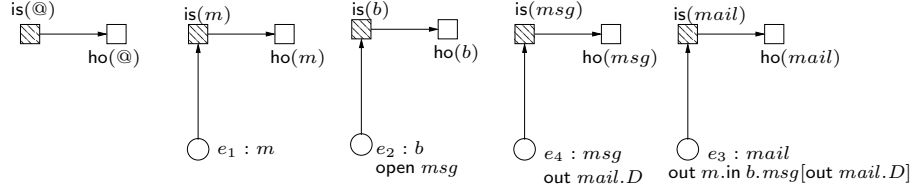
and is introduced for the special case of mobile ambients. $P(e)$ associates to the entity e the set of processes that e must execute.

Example 6.3.6. The typical state we obtain using the previous sets of special entities and the component P is depicted in Figure 6.5. The set $P(e)$ is written close to e . We do not write it if it is only the empty process. For example, entity e_1 has to execute $\{\text{in } b.\mathbf{0}\}$, whereas the other entity only $\{\mathbf{0}\}$. The inactive sites are always depicted as patterned square in order to distinguish them from host entities. \square

6.3.5 Pre-initial and initial state: an overview

Pre-initial state. We have already discussed that one of the problem in modelling mobile ambients (in HABA) is how to specify that a particular entity e represents an ambient n and if there are more than one instance of n how to distinguish among these instances: the global function $A(e) = n$ cannot be used in $\mathcal{M}\ell\text{TTL}$. The purpose of the *pre-initial* state is to address this problem. The pre-initial state is a special state we add to the model with the aim to identify for every entity which ambient it represents. The pre-initial state of a process P is built in such a way that every entity representing a copy of the ambient n leads to the inactive site $\text{is}(n) \in E_{n(P)}^{\text{is}}$. The structure of the graph does not reflect the initial topology described by P . When we specify formulae in the logic we will exploit the fact that an entity e in the pre-initial state leads to $\text{is}(n)$ to express the fact that e is an entity representing a copy of the ambient n .

Example 6.3.7. Figure 6.6 depicts the pre-initial state of the process SYS_1 in Example 6.3.1. The important point to note for the moment in this figure

Figure 6.6: Pre-initial state of the process SYS_1 of Example 6.3.1.

is that every copy of an ambient leads to the corresponding entity in $E_{n(P)}^{is}$. For example e_1 that stands for m leads to $is(m)$. Although e_1 has the label m (i.e., $A(e_1) = m$), this information cannot be exploited in the logic. However, in $\mathcal{N}allTL$ we can refer to $is(m)$. Hence, due to the pre-initial state, we can use the formula $\exists x : x \rightsquigarrow x_m$ in order to identify x as an copy representing the ambient m . \square

Example 6.3.8. The security property (UA) of Example 6.3.1 is violated if and only if the following $\mathcal{N}allTL$ formula is satisfied

$$\phi \equiv \exists x : x \rightsquigarrow x_{msg} \wedge F(x \not\rightarrow x_{msg} \wedge x.a \neq mail \wedge x.a \neq b).$$

As usual, $mail$ is a shorthand for $x_{mail}.a$ and b for $x_b.a$. ϕ states that msg eventually will be included inside an ambient different from $mail$ and b (which are the only trustworthy ones) violating therefore the security property (UA). The property (BA) of Example 6.3.2 is violated if and only if:

$$\psi \equiv \exists x : x \rightsquigarrow x_{hdata} \wedge \exists y : y \rightsquigarrow x_{filter} \wedge F(x \not\rightarrow x_{hdata} \wedge x.a \neq send \wedge x.a \neq b \wedge (x.a = filter \Rightarrow y \not\rightarrow x_{filter} \wedge y.a \neq b \wedge y.a \neq send))$$

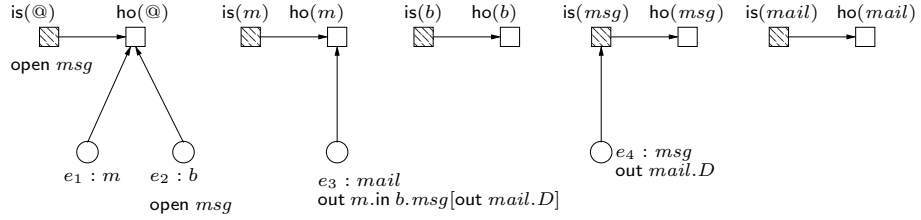
where $send \equiv x_{send}.a$, $b \equiv x_b.a$ and $filter \equiv x_{filter}.a$. ψ states that eventually $hdata$ escapes the boundary ambients b and $send$, and if it is inside $filter$ this is not protected by one of the two boundaries.

Hence, if \mathcal{H}_{SYS_1} and \mathcal{H}_{SYS_2} are the HABA modelling SYS_1 and SYS_2 , the security properties are guaranteed to hold if we verify

$$\mathcal{H}_{SYS_1} \not\models \phi \quad \text{and} \quad \mathcal{H}_{SYS_2} \not\models \psi.$$

That can be checked using the model checking algorithm defined in Chapter 5. \square

Initial state. The pre-initial state does not reflect the initial topology defined by the process P . This is done by the *initial-state* whose purpose is in fact to model the son/father relation described by the process in terms of entities and references between them. For example, Figure 6.7 shows the initial state of the process SYS_1 of Example 6.3.1. Ambients m and b are inside the outer-most

Figure 6.7: Initial state of the process SYS_1 of Example 6.3.1.

ambient $@$, whereas $mail$ is inside m . Ambients b and $mail$ are empty. $is(msg)$, $ho(msg)$ and e_4 maintain the same pointer structure as the pre-initial state since in the beginning ambient msg cannot execute any action (we say that it is *inactive*). In fact in SYS_1 , msg is guarded by $out\ m.in\ b$ (cf. the process to be executed by $e_3:mail$ in Figure 6.7 or see the definition of SYS_1). Only when both $out\ m$ and $in\ b$ have been consumed msg is enabled to execute its actions (it becomes *active*). In a HABA state we model inactive ambients by letting the copies of the ambient lead to their inactive site. Only if an ambient is inactive its inactive site has some entity leading to it. Apart from the outer-most ambient $@$, in the pre-initial state every ambient is inactive by definition (cf. Figure 6.6). The formal, definition of active, and inactive are postponed till Section 6.3.7. Finally, note that the ambient $@$ does not have a real instance (it is modelled only by $is(@)$ and $ho(@)$), therefore we delegate $is(@)$ the execution of $open$ and $!$.

6.3.6 On morphisms and canonical form for mobile ambients

In Chapter 5, where we have defined the symbolic operational semantics of \mathcal{L}_n , we introduced the notion of canonical form for HABA configurations. In this chapter, we reuse that concept and show that it is suitable also for the case of mobile ambients.

Assumption on morphisms. The employment of $E_{n(P)}^{is}$ for the same purpose of the program variables PV in Chapter 5 entails the same kind of assumptions on morphisms and reallocations. In particular, throughout this chapter we consider only morphisms that satisfy the following conditions:

$$\gamma_1 \xrightarrow{h} \gamma_2 \Rightarrow h \upharpoonright E_{n(P)}^{is} = id_{E_{n(P)}^{is}} \quad (6.5)$$

$$\gamma_1 \xrightarrow{\lambda} \gamma_2 \Rightarrow \forall e \in E_{n(P)}^{is} : \lambda(e, e) = 1 \quad (6.6)$$

$$q \xrightarrow{h} q' \Rightarrow \forall e \in E_{\gamma_q} : (P_q(e) = P_{q'}(h(e)) \wedge A(e) = A(h(e))) \quad (6.7)$$

Conditions (6.5) and (6.6) correspond to those applied in Chapter 5. They force the correspondence of the program variables in configurations related by

morphisms or reallocations. Condition (6.7) is typical for the ambient calculus and simply forces morphisms to map an entity e only onto another entity e' representing the same ambient and executing the same process.

The notions of L -safety, L -compactness and canonical form developed in Chapter 5 can be adapted for Mobile Ambients in a straightforward manner.

Definition 6.3.9 (L -safety for MA). Let $L > 0$. A configuration γ with $E_{n(P)}^{\text{ho}} \subseteq E_\gamma$ is L -safe if

$$\forall e \in E_{n(P)}^{\text{ho}} : (\forall e' : d(e', e) \leq L \Rightarrow \mathcal{C}_\gamma(e') = 1).$$

Definition 6.3.10 (L -compactness for MA). Let $L > 0$. A configuration γ with $E_{n(P)}^{\text{ho}} \subseteq E_\gamma$ is L -compact if

$$\forall e \in E_\gamma : (\text{indegree}(e) > 1 \vee \exists e' \in E_{n(P)}^{\text{ho}} : d(e, e') \leq L + 1).$$

The only difference with the L -safety and L -compactness of Chapter 5 is that here we consider the distance between a generic entity and a host one whereas there we regarded the distance from a program variable to a generic entity.

Definition 6.3.11 (canonical form for MA). A configuration γ is L -canonical (or in L -normal form) if γ is L -safe and γ is L -compact.

As we have seen in Chapter 5, using only states in canonical form has a double advantage: on the one hand, it is possible to determine precisely which entities are involved in a pointer update; on the other hand, as we will see, the canonical form helps to obtain a finite-state HABA.

Dealing with multiple instances of an ambient. We represent multiple copies of the same ambient by multiple/unbounded entities. Due to canonical form, multiple/unbounded entities are not direct children of hosts: there are L concrete entities in between. However, both the multiple/unbounded entity and the preceding concrete entities represent different copies of the *same* ambient. Hence, all these entities are assumed to be at the same level, i.e., inside the same ambient. This is according to our initial aim to collect information about what is contained at top level for every ambient whereas we do not care about inner levels. With this model we are able to distinguish that inside an ambient, say a , there are no instance of the ambient b ; or there are precisely i instances of b with $1 \leq i \leq L + M$; or there are more than $L + M$ instances of b where L and M are the usual parameters that properly tuned can be exploited to accomplish a more precise model. Furthermore, the use of canonical form ensures that the model is always finite.

Example 6.3.12. Assuming $M = 1$, in the ambient a depicted in Figure 6.8, there are *exactly* two instances of the ambient n and *any* number of copies strictly greater than 4 of the ambient b . Between copies of the same ambient, we depicted dashed horizontal arrows to stress that, at the conceptual level, these arrows do not describe a son/father relation as the solid vertical ones. \square

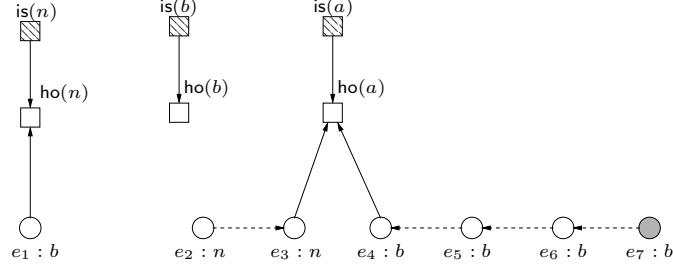


Figure 6.8: Representing multiple instances of the same ambient.

6.3.7 Coding processes into HABA configurations

In this section we define a function that given a process P codes it into a HABA state, i.e., it returns:

- a configuration γ that model the ambients topology delineated by P ;
- a function \mathcal{P} that associates to every entity in the configuration the set of capabilities it must perform.

For doing this, we first need few auxiliary definitions.

Configuration union. For a configuration γ , let $\ell_\gamma^a(e)$ be the longest pure chain of a copies in γ leading to (and including) e . That is

$$\ell_\gamma^a(e) = \{e' \in E_\gamma^c \mid A(e') = a, e \in \mu^*(e')\} \cup \{e\} \quad (6.8)$$

For example in Figure 6.8, we have $\ell_\gamma^n(\text{ho}(a)) = \{e_2, e_3, \text{ho}(a)\}$ and $\ell_\gamma^b(\text{ho}(a)) = \{e_7, e_6, e_5, e_4, \text{ho}(a)\}$ and $\ell_\gamma^b(\text{ho}(b)) = \{e_1, \text{ho}(b)\}$.

For configurations γ, γ' with distinct sets of non-fixed entities and with equal pointer structure as well as cardinality function over common entities, i.e., such that:

$$\begin{aligned} E_\gamma^c \cap E_{\gamma'}^c &= \emptyset \\ \mathcal{C}_\gamma \upharpoonright (E_\gamma \cap E_{\gamma'}) &= \mathcal{C}_{\gamma'} \\ \mu_\gamma \upharpoonright (E_\gamma \cap E_{\gamma'}) &= \mu_{\gamma'} \end{aligned}$$

we define the union configuration $\gamma \uplus \gamma' = (E, \mu, \mathcal{C})$ where:

$$\begin{aligned} E &= E_\gamma \cup E_{\gamma'} \\ \mu(e) &= \begin{cases} \mu_\gamma(e) & \text{if } e \in E_\gamma \\ \mu_{\gamma'}(e) & \text{if } e \in E_{\gamma'}^{\text{fix}} \\ \text{first}(\ell_\gamma^a(\mu_{\gamma'}(e))) & \text{if } e : a \in E_{\gamma'}^c \wedge \mu_{\gamma'}(e) \in E_{\gamma'}^{\text{ho}} \\ \mu_{\gamma'}(e) & \text{otherwise} \end{cases} \\ \mathcal{C}(e) &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e \in E_\gamma \\ \mathcal{C}_{\gamma'}(e) & \text{if } e \in E_{\gamma'} \setminus E_\gamma \end{cases} \end{aligned}$$

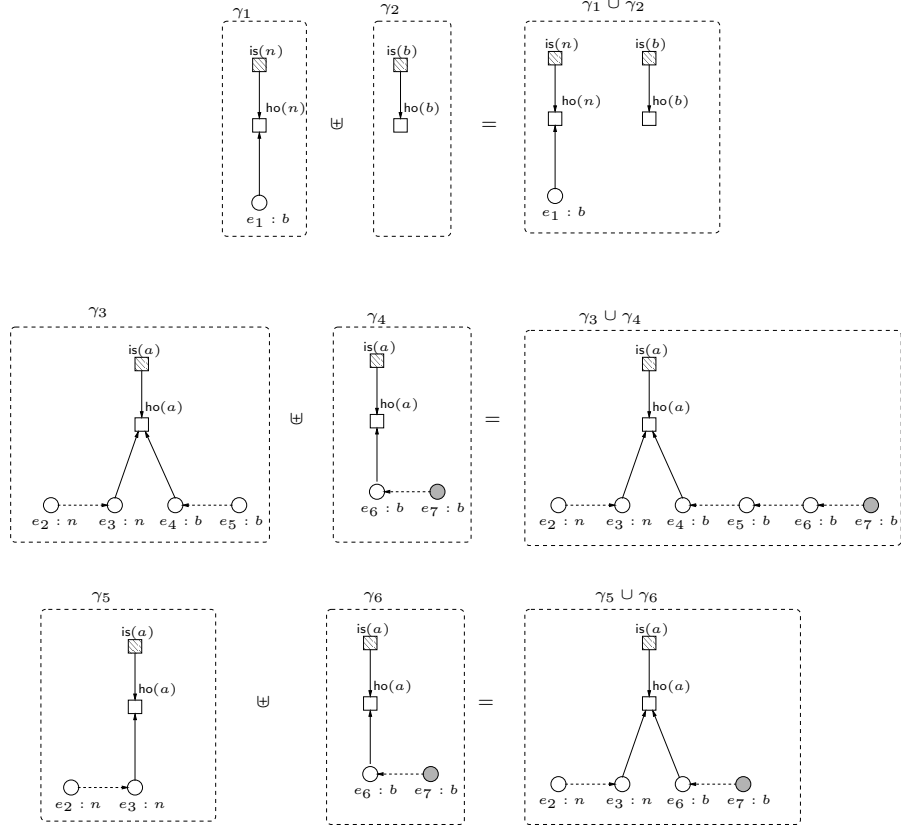


Figure 6.9: Example of configuration unions.

The definition of E is straightforward. \mathcal{C} is well defined since \mathcal{C}_γ and $\mathcal{C}_{\gamma'}$ are equal on the intersection of their domains. For $e:a \in E_{\gamma'}$, $\mu(e)$ assign the first entity in the queue of copies of a .

Example 6.3.13. Some examples of configuration unions are depicted in Figure 6.9. If both configurations have copies of an ambient, say b , inside the same ambient, say a , the union appends the copies of the second configuration to those of the first one. In the figure, this is exemplified in $\gamma_3 \uplus \gamma_4$. \square

To accomplish union of states we need also to define a notion of union for the component P of the state. This is done point-wise as follows: let $q = \langle \gamma, P \rangle$ and $q' = \langle \gamma', P' \rangle$ be two states and $e \in E_\gamma \cup E_{\gamma'}$, then

$$(P \uplus P')(e) = \begin{cases} P(e) \cup P'(e) & \text{if } e \in E_\gamma \cap E_{\gamma'} \\ P(e) & \text{if } e \in E_\gamma \setminus E_{\gamma'} \\ P'(e) & \text{if } e \in E_{\gamma'} \setminus E_\gamma \end{cases} \quad (6.9)$$

and finally we can easily define the union for states component-wise:

$$\langle \gamma, P \rangle \uplus \langle \gamma', P' \rangle = \langle \gamma \uplus \gamma', P \uplus P' \rangle \quad (6.10)$$

Subprocesses executed by ambients. In order to construct the component P of the state we need the following function $\rho : \mathbf{Proc} \rightarrow 2^{\mathbf{Proc}}$ that given a process P returns the set of sub-processes that the ambient containing P must execute. It is defined by:

$$\begin{aligned} \rho(\mathbf{0}) &= \emptyset & \rho(M.Q) &= \{M.Q\} \\ \rho(Q \mid Q') &= \rho(Q) \cup \rho(Q') & \rho(m[Q]) &= \emptyset \\ \rho(!Q) &= \{!Q\} & \rho((\nu n)Q) &= \rho(Q) \end{aligned}$$

By the previous definition, processes belonging to nested ambients are not returned. Note that because of the assumption on the bound names we can delete restriction.

Example 6.3.14. Consider our running example SYS_1 . We have:

$$\rho(\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]) = \{\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]\}$$

This will be used to define the process that must be executed by the entity representing the copy of *mail*. Moreover, since the ambient m contains only $\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]$ the copy of m does not execute anything, in fact $\rho(\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]) = \emptyset$. \square

Enabled ambients. An *enabled* ambient is an ambient which is ready to perform some action. $\text{enab}(P)$ denotes the set of all enabled ambients in P . It is defined as:

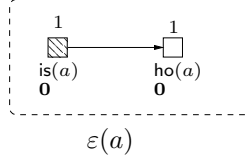
$$\begin{aligned} \text{enab}(\mathbf{0}) &= \emptyset & \text{enab}(M.Q) &= \emptyset \\ \text{enab}(Q \mid Q') &= \text{enab}(Q) \cup \text{enab}(Q') & \text{enab}(a[Q]) &= \{a\} \cup \text{enab}(Q) \\ \text{enab}(!Q) &= \text{enab}(Q) & \text{enab}((\nu n)Q) &= \text{enab}(Q). \end{aligned}$$

Example 6.3.15. Consider SYS_1 of Example 6.3.1. The set $\text{enab}(SYS_1) = \{m, \text{mail}, b\}$. Note that ambient *msg* is not enabled (yet). \square

The classification of enabled and non-enabled ambients entails a corresponding representation on the HABA configurations that represents the process. We have already anticipated that in the model we classify an ambient a according to the existence of entities leading to the inactive site. Therefore, in the model we have the following (semantical) notion that corresponds to (the syntactic one of) being enabled.

Definition 6.3.16. In state q , the ambient n is *active* if $\nexists e \in E_{\gamma_q} : e \prec_{\gamma_q} \text{is}(n)$.

If n is not active it is called *inactive*. The operational rules use this notion in order to distinguish between ambients that may perform a capability from those that cannot.

Figure 6.10: Graphical representation of $\varepsilon(a)$.

The very basic building block for the construction of the state is the representation of that (sub)part related to the fixed entities of an ambient a , i.e., $\text{is}(a)$, $\text{ho}(a)$. Let $a \in n(P)$:²

$$\varepsilon(a) = \langle \{\text{ho}(a), \text{is}(a)\}, \{(\text{is}(a), \text{ho}(a))\}, \mathbf{1}_{\{\text{is}(a), \text{ho}(a)\}}, \{(\text{ho}(a), \mathbf{0}), (\text{is}(a), \mathbf{0})\} \rangle$$

$\varepsilon(a)$ indicates the *empty (fixed) state for ambient a* and its graphical representation is shown in Figure 6.10. The function $\Omega(a, P, k, \text{act})$ returns a HABA state representing the process P contained inside the ambient a . The parameter k is used for dealing with cardinalities. The parameter act is a boolean that instructs Ω to construct the P configuration with the active or with the inactive representation of its ambients.

Formally, $\Omega : \mathcal{N} \times \mathbf{Proc} \times \mathbb{M}^* \times \mathbb{B} \rightarrow \mathbf{CONF} \times (\mathit{Ent} \rightarrow \mathbf{2}^{\mathbf{Proc}})$ is given by:

$$\begin{aligned} \Omega(a, \mathbf{0}, k, \text{act}) &= \varepsilon(a) \\ \Omega(a, m[Q], k, \text{act}) &= \varepsilon(a) \uplus \Omega(m, Q, k, \text{act}) \\ &\quad \uplus \begin{cases} \langle \{e, \text{ho}(a)\}, \{(e, \text{ho}(a))\}, \{(e, k), (\text{ho}(a), 1)\}, \\ \quad \{(e, \rho(Q)), (\text{ho}(a), \mathbf{0})\} \rangle & \text{if act} \\ \langle \{e, \text{is}(m)\}, \{(e, \text{is}(m))\}, \{(e, k), (\text{is}(m), 1)\}, \\ \quad \{(e, \rho(Q)), (\text{is}(m), \mathbf{0})\} \rangle & \text{otherwise} \end{cases} \\ &\quad \text{where } e:m \text{ is fresh} \end{aligned}$$

$$\begin{aligned} \Omega(a, Q_1|Q_2, k, \text{act}) &= \Omega(a, Q_1, k, \text{act}) \uplus \Omega(a, Q_2, k, \text{act}) \\ &\quad \text{where } E_{\gamma_{\Omega(a, Q_1, k, \text{act})}}^c \cap E_{\gamma_{\Omega(a, Q_2, k, \text{act})}}^c = \emptyset \\ \Omega(a, (\nu n)Q, k, \text{act}) &= \Omega(a, Q, k, \text{act}) \\ \Omega(a, !Q, k, \text{act}) &= \Omega(a, Q, *, \text{act}) \\ \Omega(a, N.Q, k, \text{act}) &= \varepsilon(a) \uplus \Omega(a, Q, k, \text{ff}) \end{aligned}$$

The representation of the empty process inside a is given by the empty active state for a . The representation of $m[Q]$ in a comprehends the empty state for a , the sub-state of Q inside m and a configuration with a non fixed entity

²From now on when convenient we write a state as a four tuple $\langle E, \mu, \mathcal{C}, \mathcal{P} \rangle$, where it is clear that the first three correspond to the configuration. Moreover, in this case, we write functions as set of pairs, i.e., $(e, f(e))$ for $f(e) = e'$.

e standing for the copy of m in a . Depending from the parameter act , this representation can be either the active or the inactive one. Fixed entities have always cardinality 1, whereas entities representing copies, like e , are concrete if their ambient does not occur within the scope of replication. This is recorded in the parameter k which is assigned to e as cardinality. Moreover e has associated the set of capabilities $\rho(Q)$ to be executed. The representation of $Q_1|Q_2$ is given by the union of the representation of Q_1 and Q_2 . When encountered, $\Omega(a, !Q, k)$ makes a recursive call changing the cardinality from k to $*$. This assigns cardinality $*$ to every non-fixed entity within the scope $\Omega(a, Q, *)$. Finally, the representation of $N.Q$ inside a has the empty state for a and the *inactive* representation for the process Q — which since guarded by N contains only non-enabled ambients. Therefore the recursive call $\Omega(a, Q, k, \text{ff})$ is done with the explicit value $\text{act} = \text{ff}$.

Lemma 6.3.17. For all $m \in n(P)$ and Q subprocess of P : if $\Omega(m, Q, k, \text{tt}) = \langle \gamma, P \rangle$ then m is active.

Proof. Straightforward by induction of the structure of Q . \square

Lemma 6.3.18. For every process P ,

$$a \in \text{enab}(P) \Rightarrow a \text{ is active in } \Omega(@, P, 1, \text{tt}).$$

Proof. Straightforward by induction of the structure of P and by the previous lemma. \square

Example 6.3.19. Figure 6.11 shows the HABA state representation for

$$\Omega(@, b[\text{open } \text{msg}], 1, \text{tt})$$

as well as

$$\Omega(@, m[\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]], 1, \text{tt}).$$

In the latter, note the different representation between active ambients ($@, m, \text{mail}$) and inactive (msg). Composing the two states (by the union operation) together with $\Omega(@, \text{open } \text{msg}, 1, \text{tt})$ we obtain:

$$\begin{aligned} & \Omega(@, b[\text{open } \text{msg}], 1, \text{tt}) \uplus \Omega(@, \text{open } \text{msg}, 1, \text{tt}) \uplus \\ & \Omega(@, m[\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]], 1, \text{tt}) = \Omega(@, \text{SYS}_1, 1, \text{tt}). \end{aligned}$$

The state $\Omega(@, \text{SYS}_1, 1, \text{tt})$ is depicted in Figure 6.7³. \square

Example 6.3.20. Figure 6.12 contains an example state in case of replication. By definition we have $\Omega(@, !n[\text{in } n], 1, \text{tt}) = \Omega(@, n[\text{in } n], *, \text{tt})$ therefore, the entity e_2 modelling the copies of n , becomes unbounded. \square

³Except for the capability of $\text{is}(@)$ that is be dealt with in a special way, see Definition 6.3.21.

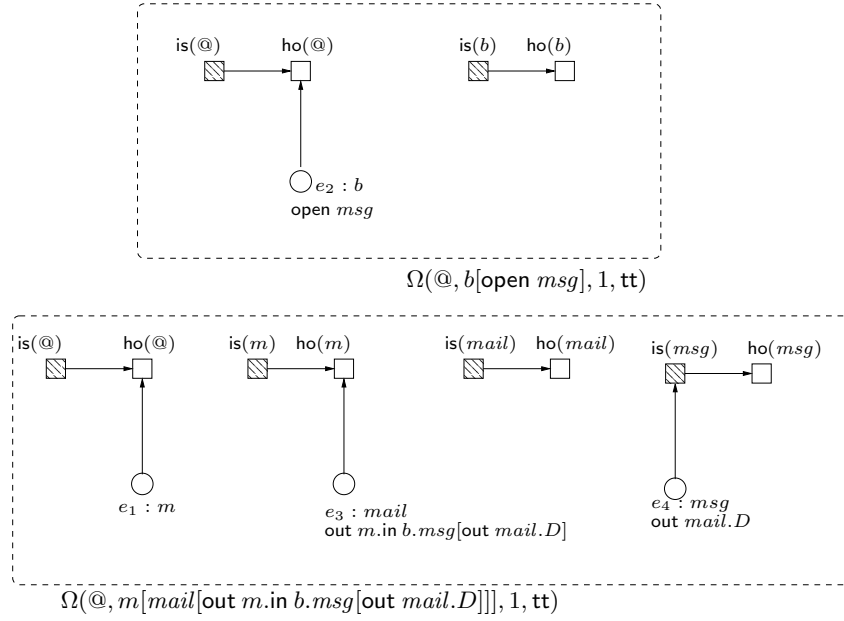


Figure 6.11: HABA states (up to isomorphism) returned by $\Omega(@, b[\text{open } msg], 1, \text{tt})$ and $\Omega(@, m[\text{mail}[\text{out } m.\text{in } b.\text{msg}[\text{out } \text{mail}.D]]], 1, \text{tt})$.

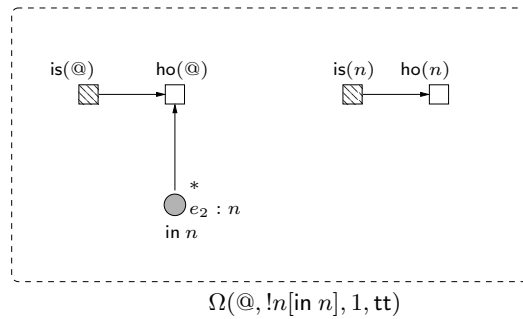
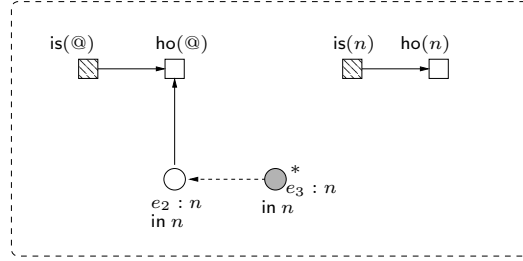


Figure 6.12: HABA state (up to isomorphism) returned by $\Omega(@, !n[\text{in } n], 1, \text{tt})$.

Finally, the next definition give the encoding of a process in a L -canonical HABA state. $\Omega(@, P, 1, \text{tt})$ does not necessarily return a canonical state. That happens only in some circumstances involving unbounded entities. In order to get a canonical state, $\Omega(@, P, 1, \text{tt})$ is first expanded to a safe state and then contracted by the canonical form.

Definition 6.3.21 (Process encoding). The *process encoding* function \mathcal{D} :

Figure 6.13: 1-canonical representation of $\Omega(@, !n[in n], 1, tt)$.

$\mathbf{Proc} \rightarrow \mathbf{CONF} \times (\mathbf{Ent} \rightarrow 2^{\mathbf{Proc}})$ is defined by:

$$\mathcal{D}(P) = \langle \text{cf}(\gamma'), P_{\Omega(@, P, 1, tt)} \{ \rho(P) / \text{is}(@) \} \rangle$$

where $h : \gamma' \xrightarrow{\text{ho}} \gamma_{\Omega(@, P, 1, tt)}$ is a contractive morphism such that $\forall e \in E_{\gamma'}^* : |h^{-1}(e)| = L + M + 1$ and γ' is L -safe.

Note that the condition on h corresponds to require that the shrink factor in *every* unbounded entity is precisely $L + M + 1$, or in other words, every unbounded entity is replaced by a chain of $L + M + 1$ entities. By the canonical form of γ' , these chains will be reduced to chains with L concrete entities pointing to an entity in $E_{n(P)}^{\text{ho}}$ (because of L -safety), followed by an unbounded entity (since there are $M + 1$ remaining entities).

For example, the state in Figure 6.12 is not L -canonical for any $L > 0$. The canonical form automatically provides us with the correct representation needed for the simulation of P 's behaviour. Figure 6.13 depicts the resulting 1-canonical state obtained after the application of the canonical form to a safe configuration γ' where the unbounded entity e_2 is expanded in $L + M + 1$ concrete entities. Note that for any $M > 0$ there is only one such γ' (up to isomorphism), moreover the canonical form is also unique up to isomorphism by Theorem 5.6.16.

The definition \mathcal{D} assigns to the inactive site $\text{is}(@)$ the set $\rho(P)$ containing the capabilities to be executed by $@$.

6.3.8 Pre-initial and initial state construction

In Section 6.3.5 we have informally introduced the idea and the motivation of pre-initial and initial state. The function Ω , and \mathcal{D} provide us with the main ingredients necessary for the definition of these states.

For any process P , its pre-initial state is given by:

$$q_{pre} = \Omega(@, P, 1, \text{ff}) \tag{6.11}$$

References among entities are defined — by the application of $\Omega(@, P, 1, \text{ff})$ — in order to follow the (inactive) scheme introduced informally in Section 6.3.2

(in particular see Figure 6.6). From the previous definition we have that in the pre-initial state every ambient in P is inactive.

Given the mapping \mathcal{D} , the definition of initial state is straightforward :

$$q_{in} = \mathcal{D}(P). \quad (6.12)$$

q_{pre} performs the transition $q_{pre} \rightarrow_{\lambda_{pre}} q_{in}$ where λ_{pre} is defined by:

$$\lambda_{pre} = h_{cf} \circ h^{-1}(\gamma_{pre} \xrightarrow{id} \gamma_{\Omega(@,P,1,tt)}). \quad (6.13)$$

The morphism $h : \gamma' \rightarrow \gamma_{\Omega(@,P,1,tt)}$ is the one given in the definition of $\mathcal{D}(P)$. This definition corresponds to λ used in the assignment rule for the symbolic semantics in Table 5.6.4 (cf. Section 5.6.4). The same definition of reallocation will be used in the operational rules for the execution of capabilities. The correspondence to the reallocation of the assignment rule for the language \mathcal{L}_n is not surprising since the execution of capabilities correspond to manipulation of pointers and therefore it is essentially an “atomic” sequence of assignments. Proposition 5.6.20 ensures that λ is indeed a reallocation.

6.3.9 Configuration link manipulations

In the definition of the operational model, the computation of a process P corresponds to specific pointer manipulations that mimic the movements of P ambients.

Activating ambients. In Section 6.3.7, we have seen that active ambients in a HABA state modelling P correspond to enabled ambients in P . Informally, if an ambient is enabled means that one of its instances can execute capabilities. During the computation if an ambient a executes the capability N of the process $N.Q$, since Q is no longer guarded by N , some ambients in Q may become enabled. In the HABA state these ambients must be activated in order to have a consistent representation of the process. Here we define a function $act_{Q,a}(\gamma)$ that modifies the pointer structure of γ in such a way that every enabled ambients of the process Q rooted in a becomes active.

$$act_{Q,a}(\gamma) = (\gamma \setminus \gamma_{\Omega(a,Q,1,ff)}) \cup \gamma_{\Omega(a,Q,1,tt)}.$$

The activation process consists of replacing the part of the configuration related to Q with inactive representation (i.e., $\Omega(a, Q, 1, ff)$) by the configuration where the enabled ambients of Q are active, i.e., $\Omega(a, Q, 1, tt)$.

Proposition 6.3.22.

$$\forall m \in n(Q) : (m \in enab(Q) \Leftrightarrow m \text{ is active in } \Omega(a, Q, 1, tt)).$$

Proof. Straightforward by induction on the structure of Q . \square

Example 6.3.23. Figure 6.14 depicts the activation of the ambient m by the outer-most ambient $@$. It corresponds to $act_{m[0],@}(\gamma)$. \square

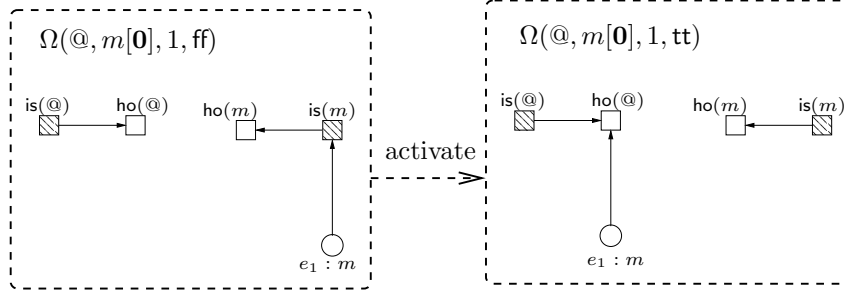


Figure 6.14: Rearrangements of pointers performed by $act_{m[0],@}(\gamma)$ activating m .

Moving ambients. The execution of capabilities `in` and `out` involves moving entities from one host entity to another. In our setting, this activity requires some re-adjustments in order to keep a consistent structure of the configuration. For example, several instances of an ambient b contained in a form a queue (cf. Figure 6.8). If another copy $e:b$ enters a , it is enqueued in the first position of the queue. Symmetrically, if a copy of b moves out a (in Figure 6.8 the entity e_4) it must be dequeued and enqueued in the target ambient. We define the function $move(\gamma, e \mapsto \hat{e})$ that, in order to move e inside \hat{e} , manipulates γ according to the consistency requirements just described. $move : \text{CONF} \times \text{Ent} \times \text{Ent} \rightarrow \text{CONF}$ is defined by:

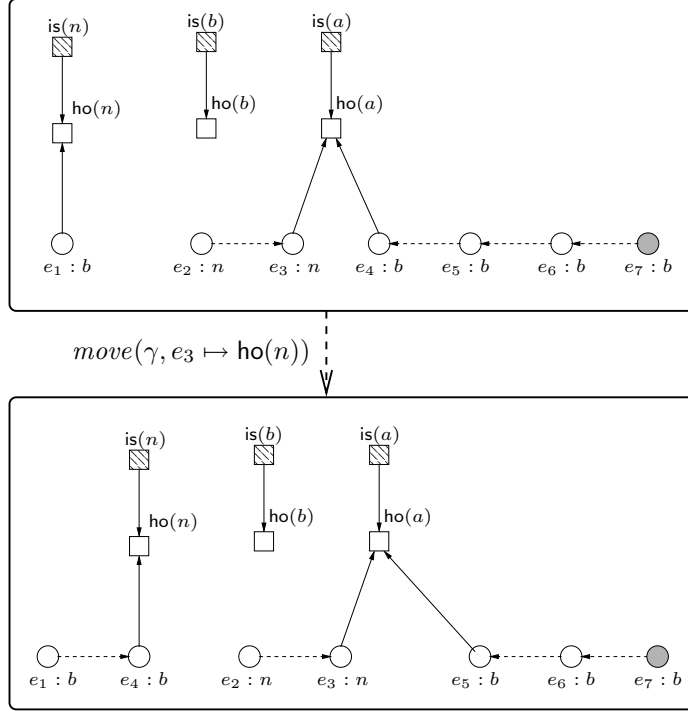
$$move(\gamma, e \mapsto \hat{e}) = (E_\gamma, \mu_\gamma\{\hat{e}/e, \mu_\gamma(e)/\mu_\gamma^{-1}(e), e/e'\}, \mathcal{C}_\gamma)$$

where $\mu_\gamma(e') = \hat{e}$, $A(e') = A(e)$. The entity e moves inside \hat{e} (the first pointers update of μ_γ). The second and the third update concern the consistency discussed above. In particular, if there is a queue starting with an entity $\mu^{-1}(e)$ pointing to e after the transition, $\mu^{-1}(e)$ points to $\mu(e)$ (cf. second link update). Moreover, \hat{e} may already have a queue (with first element e') representing copies of $A(e)$. In this case, e is inserted in the first position (third link update)⁴. Figure 6.15 shows how the configuration changes when e_4 moves inside $ho(n)$.

Updating the state for the In/Out capabilities. There are more modifications to carry out during the execution of capabilities `in` and `out` than those performed by $move$. More specifically, assume there is an entity e that executes $N.Q$ where N is either `in` or `out`. The execution of N requires three kinds of tasks:

- (i) pointer rearrangements moving e from its current location, to the target location and those due for the consistency of the configuration. As we have seen, these updates are performed by $move$.

⁴Note that e' may not exist in which case this update does not take place.

Figure 6.15: Rearrangements of pointers for $move(\gamma, e_4 \mapsto ho(n))$.

- (ii) rearrangements due to the activation the ambients that in Q become enabled.
- (iii) the set of capabilities $P(e)$ should be updated in order to record that e has executed N and that it must continue with Q .

We define the function $IOUp(q, N.Q, e \mapsto \hat{e})$ that manipulates the state q according to (i)–(iii) when e moves inside \hat{e} because of the execution of N and continue with Q . $IOUp : ((CONF \times Ent \rightarrow 2^{Proc}) \times Proc \times Ent \times Ent) \rightarrow (CONF \times Ent \rightarrow 2^{Proc})$ is defined by:

$$IOUp(q, N.Q, e \mapsto \hat{e}) = \langle act_{Q, A(e)}(move(\gamma_q, e \mapsto \hat{e})), P\{(P(e) \setminus \{N.Q\}) \cup \rho(Q)\}/e\} \rangle$$

The configuration of the state is obtained applying first the rearrangements to move e inside \hat{e} (this is done by $move$). After the manipulation of pointers activation takes place. The component $P(e)$ is obtained deleting $N.Q$ and adding the subprocesses to be executed in Q , i.e., $\rho(Q)$.

Dissolving ambients. The next operation manipulates a configuration γ in order to dissolve an ambient. This will be used to define the update of the state for the open capability rule. Informally $dissolve(\gamma, e', e)$ modifies γ in such a way that the ambient of e' , say a , opens the ambient represented by e , say b . The pointers manipulation involving the opening of b consists in the acquisition of its inner ambients by a ; The function $dissolve : (\text{CONF} \times \text{Ent} \times \text{Ent}) \rightarrow \text{CONF}$ is defined by:

$$dissolve(\gamma, e', e:b) = (E_\gamma \setminus \{e\}, \mu_\gamma \{e' / \mu_\gamma^{-1}(e), e' / \mu_\gamma^{-1}(\text{ho}(b))\}, C_\gamma \upharpoonright E_\gamma \setminus \{e\})$$

When called, $dissolve$, the entity e' is a host entity of an ambient say a . γ is updated first of all in order to link to the host e' some possible other copies of b contained in a , i.e., $\mu_\gamma^{-1}(e)$. Secondly, a acquires the ambients nested inside the copy of b (given by e), that is $\mu_\gamma^{-1}(\text{ho}(b))$.

Updating the state for the open capability. As for in and out, we now define the update of the state when open is executed. This involves as in the previous case, a rearrangement of links done by $dissolve$, the activation of the ambients that become enabled and the update of the set of subprocesses that remain to be done by the entity executing open. $\text{OpenUp} : ((\text{CONF} \times \text{Ent} \rightarrow 2^{\text{Proc}}) \times \text{Ent} \times \text{Ent} \times \text{Proc}) \rightarrow (\text{CONF} \times \text{Ent} \rightarrow 2^{\text{Proc}})$ is defined by:

$$\begin{aligned} \text{OpenUp}(q, N.Q, e':a, e) = & \langle \text{act}_{Q,a}(dissolve(\gamma_q, \text{ho}(a), e)), \\ & P\{(P(e') \setminus \{N.Q\} \cup \rho(Q) \cup P(e)) / e'\} \rangle \end{aligned}$$

Note that e' is the entity executing open. Since it is a copy of ambient a , its host is passed as parameter of $dissolve$. However, e' takes the processes $P(e)$ that were supposed to be executed by e .

6.3.10 A HABA semantics of mobile ambients

We can now define the HABA \mathcal{H}_P meant as a symbolic model for the process P .

Definition 6.3.24. Let P be a well-indexed process. The abstract semantics of P is the HABA $\mathcal{H}_P = \langle X_P, S, E, \rightarrow, I, \mathcal{F} \rangle$ where

- $X_P = \{x_n \mid n \in n(P)\} \cup \{x_\otimes\}$;
- $S \subseteq \text{CONF} \times (\text{Ent} \rightarrow 2^{\text{Proc}})$ such that $q_{pre}, q_{in} \in S$, where for state $\langle \gamma, P \rangle \in S$, the component $P(e)$ is the set of processes that must be executed by e .
- $E(\gamma, P) = \langle \gamma, P \rangle$;
- let $\mathcal{R} \subseteq S \times (\text{Ent} \times \text{Ent} \rightarrow \mathbb{M}) \times S$ be the smallest relation satisfying the rules in Table 6.3.10. Then $\rightarrow = \mathcal{R} \cup \{(q_{pre}, \lambda_{pre}, q_{in})\} \cup \{(q, id, q) \mid \neg \exists q', \lambda : (q, \lambda, q') \in \mathcal{R}\}$;

In	$\frac{e \in E^m, \text{ in } b.Q \in P_q(e), \quad b_i \in \text{siblings}(e)}{q \rightarrow_\lambda \text{cf}(\gamma''), P'}$ <p>where $\langle \gamma', P' \rangle = \text{IOUp}(q, \text{in } b.Q, e \mapsto \text{ho}(b_i))$ and $(\gamma'', h) \in \text{SExp}(\gamma')$</p>
Out	$\frac{e \in E^m, \text{ out } b.Q \in P(e), \quad e \prec \text{ho}(b_i) \quad a \in \text{parents}(b_i)}{q \rightarrow_\lambda \text{cf}(\gamma''), P'}$ <p>where $\langle \gamma', P' \rangle = \text{IOUp}(q, \text{out } b.Q, e \mapsto \text{ho}(a))$ and $(\gamma'', h) \in \text{SExp}(\gamma')$</p>
Open	$\frac{e \in E_{\text{a}}^m, \text{ open } b.Q \in P(e), \quad e': b_i \in \text{son}(A(e))}{q \rightarrow_\lambda \text{cf}(\gamma''), P'}$ <p>where $\langle \gamma', P' \rangle = \text{OpenUp}(q, \text{open } b.Q, e, e')$ and $(\gamma'', h) \in \text{SExp}(\gamma')$</p>
Bang	$\frac{e \in E_{\text{a}}^m, \quad !Q \in P(e)}{q \rightarrow_\lambda \text{cf}(\gamma'), P'}$ <p>where $P' = P_q\{(P_q(e) \cup \rho(Q))/e\}$ and $(\gamma', h) \in \text{SExp}(\text{act}_{Q, A(e)}(\gamma_q))$</p>

Table 6.3: Operational rules for Mobile ambients.

- $\text{dom}(I) = \{q_{pre}\}$ and $I(q_{pre}) = \langle \emptyset, \vartheta \rangle$ where $\vartheta(x_n) = \text{is}(n)$ ($n \in n(P)$).
- $\mathcal{F} = \{\{q \in S \mid \exists q', \lambda : q \rightarrow_\lambda q' \Rightarrow q = q'\}\}$.

The set X_P contains a logical variable for each ambient name occurring in P plus x_{a} that is used to refer to the outer-most ambient. The transition relation \rightarrow contains those transitions defined by means of the rules in Table 6.3.10 together with a transition from the pre-initial state and an “artificial” self-loop for each deadlocked state in \mathcal{R} . The set of accept states \mathcal{F} is defined as the set of states which only have a self-loop. Every computation reaches an accept state that is visited infinitely often, fulfilling therefore, the generalised Büchi acceptance condition. The set of initial states contains only the pre-initial state and the interpretation ϑ that allows us to refer to ambient names in $\mathcal{Nal}TL$ formulae.

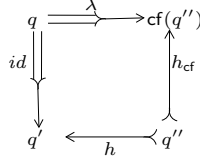


Figure 6.16: Relations among morphisms involved in a transition.

Operational rules. The execution of a capability $N.Q$, in a given state q , applies the following pattern: γ_q is first modified in order to achieve the needed link rearrangements that depends from the capability executed and P_q is updated. This is performed by IOUp (for in and out) or OpenUp (for open).

Applying such pointer updating corresponds to an *id* reallocation from γ to γ' . Because of the rearrangements of the links, γ' may be not canonical. Therefore, we consider its safe expansion, i.e. $\text{SExp}(\gamma')$ (cf. Definition 5.6.18) and for each of its element (γ'', h) we take the canonical form $\text{cf}(\gamma'')$. The overall behaviour, and the relations among morphisms and reallocations just described, can be summarised with the diagram in Figure 6.16. This strategy is the same as the one applied to the rule of the assignment statement in the symbolic semantics in Chapter 5. The reader is referred to that chapter for a full treatment of the topic. For this reason, also the reallocation λ in the rules for in, out, open of Table 6.3.10 is the same as the one of the assignment rule in Table 5.6.4 (cf. Section 5.6.4):

$$\lambda = h_{\text{cf}} \circ h^{-1}(\gamma \xrightarrow{\text{id}} \gamma')$$

Proposition 5.6.20 ensures that λ is a reallocation.

For a configuration γ , the transition relation uses some auxiliary concepts. The entities (in the source state q) that are allowed to move, called *mobile entities*, are indicated by E^m and are defined as:

$$E^m = \{e \in E_q^c \mid A(e) \text{ is active, } \mu_q(e) \in E_{n(P)}^{\text{ho}}\}.$$

Only concrete non-fixed entities modelling an active ambient and directly pointing a host can move. Other concrete entities do not move. In the rules, $E_{\text{@}}^m = E^m \cup \{\text{is}(\text{@})\}$.

Example 6.3.25. For the state represented in Figure 6.8, we have $E^m = \{e_1, e_3, e_4\}$. Entities such as e_2 and e_5 can only move when they are shifted to the beginning of the queue, i.e., when e_3 and e_4 have moved, respectively. \square

Moreover, we use:

$$siblings(e) = \{b_i \mid \exists e': b_i \neq e \wedge (\mu_\gamma(e') = \mu_\gamma(e) \vee \mu_\gamma(e') = e)\}$$

$$son(a) = \{e' \in E_q \setminus E_n^{is}(P) \mid e' \prec_q ho(a)\}$$

$$parents(b_i) = \{a \mid \exists e': b_i \in son(a)\}$$

$siblings(e)$ is the set of ambients having an instance with the same parent of e . $son(a)$ returns the entities that are sons of the ambient a . $parents(b_i)$ is the set of parents of indexed ambients b_i .

Example 6.3.26. For the initial state in Figure 6.7 we have:

$$\begin{aligned} siblings(e_1) &= \{b\} \\ siblings(e_3) &= \emptyset \\ parents(mail) &= \{m\} \\ son(@) &= \{e_1, e_2\}. \end{aligned}$$

□

We explain the rules for the transition relation.

- **In rule:** If a mobile entity e has among its enabled capabilities in $b.Q$ and there exists a sibling ambient b then e moves inside b . Any indexed b_i must be considered.
- **Out rule:** If a mobile entity e executes **out** $b.Q$ and its father is any indexed ambient b_i , i.e. $e \prec ho(b_i)$ then e must move in every ambient containing a copy of b_i .
- **Open rule:** A mobile entity e can execute **open** b , if there exists a $son(A(e))$ e' modelling a copy of b . Entity e' is dissolved and the component $P(e)$ acquires the processes contained in $P(e')$. This is done by the application of **OpenUp**.
- **Bang rule.** If a process $!Q$ is contained in the set of processes that e must execute, then $!Q$ is expanded using the equivalence $!Q \equiv Q|!Q$.

Note that we do not need structural rules for parallel composition, restriction, ambients since a those construct are implicitly represented in the configuration of a state.

Example 6.3.27. The HABA modelling SYS_1 of Example 6.3.1 is depicted in Figure 6.17. as we have seen in that example, for SYS_1 we wanted to check the secrecy property: (UA) “no untrusted ambients can access D ” and expressed by the $\mathcal{N}\ell\ell$ TTL-formula:

$$\phi \equiv \exists x : x \rightsquigarrow x_{msg} \wedge F(x \not\rightsquigarrow x_{msg} \wedge x.a \neq mail \wedge x.a \neq b).$$

No runs of the HABA satisfies ϕ therefore in SYS_1 only b can access the secret data D . \square

Example 6.3.28. Part of the HABA for the process $!n[!in\ n]$ is shown in Figure 6.18 for $L = 1$ and $M = 1$. With this parameters we can distinguish that there are 0,1,2, and more than 2 ambient nested in n at the same time. Increasing the values of L or M we can accomplish more accurate predictions at the cost of increasing the state space of the HABA.

Given a process P , the subpart of the state containing only fixed entities is $\varepsilon(n(P)) = \biguplus_{a \in n(P)} \varepsilon(a)$.

Conjecture 6.3.29. Let \tilde{P} be a well-indexed process. If $noi(P) \rightarrow Q$ then there exists λ and a well-indexed process \tilde{Q}' such that $\mathcal{D}(\tilde{P}) \rightarrow_\lambda (\mathcal{D}(\tilde{Q}') \uplus n(\tilde{P}))$ and $noi(\tilde{Q}') \equiv Q$.

Proof. See Appendix C. \square

Discussion. The abstract semantics for mobile ambients presented in this chapter provides a safe approximation of a process P . Although for many interesting processes the method provides rather precise information, some limitations occur on processes where name restriction is combined with replication. Like other analyses for mobile ambients based on static methods [40, 57, 75, 86], our semantics does not distinguish between processes $!(\nu n)P$ and $(\nu n)!P$ as instead it is done by the standard semantics (cf. observation at page 199). However, the model is able to capture precise information on the number of copies of the same ambients that may be inside another ambient, therefore it distinguish between P and $!P$. The precision can easily be increased.

6.4 Related work

The paper [86] proposes an algorithm detecting process firewalls that are not protective. The technique is based on a control flow analysis that records which ambients may end up inside what other ambients. The analysis does not distinguish between a process P and $!P$. The same authors improve the previous technique in [57]. The strategy is very similar to the previous paper, however, the precision of the analysis is improved by the use of information about the multiplicity of the number of ambients occurring within another ambient. The distinction is within the range $\{0, 1, \omega\}$ where ω means “many”.

The paper [40] defines a refinement of the analysis proposed in [86] for the special case of Safe Ambients [76]. These are a modification of mobile ambients where the execution of a capability takes place only if both the ambients involved agree. The analysis proposed — as the one in [86] — does not distinguish between different copies of the same ambient.

An abstract interpretation framework for Mobile Ambients is introduced in [75]. Based on [57] and [40] the analysis proposed in this paper introduces

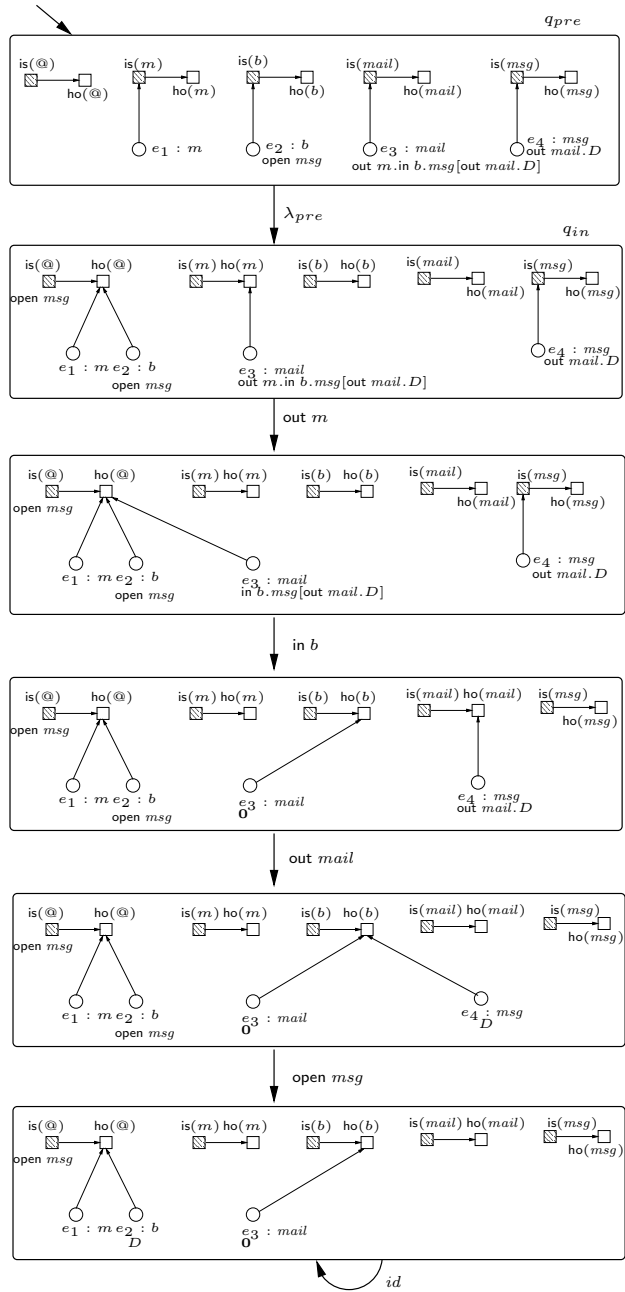


Figure 6.17: The HABA representing SYS_1 of Example 6.3.1.

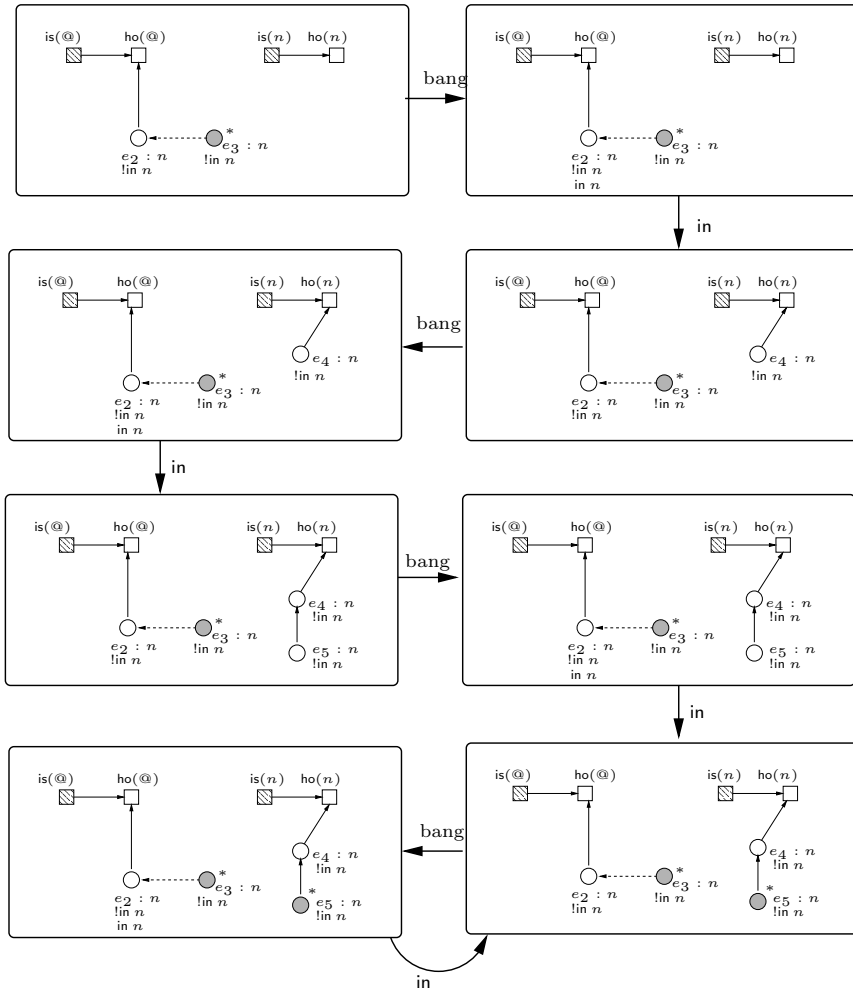


Figure 6.18: Part of the HABA representing $!n[!in n]$.

some information about multiplicity of the ambients and contextual informations.

Again, based on [57], the paper [14] defines a more accurate analysis for capturing boundary crossing. No information on multiplicities is provided.

The kind of analysis we have developed in this chapter is clearly not the main result of this thesis (as it is even closed by a conjecture). It is meant just as an illustrative application example of the techniques introduced in Chapter 5. However, it may represent an alternative approach to the main stream analysis based on abstract interpretation and control flow analysis (in the Flow Logics stile). It is somehow premature to draw conclusions on accuracy of these very different techniques. Rigorous comparisons should be provided on this respect. We leave that as future work. Nevertheless, it is worth to mention, that a strong point of our technique seems to rely on its power of counting occurrences of ambients, as well as its flexibility on tuning the precision. On the other hand, it seems that this may be more difficult in the other techniques described in the aforementioned papers. Finally, the simplicity of the model that results just as a straightforward application of $\mathcal{N}ell$ TL and HABA framework appears to us to be appealing. Even more because mobile ambients is a rather different domain than object-based systems for which the framework was designed for (this make the example in our opinion yet more interesting).

Somehow, different from the previous stream of work is the paper [22] where the authors try to identify a fragment of mobile ambients that can be verified by model-checking. For this fragment, the paper introduces a model-checking algorithm for the Ambient Logic [20].

7

Conclusions and Future work

In this last chapter we give some retrospective considerations on what has been achieved in this dissertation. Moreover, taking that as a starting point, we suggest a few possible directions that could be pursued for future research.

7.1 Achievements

In Chapter 3, we have defined BOTL, a temporal logic that is aimed at specifying properties of object-based systems. It is inspired by the Object Constraint Language (OCL) [110]. The formal semantics of BOTL was used to provide a rigorous foundation to a significant subset of OCL including, for example, invariants and pre- and postconditions. The formalisation was carried out by a translation from OCL into BOTL. The first benefit was to address a few ambiguities (as reported in the literature) that at the time of the development of BOTL were still present in OCL. More importantly, however, we believe that the mapping of OCL into BOTL sets the right formal foundation for the development of model checking tools having OCL as specification language. Being a part of the UML, OCL is rather widespread in industry. Therefore such tools could represent a significant step towards the practical introduction of formal methods among practitioners.

As we observed in Chapter 1, two crucial aspects of object-based systems are the notion of dynamic allocation (birth) and deallocation (death) as well as

dynamic references (pointers) between objects. The contribution of the second part of this thesis is related to these issues.

In Chapter 4 we have focused on dynamic allocation and deallocation by a subset of BOTL called $\mathcal{A}llTL$ in which these two notions are primitives. $\mathcal{A}llTL$ contributes to the specification of relevant properties about allocation and deallocation. On the modelling side, we have defined an extension of (generalised) Büchi automata, called HABA, whose runs generate models of $\mathcal{A}llTL$ -formulae. HABA give finite-state abstractions of infinite-state systems whose infinite nature stems from unbounded allocations. The use of HABA is exemplified by defining finite-state semantics for a simple programming language having intrinsically infinite behaviour. The most important contribution w.r.t. the notion of birth and death is the proof of decidability of the model checking problem for $\mathcal{A}llTL$. We have defined an algorithm for model checking $\mathcal{A}llTL$ -formulae against HABA. To the best of our knowledge, this represents the first *effective* approach for model-checking systems with an unbounded number of entities.

In Chapter 5, we have generalised the framework developed for $\mathcal{A}llTL$ and HABA in order to include the notion of *dynamic references*. Indeed more realistic systems can be specified and modelled with this approach. The new framework — which comprehends a logic called $\mathcal{N}allTL$ and HABA with references as models — is equipped with a more sophisticated mechanism of abstraction able to capture a wider range of infinite behaviours in a finite way. This has been illustrated by an example programming language with a simple notion of navigation. Its operational semantics defined by HABA with references is finite. Most importantly, also for this extended framework we have defined a model checking algorithm. With respect to the $\mathcal{A}llTL$ algorithm, the full decidability is lost due to the inherent complexity of the systems considered. Therefore the algorithm is sound but not complete (false negatives can be returned).

Although this dissertation focuses on aspects of object-based systems, the techniques that have been developed seem not to be limited to this domain. This is probably because at a more abstract level, allocation and deallocation of *resources* (channels, keys, processes, etc.) and manipulation of dynamic references are recurrent problems in many fields of computer science. To support this claim, we have shown (as an illustrative example) that the framework for $\mathcal{N}allTL$ and HABA may also be suitable in the context of security. In fact, in Chapter 6, we have defined an operational semantics for the calculus of Mobile Ambients tailored towards the detection of some interesting security properties of mobile processes such as *secrecy*. The properties are expressible in $\mathcal{N}allTL$ and the model checking algorithm can be exploited to automatically discover if the security properties may be violated.

7.2 Future work

Undoubtedly we close this dissertation with many open questions. Time would most probably answer the question whether this is a bad or a good sign.

The first urgent and straightforward generalisation of the framework presented here would be to drop the (unfortunately unpleasant) constraint on the single outgoing reference of entities imposed to simplify the technical machinery and obtain very precise abstractions. This means to consider full BOTL models. We strongly believe that this would allow the definition of the semantics of realistic object-based languages by means of HABA. The generalisation affects the notion of abstraction we have used so far. The main challenge is then, how to find sensible abstractions of such general models and how such an approach can scale up to systems of realistic size.

It is reasonable to believe that proper “general” abstractions for every kind of problem *do not* exist. Depending on the system to be specified, different abstractions may be better suited than others. This seems to have a traumatic impact on the definition of operational semantics for object languages. Essentially the semantics has to take into account the representation of the heap that in turn depends on the abstraction used. By changing the abstraction, a new semantics must be provided. Another interesting question is how much the operational semantics and the abstraction are actually coupled? To which extent can the two elements be designed orthogonally? With these premises, we believe that what is needed is a mechanism to easily specify abstractions: more precisely, a flexible way to define morphisms which are our main abstraction tool.

A morphism meta-language. The definition of morphism given in Chapter 5 incorporates some knowledge on the kind of structures from which to abstract. In particular, our definition explicitly contains a condition stating that multiple/unbounded entities abstract from “pure chains”. Our abstraction depends on this explicit statement. All the other conditions on morphisms are orthogonal. Accordingly, we could classify the conditions of morphisms in two categories: *abstraction-dependent* and *abstraction-independent*. Some studies we have recently carried out suggest that abstracting different kinds of heaps by morphisms would boil down to properly replacing only the abstraction-dependent conditions. On the contrary, abstraction-independent conditions seem to be necessary as well as invariant w.r.t. the way to abstract the heap. They represent minimal reasonable requirements to impose on a wide range of abstractions (e.g., the cardinality of the abstract entity corresponds to the sum of the concrete ones).

The issue is then how to simplify the specification of the abstraction-dependent conditions, so that the effort to migrate from one abstraction to another is minimised. A possibility could be the definition of some kind of “morphism language”— such as a logic — whose formulae characterise the abstraction-dependent properties.

The generalisation of the framework should be done so that most of the machinery remains unchanged, like the general structure of the operational semantics using a canonical form of the state. We have shown that this general approach can be easily transferred to Mobile Ambients. This gives us the feeling that the approach may still be applied to many different systems. Moreover, we believe that such a morphism-language should consider, besides the shape of the heap, also values (in the sense of BOTL).

Starting from a UML class model — and therefore knowing which kind of instantiation the system has — we could design ad-hoc abstractions expressing precisely what the morphism should collapse. For example, a binary tree node would always have left and right children. Therefore, it could be specified by a formula that the morphism abstracts the particular shape of a binary tree into multiple/unbounded entities. As another example, we could specify that the objects collapsed onto a multiple/unbounded entity should have a particular value in a field¹. In the example of the Hotel system in Chapter 2, we might not be interested in empty rooms, therefore we could instruct the morphism to collapse them in a single summary node. In this case the shape of this part of the heap is not relevant.

Automata theoretic approach to $\mathcal{All}TL$. Another open research question is the satisfiability of $\mathcal{All}TL$. During the design of model checking strategies for $\mathcal{All}TL$, it turned out that the definition of HABA without references is not suitable for an automata theoretic approach to model checking as described in Chapter 2. Although HABA are appropriate for automatic verification, they are not expressive enough to provide canonical models for $\mathcal{All}TL$ -formulae. $\mathcal{All}TL$ can express the order in which entities are allocated and deallocated (essentially dependencies between entities) but HABA without references cannot. HABA with references, on the other hand, may be able to circumvent the problem. It should then be investigated if indeed they are expressive enough to act as canonical models of $\mathcal{All}TL$ -formulae.

HABA as specification language. Related to the issue described above is the possibility to use HABA as specification language (like Büchi automata are used for specification purposes). This would also provide means for new algorithms on the automata theoretic style which, in turn, would require the definition of notions such as the negation of HABA as well as the intersection (cf. Section 2.1.5). For checking the emptiness of the language of HABA without references, the procedure 2 would suffice. Concerning HABA with references, though, we foresee some problems (see below).

HABA with references emptiness problem. It seems quite likely that deciding the emptiness of the language for a HABA with references is undecidable. Further research must be done in this direction in order to substantiate

¹This is somehow similar to abstract interpretation.

this conjecture. Apart from the pure theoretical curiosity, the ability to decide the emptiness of the language, can be very useful. First of all because, as we have seen, it represents one of the bases of the automata theoretic model checking algorithm. The second and most important reason has been suggested by some of our recent studies. It seems that, by imposing a stronger notion of fulfilling path, the decidability of the $\mathcal{N}\ell\ell\text{TL}$ model checking problem may reduce to deciding whether the underlying run of the path has a non-empty language².

Tools implementation. A rather important aspect that has been unfortunately neglected in this thesis is the implementation of the developed theories. Experimental results would give us some insights into the usefulness of the overall approach. Investigations in this direction should involve also optimisations of the algorithms presented.

Other applications. Along the lines of Chapter 6, other possible applications of the frameworks should be investigated. Chapter 6 encourages us that the encodings of other problems in terms of HABA should be possible.

²For a more detailed discussion on this point we refer the reader to the end of Section 5.7.4.

Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [2] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
- [3] M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *LNCS*, pages 682–696. Springer, 1997.
- [4] D.S. Andersen, L.H. Pedersen, H. Huttel, and J. Kleist. Objects, types and modal logics. In *Foundations of Object-Oriented Languages (FOOL)*, 1997. Electronic Proceedings.
- [5] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*, pages 1–3, 2002.
- [6] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, December 1983.
- [7] M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct realization of interface constraints with OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Proceedings 2nd International Conference*, volume 1723 of *LNCS*, pages 399–415. Springer, 1999.
- [8] G. Booch. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1990.
- [9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
- [10] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings 3rd International Conference*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.

- [11] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *Theoretical Computer Science (TCS)*, 114(1):31–61, 1993.
- [12] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In *Proc. ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics Based Program Manipulation (PEPM)*. ACM, 2003.
- [13] J.C. Bradfield, J.K. Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.
- [14] C. Braghin, A. Cortesi, and R. Focardi. Control flow analysis of mobile ambients with security boundaries. In B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 197–212. Kluwer, 2002.
- [15] E. Brinksma. Verification is experimentation! In C. Palamidessi, editor, *Concurrency Theory, 11th International Conference (CONCUR)*, volume 1877 of *LNCS*, pages 17–24. Springer, 2000.
- [16] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.
- [17] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). *TACS'01*, 2255:1–37, 2001.
- [18] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer, F.T. Ruiz, R. Morales, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium (ICALP)*, volume 2380 of *LNCS*, pages 597–610. Springer, 2002.
- [19] L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, 1998.
- [20] L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 365–377. ACM Press, 2000.
- [21] L. Cardelli and A.D. Gordon. Logical properties of name restriction. In *International Conference on Typed Lambda Calculi and Applications (TCLA 2001)*, volume 2044 of *LNCS*, pages 46–60. Springer, 2001.
- [22] W. Charatonik, A.D. Gordon, and J.-M. Talbot. Finite-control mobile ambients. In D. Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming (ESOP)*, volume 2305 of *LNCS*, pages 295–313. Springer, 2002.

- [23] Y. Choueka. Theories of automata on omega-tapes: A simplified approach. *Journal of Computer and System Sciences (JCSS)*, 8(2):117–141, 1974.
- [24] T. Clark. Type checking UML static diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Proceedings 2nd International Conference*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.
- [25] T. Clark and J. Warmer, editors. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.
- [26] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
- [27] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In D. Agnew, L.J.M. Claesen, and R. Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 15–30. North-Holland, 1993.
- [28] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification, 12th International Conference (CAV)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
- [29] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [30] S. Conrad and K. Turowski. Temporal OCL: Meeting specification demands for business components. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 151–166. Idea Publishing Group, 2001.
- [31] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. In Clark and Warmer [25], pages 115–149.
- [32] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448. IEEE Computer Society, 2000.
- [33] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *International Journal on Software Tools for Technology Transfer*, 4(1):34–56, 2002.
- [34] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Volume B: Formal Models and Semantics, pages 193–242. Elsevier and MIT Press, 1990.

- [35] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
- [36] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
- [37] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
- [38] J.W. de Bakker and E. de Vink. *Control Flow Semantics*. MIT Press, Cambridge, MA, 1996.
- [39] F.S. de Boer. A proof system for the parallel object-oriented language POOL. In Mike Paterson, editor, *17th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 443 of *LNCS*, pages 572–585. Springer, 1990.
- [40] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Asian Computing Science Conference*, volume 1961 of *LNCS*, pages 199–214. Springer-Verlag, 2000.
- [41] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, 1999.
- [42] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 305–326. Kluwer Academic Publishers, 2000.
- [43] D. Distefano, J.-P. Katoen, and A. Rensink. Towards model checking OCL. In *Defining Precise Semantics for UML (satellite workshop of ECOOP 2000)*, 2000. Position Paper.
- [44] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking dynamic allocation and deallocation. CTIT Technical Report TR-CTIT-01-40, Faculty of Informatics, University of Twente, December 2001.
- [45] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R.A. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Networking and Mobile Computing. In 2nd IFIP International Conference on Theoretical Computer Science (TCS)*, pages 435–447. Kluwer, 2002.
- [46] D. Distefano, A. Rensink, and J.-P. Katoen. Who is pointing when to whom: on model-checking pointer structures. CTIT Technical Report TR-CTIT-03-12, Faculty of Informatics, University of Twente, March 2003.

- [47] E.A. Emerson. Automata, tableaux and temporal logics. In Rohit Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *LNCS*, pages 79–88, Brooklyn, NY, June 1985. Springer.
- [48] S. Flake and W. Mueller. An OCL extension for real-time constraints. In Clark and Warmer [25], pages 150–171.
- [49] The Real-Time for Java Expert Group. *The Real-Time Specification for Java 1.0*. <http://www.rtfj.org>, 2000.
- [50] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, LICS, pages 214–224. IEEE Computer Society Press, 1999.
- [51] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Proceedings of the 13th International Symposium on Protocol Specification, Testing and Verification*, pages 3–18. Chapman & Hall, 1995.
- [52] M. Gogolla and M. Richters. On combining semi-formal and formal object specification techniques. In F. Parisi-Presice, editor, *Recent Trends in Algebraic Development Techniques: 12th International Workshop, (WADT'97: selected papers)*, volume 1376 of *LNCS*. Springer, 1998.
- [53] M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, 1998.
- [54] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
- [55] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. 1st International Workshop*, volume 1618 of *LNCS*, pages 162–172. Springer, 1998.
- [56] A. Hamie, J. Howse, and S. Kent. Interpreting the Object Constraint Language. In *Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98)*, pages 288–295. IEEE Computer Society, 1998.
- [57] R.R. Hansen, J.G. Jensen, F. Nielson, and H.R. Nielson. Abstract interpretation of mobile ambients. In A. Cortesi and G. Filé, editors, *Proceedings of Static Analysis, 6th International Symposium, SAS '99*, volume 1694 of *LNCS*, pages 135–148. Springer, 1999.
- [58] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

- [59] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In T. Ball and S.K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
- [60] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica* 2, pages 335–355, 1973.
- [61] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, 1997.
- [62] G.J. Holzmann. Software analysis and model checking. In E. Brinksma and K.G. Larsen, editors, *Computer Aided Verification, 14th International Conference (CAV)*, volume 2404 of *LNCS*, pages 1–16. Springer, 2002.
- [63] G.J. Holzmann and M.H. Smith. Software model checking. In J. Wu and S.T. Chanson nad Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems (FORTE)*, pages 481–497. Kluwer, 1999.
- [64] K. Huizing, R. Kuiper, and SOOP. Verification of object oriented programs using class invariants. In T.S.E. Maibaum, editor, *Fundamental Approaches to Software Engineering, (FASE)*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.
- [65] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Proceedings 3rd International Conference*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
- [66] R. Iosif and R. Sisto. On the specification and semantics of source level properties in Java. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, pages 83–88, 2000. (Held in conjunction with 22nd International Conference on Software Engineering).
- [67] S. Ishtiaq and P. O’Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–26, 2001.
- [68] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
- [69] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering*, pages 730–733. IEEE CS Press, 2000.
- [70] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL — A language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):175–211, 1996.
- [71] J.-P. Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999.

- [72] S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
- [73] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.
- [74] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In J. Palsberg, editor, *Static Analysis, 7th International Symposium (SAS)*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.
- [75] F. Levi and S. Maffei. An abstract interpretation framework for analysing mobile ambients. In P. Cousot, editor, *Static Analysis, 8th International Symposium, (SAS)*, volume 2126 of *LNCS*, pages 395–411. Springer, 2001.
- [76] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL)*, pages 352–364. ACM Press, 2000.
- [77] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. ACM, 1985.
- [78] Z. Manna and A. Pnueli. *Temporal Verification of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
- [79] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
- [80] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.
- [81] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.
- [82] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, 1992.
- [83] U. Montanari and M. Pistore. An introduction to history-dependent automata. In A. Gordon, A. Pitts, and C. Talcott, editors, *Conference Record of the Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *ENTCS*. Elsevier Science Publishers, 1997.
- [84] U. Montanari and M. Pistore. History-dependent automata. Technical Report TR-98-11, Dipartimento di Informatica University of Pisa, October 1998.
- [85] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

- [86] F. Nielson, H.R. Nielson, R.R. Hansen, and J.G. Jensen. Validating farewalls in mobile ambients. In J.C.M. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99: Concurrency Theory, 10th International Conference*, volume 1664 of *LNCS*, pages 463–477. Springer, 1999.
- [87] A.M. Odlyzko. Asymptotic enumeration methods. In R.L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, chapter 22, pages 1063–1229. North-Holland, Amsterdam, 1995.
- [88] OMG. Object constraint language specification. In *OMG Unified Modelling Language Specification, version 1.3*. Object Modeling Group, June 1999. [89] chapter 7.
- [89] OMG. *Unified Modelling Language Specification, version 1.3*. Object Modeling Group, June 1999. <http://www.omg.org>.
- [90] M. Pistore. *History-Dependent Automata*. PhD thesis, University of Pisa, March 1999.
- [91] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57. IEEE Computer Society Press, 1977.
- [92] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W.-P. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–424. Kluwer, 1998.
- [93] S. Ramakrishnan and J. McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems*, 1999.
- [94] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.
- [95] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, University of Bremen, 2002.
- [96] M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language OCL. In T. Wang Ling, S. Ram, and M. Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.
- [97] M. Richters and M. Gogolla. OCL: Syntax, semantics, and tools. In Clark and Warmer [25], pages 42–68.
- [98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

- [99] T.C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, 2001.
- [100] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.
- [101] M. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.
- [102] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.
- [103] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
- [104] M. Smyth. Topology. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 641–761. Clarendon Press, 1992.
- [105] J. Staunstrup, H.R. Andersen, H. Hulgaard, J. Lind-Nielsen, K.G. Larsen, G. Behrmann, K.J. Kristoffersen, A. Skou, and N.B. Theilgaard H. Leerberg. Practical verification of embedded software. *IEEE Computer*, 33(5):68–75, 2000.
- [106] P. Stevens and Pooley R. *Using UML: software engineering with objects and components*. Object Technology Series. Addison-Wesley Longman, 1999.
- [107] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
- [108] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B.V., 1990.
- [109] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, 1986.
- [110] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.
- [111] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13, 1999.
- [112] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In C. Norris and J.J.B. Fenwick, editors, *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL)*, pages 27–40. ACM Press, 2001.

- [113] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming (ESOP)*, volume 2618 of *LNCS*, pages 204–222. Springer, 2003.
- [114] P. Ziemann and M. Gogolla. An extension of OCL with temporal logic. In J. Jürjens, M.V. Cengarle, E.B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 Workshop*, pages 53–62. Technische Universität München, Institut für Informatik, 2002.

A

Proofs of Chapter 4

A.1 Proofs of Section 4.2

Proposition 4.2.12 For $\mathcal{A}llTL$ -formula ϕ , folded allocation sequences σ_f and allocation sequence σ_u :

1. For every $(\sigma_u, N_u, \theta_u)$ there exists a $(\sigma_f, N_f, \theta_f)$ such that

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f)$$

2. For every $(\sigma_f, N_f, \theta_f)$ there exists a $(\sigma_u, N_u, \theta_u)$ such that

$$(\sigma_u, N_u, \theta_u) \sqsubseteq^{fold} (\sigma_f, N_f, \theta_f).$$

Proof.

1. Trivial by taking $(\sigma_f, N_f, \theta_f) = (id(\sigma_u), N_u, \theta_u)$.
2. Let $\sigma_f = E_0 \lambda_0 E_1 \lambda_1 \dots$. We prove the proposition by defining $(\sigma_u, N_u, \theta_u)$ such that $id(\sigma_u) \cong \sigma_f$. Let $\sigma_u = E'_0 E'_1 \dots$. By definition of \cong , $id(\sigma_u) \cong \sigma_f$ if (for all $i \geq 0$) $E_i \cong E'_i$ and

$$\lambda_i \circ h_i = h_{i+1} \circ id_i \tag{A.1}$$

where h_i is an isomorphism between E_i and E'_i .

We show that it is possible to define such E'_i by induction on i .

[Base] It suffices to choose E'_0 such that $|E_0| = |E'_0|$. As these sets are isomorphic, let h_0 be such isomorphism between E_0 and E'_0 .

[Step] Assume we have constructed the sequences up to index i (with $i > 0$). The proof obligation is to construct $E'_{i+1} \cong E_{i+1}$ satisfying (A.1). Let $E'_{i+1} = E_{old} \cup E_{new}$ with $E_{old} \cap E_{new} = \emptyset$. Choose $E_{old} = h_i^{-1}(\text{dom}(\lambda_i))$ i.e., $E_{old} \subseteq E'_i$ is the set of entities that do not die during the transition from E_i to E_{i+1} , and that in E'_{i+1} are old. E_{new} corresponds to the new entities of E_{i+1} . Choose E_{new} such that $|E_{new}| = |E_{i+1} \setminus \text{cod}(\lambda_i)|$ and $E_{new} \cap E'_i = \emptyset$. The first constraint on E_{new} avoids to choose entities that in E'_i correspond to entities in E_i that died during the transition from E_i to E_{i+1} . The second constraint establishes $E_{old} \cap E_{new} = \emptyset$. As $|E_{new}| = |E_{i+1} \setminus \text{cod}(\lambda_i)|$, E_{new} is isomorphic to the set of new entities in E_{i+1} ; let $h : E_{new} \rightarrow (E_{i+1} \setminus \text{cod}(\lambda_i))$ be such isomorphism. Then we define $h_{i+1} : E'_{i+1} \rightarrow E_{i+1}$ by:

$$h_{i+1}(e) = \begin{cases} \lambda_i \circ h_i(e) & \text{if } e \in E_{old} \\ h(e) & \text{if } e \in E_{new}. \end{cases}$$

It is easy to see that this definition satisfies (A.1). □

A.2 Proofs of Section 4.3

Lemma 4.3.9 For HABA \mathcal{H} and any expansion $Exp(\mathcal{H})$:

- (a) $\mathcal{L}(Exp(\mathcal{H})) \sqsupseteq^{fold} \mathcal{L}(\mathcal{H})$ and
- (b) $\mathcal{L}(Exp(\mathcal{H})) \sqsubseteq^{fold} \mathcal{L}(\mathcal{H})$.

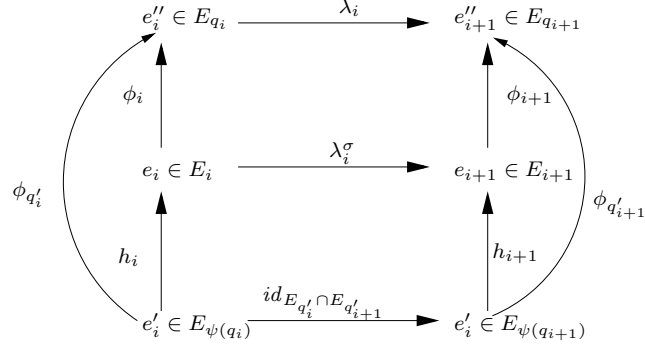
Proof. Let \mathcal{H} be a HABA and $Exp(\mathcal{H})$ an ABA that expands \mathcal{H} with $\psi : Q_{Exp(\mathcal{H})} \rightarrow Q_{\mathcal{H}}$ surjective.

(a) Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$. To prove: there exists $(\sigma', N', \theta') \in \mathcal{L}(Exp(\mathcal{H}))$ such that $(\sigma', N', \theta') \sqsubseteq^{fold} (\sigma, N, \theta)$. Let $\sigma = E_0 \lambda_0^\sigma E_1 \lambda_1^\sigma \dots$ be such that $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots \in \text{runs}(\mathcal{H})$ generates (σ, N, θ) with a generator $(\phi_i)_{i \in \mathbb{N}}$. Let $\rho' = q'_0 q'_1 \dots \in \text{runs}(Exp(\mathcal{H}))$. As $Exp(\mathcal{H})$ expands \mathcal{H} , it follows that for $i \geq 0$: $q_i = \psi(q'_i)$, $q'_i \rightarrow q'_{i+1}$ expands $q_i \rightarrow_{\lambda_i} q_{i+1}$ and $|E_{q'_i}| = |E_i|$. These facts relate ρ , ρ' and σ . Let $\sigma' = E_{q'_0} E_{q'_1} E_{q'_2} \dots$. It follows directly that ρ' accepts σ' . Note that, by construction, $\phi_{q'_i} : E_{q'_i} \rightarrow E_{q_i}$ and $\phi_i : E_i \rightarrow E_{q_i}^\infty$ are bijective on the entities that are not mapped onto ∞ , i.e., $|E_i \setminus E_i^\infty| = |E_{q_i}| = |E_{q'_i} \setminus E_{q'_i}^\infty|$ for all $i \geq 0$. In order to show that $id(\sigma') \cong \sigma$, we prove that there exists a family of bijections $(h_i)_{i \geq 0}$ with $h_i : E_{q'_i} \rightarrow E_i$ such that

$$\lambda_i^\sigma \circ h_i = h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}. \quad (\text{A.2})$$

Then it follows $(id(\sigma'), h_0^{-1}(N), h_0^{-1} \circ \theta) \cong (\sigma, N, \theta)$, and by definition,

$$(\sigma', h_0^{-1}(N), h_0^{-1} \circ \theta) \sqsubseteq^{fold} (\sigma, N, \theta)$$

Figure A.1: q_i is generated by $q'_i = \psi(q_i)$.

which proves (a). Consider the following definitions of h_i . For $i = 0$, let:

$$h_0(e) = \begin{cases} \tilde{h}(e) & \text{if } \phi_{q'_0}(e) = \infty \\ \phi_0^{-1} \circ \phi_{q'_0}(e) & \text{if } \phi_{q'_0}(e) \neq \infty \end{cases}$$

where \tilde{h} is an arbitrary isomorphism between $E_{q'_0}^\infty$ and E_0^∞ . Note that such isomorphism exists, since $|E_{q'_i}^\infty| = |E_i^\infty|$ for all i . For $i > 0$, let:

$$h_{i+1}(e) = \begin{cases} \lambda_i^\sigma \circ h_i \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}^{-1}(e) & \text{if } \phi_{q'_{i+1}}(e) = \infty \\ \phi_{i+1}^{-1} \circ \phi_{q'_{i+1}}(e) & \text{if } \phi_{q'_{i+1}}(e) \neq \infty \end{cases}$$

The fact that h_{i+1} is well defined can be seen as follows. If $\phi_{q'_{i+1}}(e) = \infty$ then $e \in E_{q'_i} \cap E_{q'_{i+1}}$, otherwise the number of new entities in $E_{q'_{i+1}}$ and $E_{q_{i+1}}$ would differ, which cannot be the case due to condition (ii) of Def. 4.3.8. Thus, $id_{E_{q'_i} \cap E_{q'_{i+1}}}^{-1}(e)$ is defined. If $\phi_{q'_{i+1}}(e) \neq \infty$ then $h_{i+1}(e)$ is defined, since ϕ_{i+1} is bijective on entities not mapped onto ∞ .

We now show that h_i satisfies (A.2) in the following steps (cf. Fig. A.1): For all $i \geq 0$:

$$\phi_{q'_i} = \phi_i \circ h_i \tag{A.3}$$

The proof is by induction on i .

(Base) For $i = 0$ it follows directly from the definition of h_0 .

(Step) Assume (A.3) holds for $i > 0$. We prove case $i+1$. According to the definition of h_{i+1} , we distinguish:

- $\phi_{q'_{i+1}}(e) \neq \infty$. Then $\phi_{i+1} \circ h_{i+1}(e) = \phi_{i+1} \circ \phi_{i+1}^{-1} \circ \phi_{q'_{i+1}}(e) = \phi_{q'_{i+1}}(e)$.

- $\phi_{q'_{i+1}}(e) = \infty$. Then we have

$$\begin{aligned}
& \phi_{i+1} \circ h_{i+1}(e) \\
&= \phi_{i+1} \circ \lambda_i^\sigma \circ h_i \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}^{-1}(e) \\
&= [\phi_{q'_{i+1}}(e) = \infty \text{ implies } e \in E_{q'_i} \cap E_{q'_{i+1}}] \\
& \quad \phi_{i+1} \circ \lambda_i^\sigma \circ h_i(e) \\
&= [\text{by condition 4 of Def. 4.3.6}] \\
& \quad \lambda_i \circ \phi_i \circ h_i(e) \\
&= [\text{by induction hypothesis}] \\
& \quad \lambda_i \circ \phi_{q'_i} \\
&= [\text{by condition (i) of Def. 4.3.8}] \\
& \quad \phi_{q'_{i+1}} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e) \\
&= \phi_{q'_{i+1}}(e)
\end{aligned}$$

This completes the proof of (A.3). Using this fact, we prove for all $i \geq 0$:

$$\text{dom}(\lambda_i^\sigma \circ h_i) = \text{dom}(h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}) \quad (\text{A.4})$$

Note that $\text{dom}(h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}) = E_{q'_i} \cap E_{q'_{i+1}}$ and $\text{dom}(\lambda_i^\sigma \circ h_i) = \{e \in E_{q'_i} \mid h_i(e) \in \text{dom}(\lambda_i^\sigma)\}$.

- ‘ \supseteq ’: let $e \in E_{q'_i} \cap E_{q'_{i+1}}$. By (A.3) it follows $\phi_{q'_i}(e) = \phi_i \circ h_i(e)$. By condition (i) of Def. 4.3.8, we have that $\lambda_i \circ \phi_i \circ h_i(e)$ is defined, since $e \in E_{q'_i} \cap E_{q'_{i+1}}$. This implies by condition 4 of Def. 4.3.6 that $\phi_{i+1} \circ \lambda_i^\sigma \circ h_i(e)$ is defined and therefore $e \in \text{dom}(\lambda_i^\sigma \circ h_i)$.
- ‘ \subseteq ’: let $e \in \text{dom}(\lambda_i^\sigma \circ h_i)$. Since $\lambda_i^\sigma \circ h_i(e)$ is defined, $\phi_{i+1} \circ \lambda_i^\sigma \circ h_i(e)$ is defined as well. By condition 4 of Def. 4.3.6, $\lambda_i \circ \phi_i \circ h_i(e)$ is defined. From (A.3) it follows that $\lambda_i \circ \phi_{q'_i}(e)$ is defined. But then by condition (i) of Def. 4.3.8, it must be $e \in E_{q'_i} \cap E_{q'_{i+1}}$.

This completes the proof of (A.4). Finally, we prove for all $i \geq 0$:

$$\forall e \in \text{dom}(\lambda_i^\sigma \circ h_i) : \left(\lambda_i^\sigma \circ h_i(e) = h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e) \right). \quad (\text{A.5})$$

The equality can be also understood by the fact that the diagram in Fig. A.1 commutes. Let $e \in E_{q'_i} \cap E_{q'_{i+1}}$. We distinguish two cases:

- $\phi_{q'_{i+1}}(e) \neq \infty$. Then

$$\begin{aligned}
& \lambda_i^\sigma \circ h_i(e) \\
&= [\text{by definition of } h_i] \\
& \lambda_i^\sigma \circ \phi_i^{-1} \circ \phi_{q'_i}(e) \\
&= [\text{since } \phi_{i+1}^{-1}(e) \text{ is defined and by Def. 4.3.6 } \phi_{i+1}^{-1} \circ \lambda_i \circ \phi_i = \lambda_i^\sigma] \\
& \phi_{i+1}^{-1} \circ \lambda_i \circ \phi_i \circ \phi_i^{-1} \circ \phi_{q'_i}(e) \\
&= \phi_{i+1}^{-1} \circ \lambda_i \circ \phi_{q'_i}(e) \\
&= [q'_i \rightarrow q'_{i+1} \text{ expands } q_i \rightarrow_{\lambda_i} q_{i+1}, \text{ by condition (i) of Def. 4.3.8}] \\
& \phi_{i+1}^{-1} \circ \phi_{q'_{i+1}} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e) \\
&= [\phi_{q'_{i+1}}(e) \neq \infty, \text{ definition of } h_{i+1}] \\
& h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e).
\end{aligned}$$

- $\phi_{q'_{i+1}}(e) = \infty$. Then we have

$$\begin{aligned}
& \lambda_i^\sigma \circ h_i(e) \\
&= \lambda_i^\sigma \circ h_i \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}^{-1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e) \\
&= [\text{by definition of } h_{i+1}] \\
& h_{i+1} \circ id_{E_{q'_i} \cap E_{q'_{i+1}}}(e).
\end{aligned}$$

From (A.4) and (A.5) it follows (A.2). This completes the proof of (a).

(b) Let $(\sigma, N, \theta) \in \mathcal{L}(Exp(\mathcal{H}))$ with $\sigma = E_0 E_1 E_2 \dots$ and $\rho = q_0 q_1 q_2 \dots$ be the run that generates σ . To prove: there exists $\sigma' \in \mathcal{L}(\mathcal{H})$ such that $\sigma' \cong id(\sigma)$. Since $\rho \in \text{runs}(Exp(\mathcal{H}))$, there exists $\rho' = \psi(q_0) \lambda_0 \psi(q_1) \lambda_1 \dots \in \text{runs}(\mathcal{H})$ such that $q_i \rightarrow q_{i+1}$ expands $\psi(q_i) \rightarrow_{\lambda_i} \psi(q_{i+1})$ for all $i \geq 0$. Since ρ is a run of $Exp(\mathcal{H})$, every state in ρ satisfies conditions 1–6 of Def. 4.3.8 for a family of functions $\phi_{q_i} : E_{q_i} \rightarrow E_{\psi(q_i)}^\infty$. Choose $\sigma' = id(\sigma)$. We show that ρ' generates $(id(\sigma), N, \theta)$. This amounts to prove the existence of a generator $\phi'_i : E_i \rightarrow E_{\psi(q_i)}^\infty$ satisfying the conditions of Def. 4.3.6. Let $\phi'_i = \phi_{q_i}$. Conditions 1–3 follow directly from the definition of ϕ_{q_i} (since by definition it satisfies conditions 1–3 of Def. 4.3.8). Furthermore, by definition of expansion: $\lambda_i \circ \phi_{q_i} = \phi_{q_{i+1}} \circ id_{E_{q_1} \cap E_{q_2}}$. Thus, condition 4 is fulfilled. Consider now condition 5. Assume $e \in E_{i+1}$, $\phi_{i+1}(e) = \infty$ and $e \notin E_i$. Then $e \in E_{q_{i+1}} \setminus E_{q_i}$ and therefore $\phi_{i+1}(e) \in E_{\psi(q_{i+1})} \setminus \text{cod}(\lambda_i)$. This is, however, impossible since by hypothesis $\phi_{i+1}(e) = \infty$. Finally, condition 6 holds by definition of ϕ_i . Hence, we conclude that $(id(\sigma), N, \theta) \in \mathcal{L}(\mathcal{H})$. \square

Theorem 4.3.10 For any HABA \mathcal{H} and expansion $Exp(\mathcal{H})$:

$$\mathcal{L}(\mathcal{H}) \cong \mathcal{L}(id(Exp(\mathcal{H}))).$$

Proof. ‘ \sqsubseteq^{fold} ’: Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$. To prove: there exists $(\sigma', N', \theta') \in \mathcal{L}(id(Exp(\mathcal{H})))$ such that $(\sigma, N, \theta) \cong (\sigma', N', \theta')$. By Lemma 4.3.9.a it follows that there exists $(\sigma'', N'', \theta'') \in \mathcal{L}(Exp(\mathcal{H}))$ such that $(\sigma, N, \theta) \sqsubseteq^{fold}$

$(\sigma'', N'', \theta'')$. By definition of \sqsubseteq^{fold} , thus

$$(\sigma, N, \theta) \cong (id(\sigma''), N'', \theta'') \in \mathcal{L}(id(Exp(\mathcal{H}))).$$

‘ \sqsubseteq^{fold} ’: Let $(\sigma, N, \theta) \in \mathcal{L}(id(Exp(\mathcal{H})))$. To prove: there exists $(\sigma', N', \theta') \in \mathcal{L}(\mathcal{H})$ such that $(\sigma, N, \theta) \cong (\sigma', N', \theta')$. As $(\sigma, N, \theta) \in \mathcal{L}(id(Exp(\mathcal{H})))$ we have $(\sigma, N, \theta) = (id(\sigma''), N, \theta)$ for some σ'' with $(\sigma'', N, \theta) \in \mathcal{L}(Exp(\mathcal{H}))$. By Lemma 4.3.9.b it follows that $(\sigma'', N, \theta) \sqsubseteq^{fold} (\sigma', N', \theta')$ for some $(\sigma', N', \theta') \in \mathcal{L}(\mathcal{H})$. By definition of \sqsubseteq^{fold} , thus $(\sigma, N, \theta) \cong (\sigma', N', \theta')$. \square

A.3 Proofs of Section 4.4

Theorem 4.4.9 For any $p \in \mathcal{L}$: \mathcal{A}_p is an expansion of \mathcal{H}_p .

Proof. Let $\psi : Q_{\mathcal{A}_p} \rightarrow Q_{\mathcal{H}_p}$ be defined in the following way:

$$\psi(s, E, \gamma) = (s, E') \text{ where } E' = \{\gamma^{-1}(e) \mid e \in E\}.$$

If entity $e \in E$ is not referenced by any variable, then $\gamma^{-1}(e) = \emptyset$. Thus, $\emptyset \in E'$ and state $\psi(s, E, \gamma)$ is unbounded. For state $q \in Q_{\mathcal{A}_p}$, let $\phi_q : E_q \rightarrow E_{\psi(q)}^\emptyset$ be defined as follows:

$$\phi_{(s, E, \gamma)}(e) = \gamma^{-1}(e).$$

To show that \mathcal{A}_p expands \mathcal{H}_p we prove that ϕ_q fulfils conditions 1–6 of Def. 4.3.8:

1. If $\phi_{(s, E, \gamma)}(e) = \phi_{(s, E, \gamma)}(e') \neq \emptyset$, then $\gamma^{-1}(e) = \gamma^{-1}(e')$. Since γ is well defined, it follows $e = e'$.
 2. Straightforward, since if $X_i \in E_{\psi(q)}$, then there exists e such that $X_i = \gamma^{-1}(e)$.
 3. If $\emptyset \notin E_{\psi(q)}$ then, by definition of ψ , we have for all $e \in E : e \in \text{cod}(\gamma)$. Therefore, by definition of ϕ_q , we have for all $e \in E : \phi_q(e) \neq \emptyset$.
- 5.+6. Follows directly, as the set of initial and final states for \mathcal{H}_p and \mathcal{A}_p are identical.

It remains to prove the 4th condition:

- a) any $\psi(s_1, E_1, \gamma_1) \rightarrow_\lambda q'_2$ is expanded by some transition $(s_1, E_1, \gamma_1) \rightarrow (s_2, E_2, \gamma_2)$, and
- b) any $(s_1, E_1, \gamma_1) \rightarrow (s_2, E_2, \gamma_2)$ is expanded by some $\psi(s_1, E_1, \gamma_1) \rightarrow_\lambda \psi(s_2, E_2, \gamma_2)$ for some λ .

These statements are proved by induction on the structure of statement s_1 . The base cases:

- **case new(v).** Let $q_1 = (\text{new}(v), E_1, \gamma_1) \in Q_{\mathcal{A}_p}$. We have that $\psi(q_1) = (\text{new}(v), E_{\psi(q_1)})$. According to the symbolic semantics (cf. Table 4.4.3), the only possible transition is

$$\text{new}(v), E_{\psi(q_1)} \rightarrow_{\lambda} \text{skip}, E' \quad (\text{A.6})$$

where $E' = \{X_i \setminus \{v\} \mid X_i \in E_{\psi(q_1)}\} \cup \{\{v\}\}$. The only transition of q_1 (cf. Table 4.4.2) is

$$\text{new}(v), E_1, \gamma_1 \rightarrow \text{skip}, E_2, \gamma_1\{e/v\} \quad (\text{A.7})$$

where $E_2 = E_1 \cup \{e\}$ and $e \notin E_1$. For convenience, let $q_2 = \text{skip}, E_2, \gamma_1\{e/v\}$. We show that (A.7) expands (A.6). First, we observe that $\psi(q_2) = (\text{skip}, E_{\psi(q_2)}) = (\text{skip}, E')$, since:

$$\begin{aligned} & E_{\psi(q_2)} \\ &= \{\gamma_1\{e/v\}^{-1}(e') \mid e' \in E_1 \cup \{e\}\} \\ &= \{\gamma_1\{e/v\}^{-1}(e') \mid e' \in E_1\} \cup \{\gamma_1\{e/v\}^{-1}(e)\} \\ &= [\text{since } e \notin E_1] \\ &\quad \{\gamma_1^{-1}(e') \setminus \{v\} \mid e' \in E_1\} \cup \{\{v\}\} \\ &= \{X_i \setminus \{v\} \mid X_i \in E_{\psi(q_1)}\} \cup \{\{v\}\} \\ &= E'. \end{aligned}$$

Thus, $\psi(\text{skip}, E_1 \cup \{e\}, \gamma_1\{e/v\}) = (\text{skip}, E_{\psi(q_2)}) = (\text{skip}, E')$. Next, we need to check conditions (i) and (ii) of Def. 4.3.8. If $e' \in E_1$, we have:

$$\begin{aligned} & \phi_{q_2}(id_{E_{q_1} \cap E_{q_2}}(e')) \\ &= [\text{since } e' \in E_{q_1} \Rightarrow e' \in E_{q_2}] \\ & \quad \phi_{q_2}(e') \\ &= \gamma_1^{-1}\{e/v\}(e') \\ &= [\text{since } e' \in E_{q_1} \Rightarrow e' \neq e] \\ & \quad \gamma_1^{-1}(e') \setminus \{v\} \\ &= [\text{by new's rule in Table 4.4.3}] \\ & \quad \lambda(\gamma_1^{-1}(e')) \\ &= \lambda(\phi_{q_1}(e')). \end{aligned}$$

This proves (i). (ii) follows directly from the fact that for **new** in both Table 4.4.3 and Table 4.4.2 only one new entity is created. Thus, transition (A.7) expands (A.6).

It remains to check condition 4.b. As we have seen above, the only possible transition $q_1 \rightarrow q_2$ in the concrete model is (A.7). Furthermore, we have $\psi(\text{new}(v), E_1, \gamma_1) = (\text{new}(v), E_{\psi(q_1)})$ and $\psi(\text{skip}, E_1 \cup \{e\}, \gamma_1\{e/v\}) = (\text{skip}, E_{\psi(q_2)}) = (\text{skip}, E')$ where $E' = \{X_i \setminus \{v\} \mid X_i \in E_{\psi(q_1)}\} \cup \{\{v\}\}$. Again, for $s_1 = \text{new}(v)$ the symbolic model prescribes only one transition, namely (A.6). As we have proved for case a), (A.7) expands (A.6).

- **case $v := w$.** Let $q_1 = (v := w, E_1, \gamma_1) \in Q_{\mathcal{A}_p}$, and $\psi(q_1) = (v := w, E_{\psi(q_1)})$. According to the rules of the symbolic model, $\psi(q_1)$ has only

one possible transition (cf. Table 4.4.3), i.e., we have

$$v := w, E_{\psi(q_1)} \rightarrow_{\lambda} \text{skip}, E' \quad (\text{A.8})$$

where $E' = \{X_i \setminus \{v\} | w \notin X_i\} \cup \{X_i \cup \{v\} | w \in X_i\}$. For q_1 there is only one possibility (cf. Table 4.4.2):

$$v := w, E_1, \gamma_1 \rightarrow \text{skip}, E_1, \gamma_1 \{\gamma_1(w)/v\}. \quad (\text{A.9})$$

We show that (A.9) expands (A.8):

$$\begin{aligned} & E_{\psi(q_2)} \\ &= \{\gamma_1 \{\gamma_1(w)/v\}^{-1}(e') | e' \in E_1\} \\ &= \{\gamma_1^{-1}(e') \setminus \{v\} | e' \in E_1 \setminus \{\gamma_1(w)\}\} \cup \\ &\quad \{\gamma_1^{-1}(\gamma_1(w)) \cup \{v\}\} \\ &= \{X_i \setminus \{v\} | w \notin X_i\} \cup \{X_i \cup \{v\} | w \in X_i\} \\ &= E'. \end{aligned}$$

Note that if there is at least another variable $v' \neq v$ referring to v 's entity, then $\psi(q_2)$ remains bounded if $\psi(q_1)$ was bounded. If $\psi(q_1)$ was unbounded, then $\psi(q_2)$ is unbounded. Now, we show that conditions (i) and (ii) of Definition 4.3.8 are fulfilled. Note that $e' \in E_{q_1} \Rightarrow e' \in E_{q_2}$. If $e' \neq \gamma_1(w)$ then $\phi_{q_2}(id_{E_{q_1} \cap E_{q_2}}(e')) = \phi_{q_2}(e') = \gamma_1^{-1} \{\gamma_1(w)/v\}(e') = \gamma_1^{-1}(e') \setminus \{v\} = \lambda(\gamma_1^{-1}(e')) = \lambda(\phi_{q_1}(e'))$. Similarly, if $e' = \gamma_1(w)$ then $\phi_{q_2}(id_{E_{q_1} \cap E_{q_2}}(e')) = \phi_{q_2}(e') = \gamma_1^{-1} \{\gamma_1(w)/v\}(e') = \gamma_1^{-1}(e') \cup \{v\} = \lambda(\gamma_1^{-1}(e')) = \lambda(\phi_{q_1}(e'))$. Note that there are no new entities both in the concrete and in the symbolic model, therefore (ii) holds.

The proof of condition 4.b follows in a straightforward manner from the above.

- **case del(v).** Let $q_1 = (\text{del}(v), E_1, \gamma_1) \in Q_{\mathcal{A}_p}$ and $\psi(q_1) = (\text{del}(v), E_{\psi(q_1)})$. According to the rule for **del** in Table 4.4.3 we have

$$\text{del}(v), E_{\psi(q_1)} \rightarrow_{\lambda} \text{skip}, E_{\psi(q_1)} \setminus \{X_i\}. \quad (\text{A.10})$$

where $X_i = \gamma_1^{-1}(\gamma_1(v))$. In the concrete model we have

$$\text{del}(v), E_1, \gamma_1 \rightarrow \text{skip}, E_1 \setminus \{\gamma_1(v)\}, \gamma_1 \{\perp/v\}. \quad (\text{A.11})$$

We show that there is correspondence between the target states of these transitions.

$$\begin{aligned} & E_{\psi(q_2)} \\ &= \{\gamma_1 \{\perp/v\}^{-1}(e') | e' \in E_1 \setminus \{\gamma_1(v)\}\} \\ &= \{\gamma_1^{-1}(e') | e' \in E_1 \setminus \{\gamma_1(v)\}\} \\ &= \{\gamma_1^{-1}(e') | e' \in E_1\} \setminus \{\gamma_1^{-1}(\gamma_1(v))\} \\ &= E_{\psi(q_1)} \setminus \{X_i\}. \end{aligned}$$

We can conclude that indeed $\psi(q_2) = (\text{skip}, E_{\psi(q_2)} \setminus \{X_i\})$. We must now show that conditions (i) and (ii) of Definition 4.3.8 hold. If $e \in E_1$ and $e \neq \gamma_1(v)$ then $\phi_{q_2}(id_{E_{q_1} \cap E_{q_2}}(e)) = \phi_{q_2}(e) = \gamma_1^{-1}\{\perp/v\}(e) = \gamma_1^{-1}(e) = \lambda(\gamma_1^{-1}(e)) = \lambda(\phi_{q_1}(e))$. If $e = \gamma_1(v)$ then $e \notin E_1 \cap E_2$ therefore there is nothing to prove because $e \notin \text{dom}(\phi_{q_2} \circ id_{E_{q_1} \cap E_{q_2}})$ and $e \notin \text{dom}(\lambda \circ \phi_{q_1})$.

Furthermore, there is no new entity after the transition in either symbolic or the concrete model. Thus, we conclude that condition 4.a holds.

We prove condition 4.b for $s_1 = \text{del}(v)$. Consider the transition (A.11). We have seen that $\psi(q_1)$ has only one possible transition, namely (A.10) and we have proved that indeed the former transition is the expansion of the latter one using λ as defined by rule for del of the symbolic model. Hence we conclude that for the $\text{del}(v)$ case, 4.b is fulfilled.

(Step) We only present the proof for sequential composition. The proofs for the other cases are conducted in a similar way and are omitted here. The case $s = \text{skip}; s_2$ is trivial. Assume $s = s_1; s_2$ and $q_1 = s_1; s_2, E_1, \gamma_1$ with $s_1 \neq \text{skip}$. We prove that any transition

$$\psi(s_1; s_2, E_1, \gamma_1) \rightarrow_\lambda s'_1; s_2, E_{\psi(q_2)} \quad (\text{A.12})$$

is expanded by transition $s_1; s_2, E_1, \gamma_1 \rightarrow q_2$. Transition (A.12) is only possible if:

$$\psi(s_1, E_1, \gamma_1) \rightarrow s'_1, E_{\psi(q_2)}. \quad (\text{A.13})$$

By the induction hypothesis, there exists a transition $s_1, E_1, \gamma_1 \rightarrow s'_1, E_2, \gamma_2$ that expands (A.13). Therefore, $s_1; s_2, E_1, \gamma_1 \rightarrow s'_1; s_2, E_2, \gamma_2$, that exists by the rules of Table 4.4.2, expands (A.12).

For condition 4.b, consider transition $s_1; s_2, E_1, \gamma_1 \rightarrow q_2$ in the concrete semantics. By the rule for sequential composition in Table 4.4.2 there must be a transition $s_1, E_1, \gamma_1 \rightarrow s'_1, E_2, \gamma_2$. By the induction hypothesis, this expands transition $\psi(s_1, E_1, \gamma_1) \rightarrow_\lambda s'_1, E_{\psi(q_2)}$. Thus, $s_1; s_2, E_1, \gamma_1 \rightarrow q_2$ expands $\psi(s_1; s_2, E_1, \gamma_1) \rightarrow_\lambda s'_1; s_2, E_{\psi(q_2)}$. \square

Proposition A.3.1. For all $q \in Q_{\mathcal{H}_p} : |E_q| \leq |\text{PVAR}|$.

Proof. The set E_q is a partition of $A \subseteq \text{PVAR}$. Thus, $|E_q| \leq |A| \leq |\text{PVAR}|$. \square

Theorem 4.4.10 For any $p \in \mathcal{L}$: \mathcal{H}_p is finite state.

Proof. We show that

$$|Q_{\mathcal{H}_p}| \leq |s_{\max}|^m \cdot \left[1 + 2 \cdot \sum_{k=1}^{|\text{PVAR}|} \binom{|\text{PVAR}|}{k} B_k \right]$$

where B_k is the number of partitions of a set of k elements, $|s_{\max}|$ the size of the longest sequential statement in p and m the number of sequential components

of p . In fact, since we are interested in the partition of subsets of PVAR that solely consist of defined variables, B_k corresponds to the number of partitions when k variables are defined. There are $\binom{|\text{PVAR}|}{k}$ different ways to choose k distinct variables from the set PVAR. Therefore, we have $\binom{|\text{PVAR}|}{k} B_k$ partitions for k defined variables. Finally, we have to consider all possible k such that $0 < k \leq |\text{PVAR}|$. The constant 2 considers the possibility to have a bounded or unbounded state. As all variables can be undefined (i.e., $E = \emptyset$), one has to be added. \square

A.4 Proofs of Section 4.5

Lemma 4.5.3 $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{H}_\delta)$.

Proof. [$\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{H}_\delta)$] Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$, with $\sigma = E_0 \lambda_0 E_1 \lambda_1 \dots$. Let $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ be the run generating (σ, N, θ) by some generator $(h_i)_{i \in \mathbb{N}}$. We define a run ρ' of \mathcal{H}_δ that generates (σ, N, θ) in the following way. Let $\rho' = q'_0 \lambda_0 q'_1 \lambda'_1 \dots$ such that $q'_0 = (q_0, h_0(N))$ and for all $i \geq 0$, $q'_{i+1} = (q_{i+1}, E_{q_{i+1}} \setminus \text{cod}(\lambda_i))$. According to the definition of duplication, we have $q'_i \in Q'$ for $i \geq 0$. Furthermore, since $q_i \rightarrow_{\lambda_i} q_{i+1}$ ($i \geq 0$) then there exists a corresponding transition $q'_i \rightarrow_{\lambda_i} q'_{i+1}$ in \mathcal{H}_δ . As q'_0 is an initial state of \mathcal{H}_δ and for every accept state q_i visited infinitely often, also the corresponding accept state q'_i is visited infinitely often, we conclude that $\rho' \in \text{runs}(\mathcal{H}_\delta)$. Finally, since $E_{q'_i} = E_{q_i}$ for all $i \geq 0$, and ρ' has the same ∞ -reallocations λ_i as ρ , then the generator $(h_i)_{i \in \mathbb{N}}$ generates (σ, N, θ) also from run ρ' . Hence we conclude that $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}_\delta)$.

[$\mathcal{L}(\mathcal{H}_\delta) \subseteq \mathcal{L}(\mathcal{H})$] Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}_\delta)$, with $\sigma = E_0 \lambda_0 E_1 \lambda_1 \dots$, and let $\rho = q'_0 \lambda_0 q'_1 \lambda_1 \dots$ be the run generating (σ, N, θ) . We define $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ such that for all $i > 0$, where $q'_i = (q_i, M_i)$. Since $\rho' \in \text{runs}(\mathcal{H}_\delta)$ implies $\rho \in \text{runs}(\mathcal{H})$ and ρ' generates (σ, N, θ) by a generator $(h_i)_{i \in \mathbb{N}}$ implies ρ generates (σ, N, θ) by the same $(h_i)_{i \in \mathbb{N}}$, we conclude that $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$. \square

Notation. Recall that

$$\begin{aligned} K(\phi) &= \max\{|\text{fv}(\psi)| \mid \psi \in CL(\phi)\} \quad \text{and} \\ \Omega(\alpha) &= |\{a \in A \mid \alpha(a) = \infty\}|. \end{aligned}$$

Furthermore, note that since $\Xi = \text{dom}(\Theta)$ for all valuations (ϕ, Ξ, Θ) , we can drop the Ξ -component without loss of information. Finally, in the following, from an arbitrary partial mapping $\theta: \text{LVAR} \rightarrow \text{Ent}$ we derive a partition-based mapping partial $[\theta]: \mathbf{2}^{\text{LVAR}} \rightarrow \text{Ent}$ as follows:

$$[\theta]: X \mapsto e \quad \text{if } X = \theta^{-1}(e).$$

The following lemma is an auxiliary result needed for the proof of Prop 4.5.16.

Lemma A.4.1. For any path $\pi = (q_0, D_0, k_0)\lambda_0(q_1, D_1, k_1) \cdots$ and for any generator $(h_j)_{j \in \mathbb{N}}$ of allocation sequences generated by the underlying run ρ of π :

$$k_0 = \min(K(\phi), \Omega(h_0)) \text{ implies } \forall j \in \mathbb{N} : k_j = \min(K(\phi), \Omega(h_j)).$$

Proof. By inductive argument, assume that $k_j = \min(K(\phi), \Omega(h_j))$. By Definition 4.5.11, $k_{j+1} = \min(K(\phi), k_j + \Omega(\lambda_j))$. There are two cases. On the one hand, if $\lceil q_{j+1} \rceil$ then trivially we have $k_{j+1} = 0 = \Omega(h_{j+1}) = \min(K(\phi), \Omega(h_{j+1}))$.

On the other hand, assume $\lfloor q_{j+1} \rfloor$. Then $k_{j+1} \geq 0$ and we distinguish two further cases:

- if $k_j = K(\phi)$ then by inductive hypothesis $\Omega(h_j) \geq K(\phi)$ and $k_j + \Omega(\lambda_j) \geq K(\phi)$ that in turn implies, together with $\lfloor q_{j+1} \rfloor$ that $\Omega(h_{j+1}) \geq K(\phi)$ and $k_{j+1} = K(\phi)$. We conclude that $k_{j+1} = \min(K(\phi), \Omega(h_{j+1}))$.
- if $k_j = \Omega(h_j) < K(\phi)$ then by Definition 4.5.11 and by inductive hypothesis, $k_{j+1} = \min(K(\phi), k_j + \Omega(\lambda_j)) = \min(K(\phi), \Omega(h_j) + \Omega(\lambda_j)) = \min(K(\phi), \Omega(h_{j+1}))$, as every imploded entity is preserved in the transition.

Hence, we conclude that $k_j = \min(K(\phi), \Omega(h_j))$ for all $j \geq 0$. □

Lemma A.4.2. Let \mathcal{H} be a HABA and ϕ an $\mathcal{A}\ell\ell$ TL-formula. Let $\pi = (q_0, D_0, k_0)\lambda_0(q_1, D_1, k_1)\lambda_1 \cdots$ be a path of $G_{\mathcal{H}}(\phi)$ with underlying run $\rho = q_0\lambda_0q_1\lambda_1 \cdots$, and let σ be an allocation sequence generated by ρ with generator $(h_j)_{j \in \mathbb{N}}$, such that $k_j = \min(K(\phi), \Omega(h_j))$ for all $j \geq 0$. Then for all $i \geq 0$, $\psi \in CL(\phi)$ and $\theta: fv(\psi) \rightarrow Ent$:

$$\sigma^i, N_i^\sigma, \theta \models \psi \iff (\psi, h_i \circ [\theta]) \in D_i. \quad (\text{A.14})$$

Proof. First of all we prove that there exists an allocation sequence σ generated by ρ with a generator $(h_j)_{j \in \mathbb{N}}$ such that $k_j = \min(K(\phi), \Omega(h_j))$ for all $j \geq 0$. By Lemma A.4.1, it is enough to show that there exists among the allocation sequences generated by ρ , one with $\Omega(h_0) = k_0$ (recall that $k_0 \leq K(\phi)$). We distinguish two cases:

- $\lceil q_0 \rceil$ then $k_0 = 0$ and ρ generates only one sequence (up to isomorphism), i.e., precisely the one such that $\Omega(h_0) = 0$.
- $\lfloor q_0 \rfloor$ then ρ generates every sequence with an arbitrary number of (initial) imploded entities. Clearly, among these sequences there exists a σ such that $\Omega(h_0) = k_0$.

Now, we can prove the statement (A.14) by induction on the structure of ψ .

Base of induction• **case $\psi = x \text{ new}$.**

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models x \text{ new}$. This implies $x \in \text{dom}(\theta)$ and $\theta(x) = e$ for some $e \in N_i^\sigma$. By definition of the generator, $h_i(e) \in N_{q_i}$; therefore, $(x \text{ new}, \{x\} \mapsto h_i(e)) \in D_i$ by Defs. 4.5.5 and 4.5.8. Since $[\theta] = (\{x\} \mapsto e)$ we are done.

[\Leftarrow] Suppose $(x \text{ new}, \Theta) \in D_i$ where $\Theta = h_i \circ [\theta]$. It follows that $x \in X \in \text{dom}(\Theta)$ such that $\Theta(X) \in N_{q_i}$; by the definition of generator and the construction of $[\theta]$ it follows that $\theta(x) = h_i^{-1}(\Theta(X)) \in N_i^\sigma$. Thus we have $\sigma^i, N_i^\sigma, \theta \models x \text{ new}$.

• **case $\psi = (x = y)$.**

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models x = y$. This implies $x, y \in \text{dom}(\theta)$ and $\theta(x) = e = \theta(y)$ for some $e \in E_i^\sigma$; hence $[\theta] = (\{x, y\} \mapsto e)$. We have two cases:

1. $h_i(e) \neq \infty$. By definition of AV_{q_i} , we have $(x = y, \{x, y\} \mapsto h_i(e)) \in D_i$.
2. $h_i(e) = \infty$. Since $K(\phi) \geq K(\psi) = 2$, it follows that

$$k_i = \min(K(\phi), \Omega(h_i)) \geq 1.$$

Hence, by the definition of atoms (Definition 4.5.8), we have $(x = y, \{x, y\} \mapsto h_i(e)) \in D_i$.

[\Leftarrow] Suppose $(x = y, \Theta) \in D_i$ with $\Theta = h_i \circ [\theta]$. It follows that $\text{dom}(\Theta) = \{\{x, y\}\}$ and $\Theta(\{x, y\}) = h_i(\theta(x)) = h_i(\theta(y))$; hence $x, y \in \text{dom}(\theta)$ and $\theta(x) = \theta(y)$ (by the construction of $[\theta]$). It follows that $\sigma^i, N_i^\sigma, \theta \models x = y$.

Inductive step• **case $\psi = \exists x.\psi'$.**

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models \exists x.\psi'$. It follows that there exists an $e \in E_i^\sigma$ such that $\sigma^i, N_i^\sigma, \theta\{e/x\} \models \psi'$. By general assumption (see Page 104), $x \in \text{fv}(\psi')$ and hence $\text{dom}(\theta\{e/x\}) \subseteq \text{fv}(\psi')$; hence by the induction hypothesis, we have $(\psi', \Theta') \in D_i$ such that $\Theta' = h_i \circ [\theta\{e/x\}]$. Note that $x \in \bigcup \text{dom}(\Theta')$. It is not difficult to check that, for $\Theta = \Theta' \upharpoonright \text{fv}(\psi)$,

$$\Theta = \{(X \setminus \{x\}, \Theta'(X)) \mid X \in \text{dom}(\Theta'), X \neq \{x\}\} = h_i \circ [\theta]$$

(using in particular that $x \notin \text{dom}(\theta)$). It follows (by Definition 4.5.8 of atoms) that $(\exists x.\psi', \Theta) \in D_i$.

[\Leftarrow] Suppose $(\exists x.\psi', \Theta) \in D_i$ such that $\Theta = h_i \circ [\theta]$. This implies by the definition of atom that $\exists(\psi', \Theta') \in D_i$ where

1. $\Theta = \Theta' \upharpoonright (\exists x.\psi') (= \{(X \setminus \{x\}, \Theta'(X)) \mid X \in \text{dom}(\Theta'), X \neq \{x\}\})$;

2. $x \in \bigcup \text{dom}(\Theta')$;
3. $\Omega(\Theta') \leq k_i$.

We now want to construct $\theta' = \theta\{e/x\}$, in such a way that $\Theta' = h_i \circ [\theta']$. This boils down to choosing an appropriate e . There are three possibilities, based on $X \in \text{dom}(\Theta')$ such that $x \in X$:

- $\Theta'(X) \neq \infty$; then $e = \Theta'(X)$ is appropriate.
- $X \supset \{x\}$; then $e = \theta(y)$ for $y \in X \setminus \{x\}$ is appropriate.
- $\Theta'(X) = \infty$ and $X = \{x\}$. Then $\Omega(\Theta') = \Omega(\Theta) + 1$, hence $k_i \geq \Omega(\Theta) + 1$. It follows that $h_i(e) = \infty$ for some $e \notin \text{cod}(\theta)$; this e is appropriate.

By the induction hypothesis, it follows that $\sigma^i, N_i^\sigma, \theta' \models \psi'$. But then also $\sigma^i, N_i^\sigma, \theta \models \psi$.

- **case** $\psi = \neg\psi'$.

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models \neg\psi'$. This implies $\sigma^i, N_i^\sigma, \theta \not\models \psi'$. By the induction hypothesis, it follows that $(\psi', h_i \circ [\theta]) \notin D_i$. But then (by the definition of atom) $(\neg\psi', h_i \circ [\theta]) \in D_i$.

[\Leftarrow] Inverse to the above.

- **case** $\psi = \psi_1 \vee \psi_2$.

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models \psi_1 \vee \psi_2$. This implies either $\sigma^i, N_i^\sigma, \theta \models \psi_1$ or $\sigma^i, N_i^\sigma, \theta \models \psi_2$; w.l.o.g. assume the former. Let $\theta_1 = \theta \upharpoonright_{fv(\psi_1)}$; it follows that also $\sigma^i, N_i^\sigma, \theta_1 \models \psi_1$. By the induction hypothesis, we have $(\psi_1, \Theta_1) \in D_i$ for $\Theta_1 = h_i \circ [\theta_1]$. Now let $\Theta = h_i \circ [\theta]$. Due to $\Omega(\Theta) \leq fv(\psi) \leq K(\phi)$, $\Omega(\Theta) \leq \Omega(h_i)$ and $k_i = \min(K(\phi), \Omega(h_i))$ we may conclude that $\Omega(\Theta) \leq k_i$. Since (as is easily checked) $\Theta_1 = \Theta \upharpoonright \psi_1$, we may conclude (by the definition of atom) that $(\psi_1 \vee \psi_2, \Theta) \in D_i$.

[\Leftarrow] Suppose $(\psi_1 \vee \psi_2, \Theta) \in D_i$ such that $\Theta = h_i \circ [\theta]$. By the definition of atom, this implies either $(\psi_1, \Theta \upharpoonright \psi_1) \in D_i$ or $(\psi_2, \Theta \upharpoonright \psi_2) \in D_i$; w.l.o.g. assume the former. Again, it is not difficult to see that $\Theta \upharpoonright \psi_1 = h_i \circ [\theta_1]$ where $\theta_1 = \theta \upharpoonright_{fv(\psi_1)}$; hence by the induction hypothesis, $\sigma^i, N_i^\sigma, \theta_1 \models \psi_1$. But then also $\sigma^i, N_i^\sigma, \theta \models \psi_1$ and hence $\sigma^i, N_i^\sigma, \theta \models \psi_1 \vee \psi_2$.

- **case** $\psi = X\psi'$.

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models X\psi'$. It follows that $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \models \psi'$. By the induction hypothesis, $(\psi', \Theta) \in D_{i+1}$ such that $\Theta = h_{i+1} \circ [\lambda_i^\sigma \circ \theta]$; moreover, $h_{i+1} \circ \lambda_i^\sigma = \lambda_i \circ h_i$. Since λ_i^σ is injective, it follows that $[\lambda_i^\sigma \circ \theta] = \lambda_i^\sigma \circ [\theta]$ and hence $\Theta = \lambda_i \circ h_i \circ [\theta]$. By the definition of transitions in the tableau graph (Def. 4.5.11), we may conclude that $(X\psi', \Theta) \in D_i$.

[\Leftarrow] Suppose $(X\psi', \Theta) \in D_i$ with $\Theta = h_i \circ [\theta]$. Due to the definition of transitions in the tableau graph, we may conclude $(\psi', \lambda_i \circ \Theta) \in D_{i+1}$.

Note that (just as above) $\lambda_i \circ \Theta = h_{i+1} \circ [\lambda_i^\sigma \circ \theta]$ and hence (due to the induction hypothesis) $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \models \psi'$. It follows that $\sigma^i, N_i^\sigma, \theta \models \mathsf{X}\psi'$.

- **case** $\psi = \psi_1 \mathsf{U} \psi_2$. For this case we repeatedly use the following correspondence, which holds for all $j \geq i$ (provable by induction on j , using the properties of the generator $(h_i)_{i \in \mathbb{N}}$, see Def. 4.3.6):

$$\lambda_{j-1} \circ \cdots \circ \lambda_i \circ h_i \circ [\theta] = h_j \circ [\lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta] . \quad (\text{A.15})$$

[\Rightarrow] Suppose $\sigma^i, N_i^\sigma, \theta \models \psi_1 \mathsf{U} \psi_2$. It follows that there is a $n \geq i$ such that

1. $\sigma^j, N_j^\sigma, \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta \models \psi_1$ for all $i \leq j < n$;
2. $\sigma^n, N_n^\sigma, \lambda_{n-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta \models \psi_2$.

By the induction hypothesis, and using (A.15), it follows that

1. $(\psi_1, \lambda_{j-1} \circ \cdots \circ \lambda_i \circ h_i \circ [\theta] \upharpoonright \psi_1) \in D_j$ for all $i \leq j < n$;
2. $(\psi_2, \lambda_{n-1} \circ \cdots \circ \lambda_i \circ h_i \circ [\theta] \upharpoonright \psi_2) \in D_n$.

But then one can show (using the definition of atom) that also

$$(\mathsf{X}(\psi_1 \mathsf{U} \psi_2), \lambda_{j-1} \circ \cdots \circ \lambda_i \circ [\theta] \upharpoonright \psi_1) \in D_j$$

for all $i \leq j < n$. (This is shown by induction starting at $j = n - 1$ and going down to $j = i$.) We may conclude that, in all cases, $(\psi_1 \mathsf{U} \psi_2, h_i \circ [\theta]) \in D_i$.

[\Leftarrow] Suppose $(\psi_1 \mathsf{U} \psi_2, \Theta) \in D_i$ with $\Theta = h_i \circ [\theta]$. By Definition 4.5.14 (condition 3) there exists an $n \geq i$ such that $(\psi_2, \lambda_{n-1} \circ \cdots \circ \lambda_i \circ \Theta \upharpoonright \psi_2) \in D_n$. Let n be the smallest such; then it follows (due to Def. 4.5.8) that $(\psi_1, \lambda_{j-1} \circ \cdots \circ \lambda_i \circ \Theta \upharpoonright \psi_1) \in D_j$ for all $i \leq j < n$. (This is proved by induction on $j \in \{i, \dots, n-1\}$, using the fact that $(\psi_2, \lambda_{j-1} \circ \cdots \circ \lambda_i \circ \Theta \upharpoonright \psi_2) \notin D_j$.) Using (A.15) we get

- $\sigma^j, N_j^\sigma, \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta \models \psi_1$ for all $i \leq j < n$;
- $\sigma^n, N_n^\sigma, \lambda_{n-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta \models \psi_2$.

This implies $\sigma^i, N_i^\sigma, \theta \models \psi_1 \mathsf{U} \psi_2$.

□

Proposition 4.5.16. A path π in $G_{\mathcal{H}}(\phi)$ fulfils ϕ if and only if there exists $(\phi, \Theta) \in D_0$ (for some Θ) such that $I_{\mathcal{H}}(q_0) = \overline{\Theta}$.

Proof. (**only if**) By definition, if π fulfils ϕ there exist a (σ, N, θ) generated from the underlying run ρ by a generator $(h_i)_{i \in \mathbb{N}}$ such that $\sigma, N, \theta \models \phi$ and

$k_0 = \min(K(\phi), \Omega(h_0))$. By Lemma A.4.2, $(\phi, h_0 \circ [\theta]) \in D_0$. Furthermore, we have $h_0 \circ [\theta] = h_0 \circ \theta = I_{\mathcal{H}}(q_0)$.

(if) By Lemma A.4.2, taking the triple (σ, N, θ) generated by $(h_i)_{i \in \mathbb{N}}$ such that $k_0 = \min(K(\phi), \Omega(h_0))$ if $(\phi, \Theta) \in D_0$ with $I_{\mathcal{H}}(q_0) = h_0 \circ \theta = h_0 \circ [\theta] = \Theta$ then $\sigma, N, \theta \models \phi$. The existence of the right allocation sequence with the right number of initial imploded entities is explicitly proved. Thus, ϕ is satisfiable. Since (σ, N, θ) is generated by the underlying run of π and the condition on k_0 is satisfied, by definition it follows that π fulfils ϕ . \square

Proposition 4.5.17. ϕ is \mathcal{H} -satisfiable if and only if there exists a path in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ .

Proof.

[\Leftarrow] If there exists π in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ , by definition this implies that ϕ is satisfied by an allocation triple (σ, N, θ) generated by the underlying run of π .

[\Rightarrow] Now assume that ϕ is \mathcal{H} -satisfiable, and let $\rho = q_0 \lambda_0 q_1 \lambda_1 \cdots$ be a run generating a triple (σ, N, θ) with generator $(h_i)_{i \in \mathbb{N}}$ such that $\sigma, N, \theta \models \phi$. We construct a path $\pi = (q_0, D_0, k_0) \lambda_0 (q_1, D_1, k_1) \lambda_1 \cdots$ that fulfils ϕ . For all $i \in \mathbb{N}$ let

$$D_i = \{(\psi, h_i \circ [\theta]) \mid \psi \in CL(\phi), \sigma^i, N_i^\sigma, \theta \models \psi\}$$

and

$$k_i = \min(K(\phi), \Omega(h_i)).$$

It can be proved (by induction on the structure of the formulae in $CL(\phi)$) that the D_i and k_i satisfy the conditions of Definition 4.5.8 i.e., (q_i, D_i, k_i) is an atom for all $i \in \mathbb{N}$. We show that π is a path by proving that the conditions of Definition 4.5.14 are satisfied by π . Since π then fulfils ϕ by construction, we are done.

1. By the construction of π .
2. By contradiction. Assume that there exists an $i \geq 0$ such that $(q_i, D_i, k_i) \rightarrow_{\lambda_i} (q_{i+1}, D_{i+1}, k_{i+1})$ is not a transition of G . Take the least such i . Then one of the following must hold:
 - i) $q_i \rightarrow_{\lambda_i} q_{i+1}$ is not a transition in \mathcal{H} . But this contradicts the fact that ρ is a run of \mathcal{H} .
 - ii) there exists $(X\psi, \Theta) \in D_i$ such that $(\psi, \lambda_i \circ \Theta) \notin D_{i+1}$. By the properties of the generator we have that $\lambda_i \circ \Theta = h_{i+1} \circ [\lambda_i^\sigma \circ \theta]$ (see also (A.15)); hence this would imply $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \models \psi$. But then also $\sigma^i, N_i^\sigma, \theta \not\models X\psi$, contradicting the construction of D_i .

If $(\psi, \lambda_i \circ \Theta) \in D_{i+1}$, but $(X\psi, \Theta) \notin D_i$ then again for the properties of the generator, $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \models \psi$ and by the semantics of \mathcal{ALLTL} , $\sigma^i, N_i^\sigma, \theta \models X\psi$. Thus by definition of D we must have $(X\psi, \Theta) \in D_i$. Contradiction.

iii) $k_{i+1} \neq \min(K(\phi), k_i + \Omega(\lambda_i))$. Since $\Omega(h_{i+1}) = \Omega(h_i) + \Omega(\lambda_i)$, this is also contradictory.

3. Assume $(\psi_1 \cup \psi_2, \Theta) \in D_i$. By the construction of D_i , $\sigma^i, N_i^\sigma, \theta \models \psi_1 \cup \psi_2$ and $\Theta = h_i \circ [\theta]$. Therefore $\sigma^j, N_j^\sigma, \lambda_{j-1}^\sigma \circ \dots \circ \lambda_i^\sigma \circ \theta \upharpoonright fv(\psi_2) \models \psi_2$ for some $j \geq i$. Due to (A.15) and the construction of D_j , it follows that $(\psi_2, \lambda_{j-1} \circ \dots \circ \lambda_i \circ \Theta \upharpoonright \psi_2) \in D_j$.

□

Proposition 4.5.20. If π is fulfilling path in $G_{\mathcal{H}}(\phi)$, then $Inf(\pi)$ is a self-fulfilling SCS of $G_{\mathcal{H}}(\phi)$.

Proof. Let $G' = Inf(\pi)$. G' is strongly connected. From the definition of infinite set it follows that there exists $i \geq 0$ such that the atoms in $\pi^i = (q_i, D_i, k_i)\lambda_i(q_{i+1}, D_{i+1}, k_{i+1})\lambda_{i+1} \dots$ are precisely those in G' . Furthermore, if there is an atom $A \in G'$ such that $(\psi_1 \cup \psi_2, \Theta) \in D_A$, then there exists $j \geq i : (q_j, D_j, k_j) = A$. By condition 3 of Def. 4.5.14 we have that there exists $n \geq j$ such that $(\psi_2, \lambda_{n-1} \circ \dots \circ \lambda_j \circ \Theta \upharpoonright \psi_2) \in D_n$. But then $(q_n, D_n, k_n) \in G'$, hence G' is self-fulfilling. □

Proposition 4.5.21. Let $G' \subseteq G_{\mathcal{H}}(\phi)$ be self-fulfilling SCS such that

- there exists a fulfilling prefix of G' starting at an initial atom A with $(\phi, \Theta) \in D_A$ such that $I_{\mathcal{H}}(q_A) = \overline{\Theta}$;
- for all $F \in \mathcal{F}_{\mathcal{H}} : F \cap \{q_B | B \in G'\} \neq \emptyset$;

Then there exists a path π in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ and such that $Inf(\pi) = G'$.

Proof. Satisfaction of an until-valuation. In a (finite or infinite) transition sequence through $G_{\mathcal{H}}(\phi)$, say $A_0 \rightarrow_{\lambda_0} A_1 \rightarrow_{\lambda_1} \dots$, we call an until-valuation $v = (\psi_1 \cup \psi_2, \Theta) \in D_{A_0}$ *satisfied at A_i* (or just *satisfied*) if $(\psi_2, \lambda_{i-1} \circ \dots \circ \lambda_0 \circ \Theta \upharpoonright \psi_2) \in D_{A_i}$.

Observation. Due to the properties of atoms and of transitions in $G_{\mathcal{H}}(\phi)$, if v is not satisfied at any A_j with $j \leq i$, then it follows that $(\psi_1 \cup \psi_2, \lambda_{j-1} \circ \dots \circ \lambda_0 \circ \theta) \in D_{A_j}$.

By the properties assumed for $G_{\mathcal{H}}(\phi)$, there exists a sequence

$$\pi_1 = A_0 \lambda_0 \dots \lambda_{m-1} A_m$$

such that $A = A_0$, $A_i \rightarrow_{\lambda_i} A_{i+1}$ for all $0 \leq i < m$, and $A_m = B$ is a node of G' . Furthermore, starting from B it is possible to construct a finite transition sequence

$$B = B_0 \rightarrow_{\mu_0} B_1 \rightarrow_{\mu_1} \dots \rightarrow_{\mu_{i-1}} B_i$$

in which all $(\psi_1 \cup \psi_2, \Theta) \in D_B$ are satisfied, in the above sense. The existence of such a transition sequence can be proved by contradiction: suppose that the minimal number of until-valuations $(\psi_1 \cup \psi_2, \Theta) \in D_B$ that remain unsatisfied in any finite fragment starting at B_0 is u (> 0); take an optimal transition sequence that leaves u until-valuations unsatisfied, and let $(\psi_1 \cup \psi_2, \Theta) \in D_B$ be one of the unsatisfied ones. As observed above, it follows that $v' = (\psi_1 \cup \psi_2, \mu_{i-1} \circ \dots \circ \mu_0 \circ \Theta) \in D_{B_i}$. However, due to the fact that G' is self-fulfilling, there is a transition sequence

$$B_i \rightarrow_{\mu_i} \dots \rightarrow_{\mu_{n-1}} B_n$$

that satisfies v' , i.e., such that $(\psi_2, \mu_{n-1} \circ \dots \circ \mu_i \circ \dots \circ \mu_0 \circ \Theta \upharpoonright \psi_2) \in D_{B_n}$. But then the combined sequence starting at B_0 and going through B_i to B_n leaves at most $u - 1$ until-valuations of B_0 unsatisfied; contradiction.

We extend this finite transition sequence to a cycle

$$\pi_2 = B_0 \lambda_0 B_1 \lambda_1 \dots \lambda_{n-1} B_n$$

with $B_n = B_0$, that visits all nodes of G' (note that π_2 exists because G' is strongly connected).

Let $\pi = \pi_1 \cdot \pi_2^\omega$. We show that π is an allocation path. Since (by assumption) $(\phi, \Theta) \in D_A$ such that $I_{\mathcal{H}}(q_A) = \overline{\Theta}$ it then follows (by Prop. 4.5.16) that π fulfils ϕ . Since clearly $\text{Inf}(\pi) = G'$ we are then done.

For this purpose we show that the conditions of Def. 4.5.14 hold.

1. Let $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ be the underlying run of π . Note that for all $i < n$ and for all $k \in \mathbb{N}$, $q_{m+i+n*k} = q_{B_i}$. We show that ρ is a run of \mathcal{H} .

$q_0 \in \text{dom}(I_{\mathcal{H}})$ is guaranteed by assumption on A ; furthermore, $q_i \xrightarrow{\lambda_i} q_{i+1}$ by construction of the graph $G_{\mathcal{H}}(\phi)$. Finally, take an arbitrary $F \in \mathcal{F}_{\mathcal{H}}$. By the assumption in the proposition, we have that $q_B \in F$ for some $B \in G'$; hence $B = B_i$ for some $i < n$. Since then $q_{m+i+k*n} = q_B$ for all $k \in \mathbb{N}$, q_B is visited infinitely often by ρ .

2. By construction of π .
3. Assume $v = (\psi_1 \cup \psi_2, \Theta)$ is in one of the atoms in π . We have to show that v is satisfied somewhere during the sequence. We distinguish three cases:

- If $v \in D_{A_i}$ for $i < m$ then either $(\psi_2, \lambda_{j-1} \circ \dots \circ \lambda_i \circ \Theta \upharpoonright \psi_2) \in D_{A_j}$ for some $i \leq j < m$, or $(\psi_1 \cup \psi_2, \lambda_{m-1} \circ \dots \circ \lambda_i \circ \Theta) \in D_{B_0}$. The latter case is dealt with below.
- If $v \in D_{B_i}$ for $0 < i < n$, then either $(\psi_2, \lambda_{j-1} \circ \dots \circ \lambda_i \circ \Theta \upharpoonright \psi_2) \in D_{B_j}$ for some $i \leq j < n$, or $(\psi_1 \cup \psi_2, \lambda_{n-1} \circ \dots \circ \lambda_i \circ \Theta) \in D_{B_0}$. The latter case is dealt with below.
- Otherwise, $v \in D_{B_0}$. Due to the construction of π , v is satisfied in one of the B_i ($0 \leq i < n$).

□

Theorem 4.5.22. For any HABA \mathcal{H} and formula ϕ , it is decidable whether or not ϕ is \mathcal{H} -satisfiable.

Proof. By Propositions 4.5.16 and 4.5.17, in order to prove that ϕ is \mathcal{H} -satisfiable, it is necessary and sufficient to search in the graph $G_{\mathcal{H}}(\phi)$ for a fulfilling path π . By Propositions 4.5.21, it is necessary and sufficient to check only for a self-fulfilling SCS that has a fulfilling prefix whose initial atom (q_0, D_0, k_0) contains (ϕ, Θ) for some Θ such that $\bar{\Theta} = I_{\mathcal{H}}(q_0)$ and has a non-empty intersection with every set of final states. Since SCS are finite objects and there are only a finite number of them, this search can be effective and exhaustive. \square

B

Proofs of Chapter 5

B.1 Proofs of Sections 5.3 and 5.4

Proposition 5.3.16 Let γ_1 and γ_2 be two configurations. If $\gamma_1 \xrightarrow{h} \gamma_2$ then $\gamma_1 \xrightarrow{\lambda} \gamma_2$ where let $e \in E_{\gamma_1}$ and $e' \in E_{\gamma_2}$:

$$\lambda(e, e') = \begin{cases} \mathcal{C}_{\gamma_1}(e) & \text{if } e' = h(e) \\ 0 & \text{otherwise.} \end{cases}$$

Proof. It is enough to prove that λ satisfies each of the conditions of Definition 5.3.12.

1. This condition is trivially satisfied since h is a morphism and therefore a function. Thus the complete cardinality of an entity e is transferred to $h(e)$.
2. Follows by condition 4m of Definition 5.3.3.
3. Straightforward since h is a function.
4. Straightforward since $\lambda(e)$ contains only one element.
5. Straightforward from 1m of Definition 5.3.3.

□

Lemma 5.3.19 If $\gamma_1 \xrightarrow{\lambda} \gamma_2$ and $\gamma'_1 \xrightarrow{\lambda'} \gamma'_2$ and $\lambda' \triangleright \lambda$ via h_1 and h_2 then:

- a) $E_{\gamma'_2} \setminus \text{cod}(\lambda') = h_2^{-1}(E_{\gamma_2} \setminus \text{cod}(\lambda))$
 b) $E_{\gamma_2}^* \subseteq \text{cod}(\lambda)$.

Proof.

- a) we prove part a) by showing the two set inclusions.

' \subseteq ' Let $e \in E_{\gamma_2} \setminus \text{cod}(\lambda)$, by condition 4 of Definition 5.3.17 we have $\mathcal{C}_{\gamma_2}(h_2(e)) \neq *$. By contradiction assume $h_2(e) \in \text{cod}(\lambda)$, this implies that there exists $e_1, \dots, e_k \in E_{\gamma_1}$ ($k > 0$) such that

$$\lambda(e_1, h_2(e)) \oplus \dots \oplus \lambda(e_k, h_2(e)) = \mathcal{C}_{\gamma_2}(h_2(e)) \neq *$$

By condition 2 of \triangleright we have:

$$\sum_{e_1, h_2(e) = (h_1(e'_1), h_2(e'_2))} \lambda'(e'_1, e'_2) \oplus \dots \oplus \sum_{e_k, h_2(e) = (h_1(e'_k), h_2(e'_2))} \lambda'(e'_k, e'_2) = \mathcal{C}_{\gamma_2}(h_2(e)) \neq *$$

But since $\mathcal{C}_{\gamma_2}(h_2(e)) \neq *$ there must be $e' \in h_1^{-1}(\{e_1, \dots, e_k\})$ such that $\lambda(e', e) \neq 0$ otherwise the sum could not be precisely $\mathcal{C}_{\gamma_2}(h_2(e)) \neq *$. Therefore $e \in \text{cod}(\lambda')$ that contradict our initial hypothesis.

' \supseteq ' Let $e \in h_2^{-1}(E_{\gamma_2} \setminus \text{cod}(\lambda))$ we prove that $e \in E_{\gamma_2} \setminus \text{cod}(\lambda')$. To this end assume $e \in \text{cod}(\lambda')$ then there exists $e' \neq \perp$ such that $\lambda(e', e) = 1$. This implies by condition 2 of definition \triangleright that $\lambda(h_1(e'), h_2(e)) \geq \lambda(e', e) = 1$. Hence $h_2(e) \in \text{cod}(\lambda)$ that is impossible since we have assumed $e \in h_2^{-1}(E_{\gamma_2} \setminus \text{cod}(\lambda))$.

- b) Let $e \in E_{\gamma_2}^*$. By condition 4 of Definition 5.3.17 we have $h_2^{-1}(e) \subseteq \text{cod}(\lambda')$. Hence, by part a) of the lemma we can conclude that $e \in \text{cod}(\lambda)$. \square

Lemma 5.3.20 (\triangleright transitivity). Let $q_1 \xrightarrow{\lambda} q_2$, $q'_1 \xrightarrow{\lambda'} q'_2$ and $q''_1 \xrightarrow{\lambda''} q''_2$. Then:

$$(\lambda'' \triangleright \lambda' \wedge \lambda' \triangleright \lambda) \Rightarrow \lambda'' \triangleright \lambda.$$

Proof. By Definition 5.3.17 the following morphisms exist:

$$q'_1 \xrightarrow{h_1} q_1, \quad q'_2 \xrightarrow{h_2} q_2, \quad q''_1 \xrightarrow{h'_1} q'_1, \quad q''_2 \xrightarrow{h'_2} q'_2.$$

Therefore by the properties of composition of morphisms we have $q''_1 \xrightarrow{h''_1} q_1$ and $q''_2 \xrightarrow{h''_2} q_2$ defined as

$$\begin{aligned} h''_1 &= h'_1 \circ h_1 \\ h''_2 &= h'_2 \circ h_2. \end{aligned}$$

Thus the first condition of Definition 5.3.17 for $\lambda'' \triangleright \lambda$ is satisfied.

For the second condition, first of all observe that by definition, q'_1, q'_2 and q''_1 and q''_2 are concrete states since reallocations λ' and λ'' are concretions. This implies that h'_1 and h'_2 are bijective. Let $e_1 \in E_{q'_1}$ and $e_2 \in E_{q'_2}$, we have:

$$\begin{aligned} \lambda(e_1, e_2) &: \mapsto \sum_{(e_1, e_2) = (h_1(e'_1), h_2(e'_2))} \lambda'(e'_1, e'_2) && \text{[by hp: } \lambda' \triangleright \lambda] \\ &\mapsto \sum_{(e_1, e_2) = (h_1(h'_1(e''_1)), h_2(h'_2(e''_2)))} \lambda''(e''_1, e''_2) && \text{[by hp: } \lambda'' \triangleright \lambda' \text{ and} \\ & && h'_1, h'_2 \text{ bijective]} \\ &\mapsto \sum_{(e_1, e_2) = (h''_1(e''_1), h''_2(e''_2))} \lambda''(e''_1, e''_2) && \text{[by def. } h''_1, h''_2]. \end{aligned}$$

Hence also the second condition of Definition 5.3.17 holds for $\lambda'' \triangleright \lambda$.

For the condition **(No-Cross)**, we first observe that:

$$\begin{aligned} &h''_1(e''_1) = h''_1(e''_2) \vee h''_2(\lambda''(e''_1)) = h''_2(\lambda''(e''_2)) && \text{[by def. of } h''_1, h''_2] \\ \Rightarrow &h_1(h'_1(e''_1)) = h_1(h'_1(e''_2)) \vee && \\ &h_2(h'_2(\lambda''(e''_1))) = h_2(h'_2(\lambda''(e''_2))) && \text{[by (No-Cross) of } \lambda'] \\ \Rightarrow &(h'_1(e''_1) \prec_{q'_1} h'_1(e''_2)) \iff \lambda'(h'_1(e''_1)) \prec_{q'_2} \lambda'(h'_1(e''_2)) && \text{(B.1)} \end{aligned}$$

Now, note that since $\lambda'' \triangleright \lambda'$, for the second property of concretion and the bijectivity of h'_1, h'_2 , we have $\lambda'(h'_1(e''_1), h'_2(\lambda''(e''_1))) = \lambda''(e''_1, \lambda''(e''_1)) = 1$. Therefore:

$$\lambda'(h'_1(e''_1)) = h'_2(\lambda''(e''_1)) \quad \text{(B.2)}$$

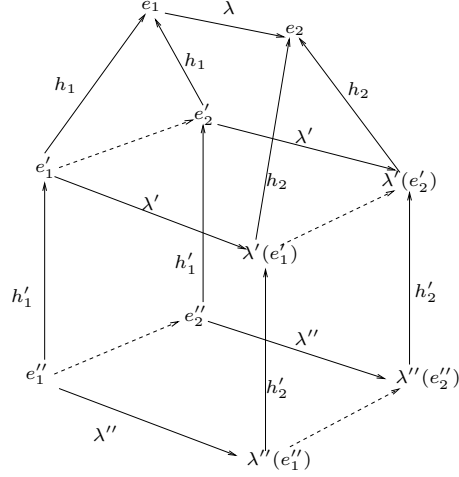
In Figure B.1 this corresponds to say that the front-low part of the diagram commutes. Hence, assume $h''_1(e''_1) = h''_1(e''_2) \vee h''_2(\lambda''(e''_1)) = h''_2(\lambda''(e''_2))$, we have:

$$\begin{aligned} &e''_1 \prec_{q''_2} e''_2 && [q'_1 \text{ concrete and Def. 5.3.3 3m}] \\ \Rightarrow &h'_1(e''_1) \prec_{q'_1} h'_1(e''_2) && \text{[by (B.1)]} \\ \Rightarrow &\lambda'(h'_1(e''_1)) \prec_{q'_2} \lambda'(h'_1(e''_2)) && \text{[by (B.2)]} \\ \Rightarrow &h'_2(\lambda''(e''_1)) \prec_{q'_2} h'_2(\lambda''(e''_2)) && \text{[by Def. 5.3.3 2m]} \\ \Rightarrow &\lambda''(e''_1) \prec_{q''_2} \lambda''(e''_2) \end{aligned}$$

Vice-versa:

$$\begin{aligned} &\lambda''(e''_1) \prec_{q''_2} \lambda''(e''_2) && [q'_2 \text{ concrete and Def. 5.3.3 3m}] \\ \Rightarrow &h'_2(\lambda''(e''_1)) \prec_{q'_2} h'_2(\lambda''(e''_2)) && \text{[by (B.2)]} \\ \Rightarrow &\lambda'(h'_1(e''_1)) \prec_{q'_2} \lambda'(h'_1(e''_2)) && \text{[by (B.1)]} \\ \Rightarrow &h'_1(e''_1) \prec_{q'_1} h'_1(e''_2) \\ \Rightarrow &e''_1 \prec_{q''_2} e''_2 \end{aligned}$$

Therefore also condition **(No-Cross)** holds for $\lambda'' \triangleright \lambda$. Finally, for condition 4, let $e \in E_{q''_2}$. If $\mathcal{C}_{q_2}(h_2(e)) = *$ then $\mathcal{C}_{q_2}(h_2(h'_2(e))) = *$ and because $\lambda' \triangleright \lambda$ (using condition 4) it follows $h'_2(e) \in \text{cod}(\lambda')$ that finally implies (by $\lambda'' \triangleright \lambda'$ applying in particular condition 2) $e \in \text{cod}(\lambda'')$. Since also condition 4 is satisfied we conclude that $\lambda'' \triangleright \lambda$. \square

Figure B.1: Transitivity of concretions: $\lambda'' \triangleright \lambda' \triangleright \lambda$.

Theorem 5.4.5 If $\mathcal{H} \sqsubseteq \mathcal{H}'$ then $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{H}')$.

Proof. Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$. Then there exists a run $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ such that ρ generates (σ, N, θ) via a generator $(h_i)_{i \in \mathbb{N}}$. Since $\mathcal{H} \sqsubseteq \mathcal{H}'$ there exists $q'_0 \in I_{\mathcal{H}'}$ such that $q_0 \sqsubseteq_{h'_0} q'_0$ and moreover for all $i \geq 0$ there exists $h'_i : q_i \twoheadrightarrow q'_i$ and λ'_i such that:

- i) $q_i \twoheadrightarrow_{\lambda'_i} q_{i+1}$ and $q_{i+1} \sqsubseteq_{h'_{i+1}} q'_{i+1}$
- ii) $\lambda_i \triangleright \lambda'_i$ via h'_i, h'_{i+1} ;

Consider the sequence:

$$\rho' = q'_0 \lambda'_0 q'_1 \lambda'_1 \dots$$

where if $q_i \in F \in \mathcal{F}$, then we take — among all states that simulate q_i — an accept state in \mathcal{H}' that satisfies condition 2.b) of Definition 5.4.1, i.e., $q'_i \in \psi(F)$ (where ψ is the bijective function described in that condition). Since ρ is an accepting run of \mathcal{H} , and because of the choice of the accept states on ρ' described above, then also ρ' is an accepting run of \mathcal{H}' .

Hence, in order to prove that $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}')$ we show that it is generated by ρ' by some generator $(h''_i)_{i \in \mathbb{N}}$. This corresponds to prove the existence of such generator. For all $i \in \mathbb{N}$, let

$$h''_i = h'_i \circ h_i.$$

Since h''_i is the composition of two morphisms it is a morphism by Proposition 5.3.7. We show that h''_i satisfies condition (1)-(4) of Definition 5.3.24.

1. since $\lambda_i^\sigma \triangleright \lambda_i \triangleright \lambda'_i$ then by Lemma 5.3.20 it follows $\lambda_i^\sigma \triangleright \lambda'_i$ via h''_i and h''_{i+1} .

2. By definition of simulation we have $I(q'_0) = (N', h_0 \circ \theta')$ where $I(q_0) = (h_0^{-1}(N'), \theta')$ and by definition of generator $N = (h'_0)^{-1}(N')$. Therefore $N = (h''_0)^{-1}(N')$ and $I(q'_0) = (N', h''_0 \circ \theta)$. But then h''_0 that satisfy condition 2 of the definition of generator.

Hence $(h''_i)_{i \in \mathbb{N}}$ is a generator from which it follows $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H}')$. \square

B.2 Proofs of Section 5.5

Proposition 5.6.14 If a state γ is L -canonical then:

- a) γ is PV -reachable;
- b) for every state γ' , if there exists a morphism $h : \gamma \rightsquigarrow \gamma'$ then either $\gamma \cong \gamma'$ or γ' is L -unsafe.

Proof.

- a) Let us partition E_γ in two parts: the set E_γ^{PV} of PV -reachable entities and the set $E_\gamma^{\neg PV}$ given by all the entities not reachable by any program variable. It is clear that $E_\gamma^{\neg PV} \cup E_\gamma^{PV} = E_\gamma$ and $E_\gamma^{\neg PV} \cap E_\gamma^{PV} = \emptyset$. By contradiction assume γ is not PV -reachable, i.e., $E_\gamma^{\neg PV} \neq \emptyset$ and let $e \in E_\gamma^{\neg PV}$. By the properties of L -compactness (that must be satisfied by γ since it is L -canonical), since e is unreachable we have that $\text{indegree}(e) > 1$, i.e., there exists at least e', e'' such that $\mu_\gamma(e') = \mu_\gamma(e'') = e$. Furthermore, it must be $e', e'' \in E_\gamma^{\neg PV}$ otherwise we would have $e \in E_\gamma^{PV}$ that would contradict our original hypothesis. Hence, we can construct the following infinite series of sets:

$$\begin{aligned} A_0 &= \{e\} \\ A_{i+1} &= \{e' \mid \mu_\gamma(e') \in A_i\}. \end{aligned}$$

It can be proved by induction on i that

$$\forall i \in \mathbb{N} : A_i \subseteq E_\gamma^{\neg PV}. \quad (\text{B.3})$$

This implies the following statement (by set theory)

$$\bigcup_{i \in \mathbb{N}} A_i \subseteq E_\gamma^{\neg PV}. \quad (\text{B.4})$$

However, since every entities has only one outgoing reference, it can be also proved by induction, that for all $i \in \mathbb{N}$ we have:

$$\bigcup_{0 \leq j < i} A_j \subset \bigcup_{0 \leq j \leq i+1} A_j.$$

Therefore, $|\bigcup_{i \in \mathbb{N}} A_i| = \omega$ that together with (B.4) implies both $E_\gamma^{\neg PV}$ and E_γ are infinite. But this is impossible since in every state there is only a finite number of live entities.

b) Let γ' be a state such that $\gamma \xrightarrow{h} \gamma'$. We distinguish two cases:

- h is not contractive. Then h must be an isomorphism by Proposition 5.6.3.
- h is contractive. Therefore there exists $e' \in E_{\gamma'}$ such that $|h^{-1}(e')| > 1$, i.e., the pure chain $h^{-1}(e')$ that has more than one entity is contracted in e' . Since γ is L -compact, there exists $e_v \in PV$ such that $d(e_v, \text{last}(h^{-1}(e'))) \leq L + 1$, i.e., the last entity of the pure chain cannot be distant more than $L + 1$ from some program variable (in this case e_v). But, this implies that $d(e_v, \text{first}(h^{-1}(e'))) \leq L$. We have that

$$d(h(e_v), e') \leq L \quad (\text{B.5})$$

(the inequality is strict if h is also contracting for some other e'' preceding e' or if more than two entities are mapped onto e'). However, $|h^{-1}(e')| > 1$ implies $\mathcal{C}(e') > 1$ that together with (B.5) violates the L -safeness condition for e' since $h(e_v) \in PV$ (cf. condition (5.11)). Therefore γ' is L -unsafe. □

Proposition 5.6.15 Let γ_1 and γ_2 be a PV -reachable and a L -canonical configurations, respectively. If $h_1 : \gamma_1 \xrightarrow{\text{h}} \gamma_2$ and $h_2 : \gamma_1 \xrightarrow{\text{h}} \gamma_2$ then $h_1 = h_2$.

Proof. Since γ_1 and γ_2 are PV -reachable, if $e \in E_{\gamma_1}$ then there exists $e_v \in PV$ such that $\mu_{\gamma_1}^n(e_v) = e$ for some $n \geq 0$. According to Definition 5.6.4, the number n is the distance between e_v and e and we denote it by $d(e_v, e)$. Therefore we can define the shortest distance from all the program variables: $d(PV, e) = \min \{d(e_v, e) \neq \perp \mid e_v \in PV\}$. Now, we prove:

$$\forall e \in E_{\gamma_1} : h_1(e) = h_2(e) \quad (\text{B.6})$$

by induction on the distance $d(PV, e)$.

- **Base case** $d(PV, e) = 0$. That is $e \in PV$. By the global constraint (5.11) on page 155 we have $h_1(e) = h_2(e)$.
- **Inductive case** $d(PV, e) = n + 1$. By contradiction assume $h_1(e) \neq h_2(e)$. Since γ_1 is PV -reachable, there exists $e' \prec_1 e$. By condition 3m of Definition 5.3.3 this implies that $h_1(e') \preceq_2 h_1(e)$ and $h_2(e') \preceq_2 h_2(e)$. We have four different cases:
 1. $h_1(e') \prec_2 h_1(e)$ and $h_2(e') \prec_2 h_2(e)$
 2. $h_1(e') =_2 h_1(e)$ and $h_2(e') \prec_2 h_2(e)$
 3. $h_1(e') \prec_2 h_1(e)$ and $h_2(e') = h_2(e)$
 4. $h_1(e') = h_1(e)$ and $h_2(e') = h_2(e)$

Case 1 implies $h_1(e') \neq h_2(e')$ because for every entity there exists only one outgoing reference. But this is a contradiction since, $d(PV, e') = n$, thus, by induction hypothesis, $h_1(e') = h_2(e')$.

Case 4 implies by induction hypothesis, $h_1(e) = h_1(e') = h_2(e') = h_2(e)$ that again is a contradiction because we have assumed $h_1(e) \neq h_2(e)$.

Finally, the more involved cases are 2 and 3. We prove 2 since 3 is symmetrical and can be shown precisely in the same way. Since $h_1(e') = h_1(e)$ and $h_2(e') \prec_2 h_2(e)$ and by induction hypothesis $h_1(e') = h_2(e')$, we have $\mathcal{C}_{\gamma_2}(h_1(e')) = *$ because, one morphism maps on it both e, e' whereas the other only one. Moreover $\{e', e\}$ must be a pure chain since h_1 maps this set in the same entity. In turn, this implies

$$\{h_2(e'), h_2(e)\} \text{ is a pure chain} \quad (\text{B.7})$$

because being a morphism h_2 maps pure chains onto pure chains. Moreover, since γ_2 is L -canonical, $h_1(e')$ is at a distance at least $L + 1$ from every program variables otherwise γ_2 would be not L -safe. But since $h_1(e') = h_2(e')$ and $h_2(e') \prec_2 h_2(e)$ then $\{h_2(e'), h_2(e)\}$ is *not* a pure chain otherwise L -compactness would be violated. But this contradicts (B.7).

□

Theorem 5.6.16 (Existence of the canonical form). For every L -safe and PV -reachable configuration γ there exists an L -canonical configuration γ' and a unique morphism $h : \gamma \rightarrow \gamma'$.

Proof. If γ is L -compact, then it is L -canonical by definition, and we can take $h = id_\gamma$.

Assume then that γ is not L -compact, we construct a L -compact γ' out of γ . Since γ is not L -compact then by definition:

$$\exists e \in E_\gamma : (\text{indegree}(e) = 1 \wedge \forall e' \in PV : d(e', e) > L + 1). \quad (\text{B.8})$$

In other words entities satisfying (B.8) form pure chains distant more than $L + 1$ from every program variable. Let $C_1, \dots, C_n \subseteq E_\gamma$ be *all* such chains. In the construction of γ' , we need to exclude them. Let:

$$\begin{aligned} E_{\gamma'} &= (E_\gamma \setminus (C_1 \cup \dots \cup C_n)) \cup \{\text{first}(C_1), \dots, \text{first}(C_n)\} \\ \mu_{\gamma'} &= (\mu_\gamma \upharpoonright E_\gamma \setminus (C_1 \cup \dots \cup C_n)) \cup \{(\text{first}(C_i), \mu_\gamma(\text{last}(C_i))) \mid 1 \leq i \leq n\} \\ \mathcal{C}_{\gamma'}(e) &= \begin{cases} \mathcal{C}_\gamma(e) & \text{if } e \in E_\gamma \setminus (C_1 \cup \dots \cup C_n) \\ \sum_{e' \in C_i} \mathcal{C}_\gamma(e') & \text{if } e = \text{first}(C_i). \end{cases} \end{aligned}$$

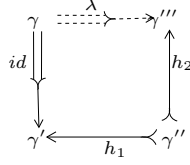


Figure B.2: Diagram of Proposition 5.6.20.

Hence, $\gamma' = (E_{\gamma'}, \mu_{\gamma'}, \mathcal{C}_{\gamma'})$ is L -compact by construction since every pure chain violating the L -compactness condition in γ has been collapsed in the first entity of the chain itself. Note that γ' is L -safe since γ is L -safe and the compacted chains are those in γ that are distant more than $L + 1$ from the program variables, therefore, the multiple, unbounded entities that substitute these chain in γ' are still at a distance at least $L + 1$. We conclude that γ' is indeed L -canonical.

We construct now, a morphism $h : \gamma \rightarrow \gamma'$. For $e \in E_\gamma$ and $1 \leq i \leq n$ let:

$$h(e) = \begin{cases} e & \text{if } e \in E_\gamma \setminus (C_1 \cup \dots \cup C_n) \\ \text{first}(C_i) & \text{if } e \in C_i. \end{cases}$$

As expected, h corresponds to id on those entities that do not violate compactness, otherwise h collapses the pure chains C_1, \dots, C_n into their corresponding first elements. It can be shown that h is a morphism. In fact, by construction h satisfies condition 1m-4m of the Definition 5.3.3. Note that requiring γ to be PV -reachable is essential for the existence of h .

Finally, to complete the proof, we show that h is unique (up to isomorphism). To this end, consider a generic morphism $h' : \gamma \rightarrow \gamma'$. Since γ PV -reachable by hypothesis of this theorem, and γ' is L -canonical by construction, we can conclude by Proposition 5.6.15 that $h = h'$. Therefore, h is unique. \square

In the following lemma we prove that we can complete the diagram reported in Figure B.2 by a reallocation γ .

Proposition 5.6.20 Let γ and γ''' be two L -canonical states. If $\gamma \xrightarrow{id} \gamma' \xleftarrow{h_1} \gamma'' \xrightarrow{h_2} \gamma'''$ such that

- (a) $\forall e \in E_{\gamma''} : id^{-1}(h_1 \circ h_2^{-1}(e)) \subseteq E_\gamma^\perp$ is a chain and
- (b) $\mathcal{C}_{\gamma'''}(e) = * \Rightarrow \perp \notin id^{-1}(h_1 \circ h_2^{-1}(e))$.

Then $\gamma \xrightarrow{\lambda} \gamma'''$ where:

$$\begin{aligned} \lambda = & \{(e_1, e_2, \mathcal{C}_{\gamma''}(h_1^{-1}(e_1) \cap h_2^{-1}(e_2))) \mid e_1 \in E_\gamma\} \cup \\ & \{(e, \perp, \mathcal{C}_\gamma(e)) \mid e \in E_\gamma \setminus E_{\gamma'}\} \cup \\ & \{(\perp, e, \mathcal{C}_{\gamma'''}(e)) \mid e \in E_{\gamma'''} \wedge h_1 \circ h_2^{-1}(e) \cap E_\gamma = \emptyset\}. \end{aligned}$$

Proof. We show that λ satisfies all the condition of Definition 5.3.12.

1. First of all note that $\gamma \xrightarrow{id} \gamma'$ and therefore $\mathcal{C}_\gamma \upharpoonright E_\gamma \cap E_{\gamma'} = \mathcal{C}_{\gamma'} \upharpoonright E_\gamma \cap E_{\gamma'}$. Let $e_1 \in E_\gamma \cap E_{\gamma'}$, we have $\mathcal{C}(e_1) = \mathcal{C}_{\gamma'}(e_1) = \mathcal{C}_{\gamma''}(h_1^{-1}(e_1))$ by definition of morphism. Every element of $h_1^{-1}(e_1)$ is then remapped by h_2 to the corresponding entities in $\lambda(e_1)$. Therefore

$$\begin{aligned} \mathcal{C}_\gamma(e_1) &= \mathcal{C}_{\gamma''}(h_1^{-1}(e_1)) \\ &= \sum_{e_2 \in E_{\gamma''}} \sum_{e_1 = h_1(e), e_2 = h_2(e)} \mathcal{C}_{\gamma''}(e) \\ &= \sum_{e_2 \in E_{\gamma''}} \lambda(e_1, e_2) \end{aligned}$$

For $e_1 \in E_{\gamma'} \setminus E_\gamma$ we can show the correspondence of the cardinality by the same argument. For $e_1 \in E_\gamma \setminus E_{\gamma'}$ the correspondence of the cardinality is ensured since λ is defined as *id* that is a reallocation.

2. Similar to 1.
3. Let $e \in E_\gamma$ and $A = \{e' \mid \lambda(e, e') = *\}$. By contradiction, assume $|A| > 1$, then there exist $e_1, e_2 \in A$ with $\mathcal{C}_{\gamma'''}(e_1) = \mathcal{C}_{\gamma'''}(e_2) = *$. $|A| > 1$ implies that e is split in more than one entity during the transition. However, *id* precedes the application of morphism therefore, the splitting of e (that can only happens by h_1) must happen *after* any change in the topology of γ performed by *id*. Hence $h_2 \circ h_1^{-1}(e)$ must be a chain containing at least e_1 and e_2 . In order to prove the statement it is enough to consider the case where $h_2 \circ h_1^{-1}(e)$ is only composed of e_1 and e_2 and we can either have $e_1 \prec_{\gamma'''} e_2$ or $e_2 \prec_{\gamma'''} e_1$. Without loss of generality, assume $e_1 \prec e_2$ (the other case is symmetric and can be proved in the same way). By hypothesis γ''' is compact, therefore by definition of L -safeness we have

$$\forall e_v \in PV : d(e_v, e_1) > L \wedge d(e_v, e_2) > L \quad (\text{B.9})$$

and by the hypothesis $e_1 \prec_{\gamma'''} e_2$ we have

$$d(e_v, e_2) > d(e_v, e_1) > L. \quad (\text{B.10})$$

Moreover, by definition of L -compactness we have

$$\text{indegree}(e_2) > 1 \vee \exists e_{v'} \in PV : d(e_{v'}, e_2) \leq L + 1. \quad (\text{B.11})$$

We distinguish the two cases:

- if $\exists e_{v'} \in PV : d(e_{v'}, e_2) \leq L + 1$ then by (B.10) we have $L < d(e_v, e_1) < d(e_v, e_2) \leq L + 1$, which is a contradiction.
- $\text{indegree}(e_2) > 1$ is also impossible since the rearrangement of links takes place before splitting, therefore $h_2 \circ h_1^{-1}(e)$ is a pure chain by definition of morphism, and $\text{indegree}(e_2) = 1$.

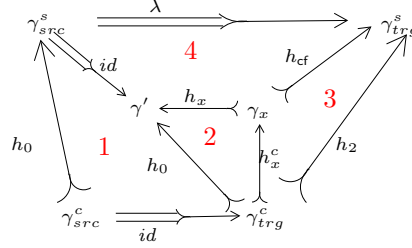


Figure B.3: Interdependence among morphisms and reallocations in the assignment rule.

The second part of this condition, i.e., $|\{e' \in E_{\gamma'''} \mid \lambda(\perp, e') = *\}| = 0$ follows in a straightforward manner from hypothesis (b).

4. If $\mathcal{C}_\gamma(e) = 1$, then e can be reallocated only on a single entity that is a chain by definition. Otherwise if $\mathcal{C}_\gamma(e) \neq 1$ then e can be reallocated in more than one entity. Nevertheless, e can be split in more than one entity only by the application of the inverse morphism h_1^{-1} , that is applied after the application of id . This means that when e is split, no other modifications in the link structure can happen. Therefore, since $h_1^{-1}(e)$ is a pure chain, it will be remapped by h_2 into another pure chain.
5. This condition follows in a straightforward manner from hypothesis (a).

□

Lemma B.2.1. Let γ_1 be a concrete configuration. Then:

$$\gamma_1 \xrightarrow{h_1} \gamma \Rightarrow \exists(\gamma_2, h_2) \in \text{SExp}(\gamma), \exists h_{12} : \gamma_1 \xrightarrow{h_{12}} \gamma_2.$$

Proof. We show the existence of $(\gamma_2, h_2) \in \text{SExp}(\gamma)$ and h_{12} by construction. First of all, if γ is L -safe then we trivially have $(\gamma, id_\gamma) \in \text{SExp}(\gamma)$ and $h_{12} = h_1$. Therefore, let us assume that γ is not L -safe and let

$$E_\gamma^{<L} = \{e \in E_\gamma \mid \mathcal{C}(e) \neq 1 \wedge \exists e' \in PV : d(e', e) \leq L\}$$

that is $E_\gamma^{<L}$ is the subset of non concrete entities that — being closer than $L + 1$ for some program variables — make γ non L -safe. Moreover, for $e \in E_\gamma$, with $h_1^{-1}(e) = e_1, \dots, e_n$, given a $m > 0$ let $\text{Prefix}_m(e) = \{e_1, \dots, e_m\}$, i.e., $\text{Prefix}_m(e)$ is the prefix of the pure chain $h_1^{-1}(e)$ containing the first m entities¹.

¹In case $m \geq n$ then $\text{Prefix}_m(e) = h_1^{-1}(e)$.

Now, let $\gamma_2 = (E_{\gamma_2}, \mu_{\gamma_2}, \mathcal{C}_{\gamma_2})$ defined as:

$$E_{\gamma_2} = E_{\gamma_1} \setminus \{e \in h_1^{-1}(e') \mid e' \in E_{\gamma_1}^{<L}\} \cup \bigcup_{e \in E_{\gamma_1}^{<L}} \text{Prefix}_{L+1}(e)$$

$$\forall e \in E_{\gamma_2} : \mu_{\gamma_2}(e) = \begin{cases} \mu_{\gamma_1}(\text{last}(h_1^{-1}(h_1(e)))) & \text{if } e = \text{last}(\text{Prefix}_{L+1}(e')) \wedge e' \in E_{\gamma_1}^{<L} \\ \mu_{\gamma_1}(e) & \text{otherwise} \end{cases}$$

$$\forall e \in E_{\gamma_2} : \mathcal{C}_{\gamma_2}(e) = \begin{cases} \lceil h_1^{-1}(e) - L \rceil & \text{if } e = \text{last}(\text{Prefix}_{L+1}(e')) \wedge e' \in E_{\gamma_1}^{<L} \\ \mathcal{C}_{\gamma_1}(e) & \text{otherwise.} \end{cases}$$

The previous definition has the following intuition: E_{γ_2} is a subset of entities in E_{γ_1} . Entities mapped by h_1 in $E_{\gamma_1}^{<L}$ are included only if they are among the first $L + 1$ entities of pure chain described by h_1 . μ_{γ_2} and \mathcal{C}_{γ_2} are defined according to E_{γ_2} so that γ_2 corresponds to an abstract version of E_{γ_1} .

We now define the morphism h_2 and h_{12} . In particular for $e \in E_{\gamma_2}$ let:

$$h_2(e) = h_1(e) \quad (\text{B.12})$$

and for $e' \in E_{\gamma_1}$ let $h_1^{-1}(h_1(e')) = e_1, \dots, e_n$ ($n > 0$):

$$h_{12}(e') = \begin{cases} e' & \text{if } e' = e_j \text{ and } 1 \leq j \leq L \\ e_{L+1} & \text{otherwise.} \end{cases} \quad (\text{B.13})$$

Note that according to this definition:

$$h_1 = h_2 \circ h_{12}. \quad (\text{B.14})$$

Since h_1 is a morphism (and by construction of γ_2) also h_2 is such. We prove that h_{12} is a morphism by showing that it fulfils condition **1m-4m** of Definition 5.3.3.

1m. Let $e \in E_{\gamma_2}$. Let $h_1^{-1}(h_2(e)) = e_1, \dots, e_n$. By definition of h_{12} , we have $e = e_j$ for some $1 \leq j \leq n$. We distinguish two cases:

1. if $j \leq L$ then $|h_{12}^{-1}(e)| = 1$ by construction, and therefore $h_{12}^{-1}(e)$ is a pure chain.
2. otherwise (by construction of h_{12}) it is $j = L + 1$, and $|h_{12}^{-1}(e)| = e_{L+1}, \dots, e_n$ which is also a pure chain.

2m. Let $e, e' \in E_{\gamma_2}$ such that $e \prec_{\gamma_2} e'$. Since h_2 is a morphism, then either $h_2(e) \prec_{\gamma} h_2(e')$ or $h_2(e) = h_2(e')$. In the first case, we have that (also because h_1 is a morphism):

$$\text{last}(h_{12}^{-1}(e)) = \text{last}(h_1^{-1}(h_2(e))) \preceq_{\gamma_1} \text{first}(h_1^{-1}(h_2(e'))) = \text{first}(h_{12}^{-1}(e'))$$

If $h_2(e) = h_2(e')$ the the property is fulfilled observing that by construction $h_1^{-1}(h_2(e)) = h_1^{-1}(h_2(e'))$. Let $h_1^{-1}(h_2(e)) = e_1, \dots, e_n$, then,

if e, e' are both within the prefix of the first L entities of the pure chain $h_1^{-1}(h_2(e))$ then $last(h_1^{-1}(e)) = e \prec_{\gamma_1} e' = first(h_1^{-1}(e'))$. Otherwise, whereas e must be at position L and e' at position $L + 1$. Thus, $last(h_1^{-1}(e)) = e_L \prec_{\gamma_1} first(h_1^{-1}(e')) = e_{L+1}$.

- 3m. Let $e, e' \in E_{\gamma_1}$ with $e \prec_{\gamma_1} e'$. Assume by contradiction that $h_{12}(e') \prec_{\gamma_2} h_{12}(e)$ then by definition of morphism $h_2(h_{12}(e')) \prec_{\gamma_2} h_2(h_{12}(e))$ that in turn implies, by (B.14), $h_1(e') \prec_{\gamma} h_1(e)$ that is a contradiction because h_1 is a morphism. Hence it must be $h_{12}(e) \preceq_{\gamma_2} h_{12}(e')$.
- 4m. For $e \in E_{\gamma_2}$, we prove that $C_{\gamma_2}(e) = C_{\gamma_1}(h_{12}^{-1}(e))$. According to the definition of γ_2 , we have that the cardinality of entities are the same as in γ_1 except for every entity that corresponds to $e = last(Prefix_{L+1}(e'))$ for some $e' \in E_{\gamma}^{<L}$. By definition h_{12} maps onto e the subchain of (concrete) entities in the suffix $E_{L+1}, \dots, e_{|h^{-1}(h_2(e))|}$ that corresponds to $C_{\gamma_2}(e)$.

Hence, h_{12} satisfies all the conditions 1m-4m, we conclude that it is a morphism. \square

The next lemma proves a property enjoyed by configurations related by a morphism h after the application of operations *add*, *cancel* or *modify* on both configurations using the same parameters. The property is described visually in Figure B.4: after the application of the corresponding operation the configuration are related by a morphism h' which, were defined, corresponds to h .

Lemma B.2.2. Let γ^s, γ^c be two $(0 <)L$ -safe configurations such that $\gamma^c \xrightarrow{h} \gamma^s$ and $\alpha = x.a^n$, and $\alpha' = y.a^m$ with $n, m \leq L$. Then:

1. $add(\gamma^c, \alpha) \xrightarrow{h'} add(\gamma^s, \alpha)$ where $h \upharpoonright (E_{\gamma^c} \cap E_{add(\gamma^c, \alpha)}) = h'$;
2. $cancel(\gamma^c, \alpha) \xrightarrow{h'} cancel(\gamma^s, \alpha)$ where $h \upharpoonright E_{cancel(\gamma^c, \alpha)} = h'$;
3. $modify(\gamma^c, \alpha, \alpha') \xrightarrow{h'} modify(\gamma^s, \alpha, \alpha')$ where $h \upharpoonright E_{modify(\gamma^c, \alpha, \alpha')} = h'$.

Proof.

1. By definition we have:

$$\begin{aligned} add(\gamma^c, \alpha) &= \langle E_{\gamma^c} \cup \{e\}, \mu_{\gamma^c}\{e/\llbracket \alpha \rrbracket\}, \mathcal{C}_{\gamma^c}\{1/e\} \rangle_{PV} \\ add(\gamma^s, \alpha) &= \langle E_{\gamma^s} \cup \{e'\}, \mu_{\gamma^s}\{e'/\llbracket \alpha \rrbracket\}, \mathcal{C}_{\gamma^s}\{1/e'\} \rangle_{PV} \end{aligned}$$

where $e = \min(Ent \setminus E_{\gamma^c})$ and $e' = \min(Ent \setminus E_{\gamma^s})$. We define the function $h' : E_{add(\gamma^c, \alpha)} \rightarrow E_{add(\gamma^s, \alpha)}$ as follows:

$$\forall \tilde{e} \in E_{add(\gamma^c, \alpha)} : h'(\tilde{e}) = \begin{cases} e' & \text{if } \tilde{e} = e \\ h(\tilde{e}) & \text{otherwise.} \end{cases}$$

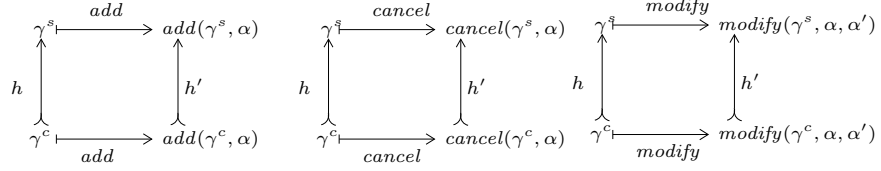


Figure B.4: Applying the same operation on configurations related by a morphism results in configurations related by a morphism.

A consequence of the interpretation ϑ for program variables and of the hypothesis on the morphisms (5.11) of Section 5.6.1 (see page 155) is that the update on γ^s (being L -safe) corresponds to that in γ^c , i.e., $h(\llbracket \alpha \rrbracket_{\mu_{\gamma^c}, \vartheta}) = h(\llbracket \alpha \rrbracket_{\mu_{\gamma^s}, \vartheta})$. In particular, if unreachable entities are obtained by the changes on the topology, they are a subset of $\mu_{\gamma^c}^*(\llbracket \alpha \rrbracket)$ and $\mu_{\gamma^s}^*(\llbracket \alpha \rrbracket)$, respectively. This means, that in γ^c and γ^s entities garbage collected by the application of $\langle \cdot \rangle_{PV}$ are related by the morphism h . Hence, since h is surjective, then it follows that h' is also surjective.

Moreover, because h is a morphism, it is possible to show that also h' satisfies condition 1m-4m of Definition 5.3.3 and thus h' is a morphism itself.

2. For the second case we have:

$$\begin{aligned} cancel(\gamma^c, \alpha) &= \langle E_{\gamma^c} \setminus \{\llbracket \alpha \rrbracket\}, \mu_{\gamma^c} \circ \psi^c, \mathcal{C}_{\gamma^c} \upharpoonright (E_{\gamma^c} \setminus \{\llbracket \alpha \rrbracket\}) \rangle_{PV} \\ cancel(\gamma^s, \alpha) &= \langle E_{\gamma^s} \setminus \{\llbracket \alpha \rrbracket\}, \mu_{\gamma^s} \circ \psi^s, \mathcal{C}_{\gamma^s} \upharpoonright (E_{\gamma^s} \setminus \{\llbracket \alpha \rrbracket\}) \rangle_{PV} \end{aligned}$$

where $\psi : E_{\gamma^c} \rightarrow E_{\gamma^c}^\perp$ and $\psi : E_{\gamma^s} \rightarrow E_{\gamma^s}^\perp$ are defined as

$$\begin{aligned} \psi^c(e) &= \begin{cases} \perp & \text{if } e \in \mu_{\gamma^c}^{-1}(\llbracket \alpha \rrbracket) \cup \llbracket \alpha \rrbracket \\ e & \text{otherwise} \end{cases} \\ \psi^s(e) &= \begin{cases} \perp & \text{if } e \in \mu_{\gamma^s}^{-1}(\llbracket \alpha \rrbracket) \cup \llbracket \alpha \rrbracket \\ e & \text{otherwise} \end{cases} \end{aligned}$$

In this case we define h' as restriction of h on the remaining entity after $\llbracket \alpha \rrbracket$ has been removed and garbage collection has taken place. That is

$$h' = h \upharpoonright E_{cancel(\gamma^c, \alpha)}.$$

By the same argument given for $add(\gamma^c, \alpha)$, it follows that h' is surjective and, by being a restriction of h , is a morphism.

3. Similar two the previous two cases. □

Lemma 5.6.25 λ defined in rule (ASGN-s) is a reallocation.

Proof. By Proposition 5.6.20, it is enough to prove that its preconditions (a) and (b) are satisfied by λ . For condition (a) we have to show that there does not exist an e in the target state such that $h \circ h_{\text{cf}}^{-1}(e)$ is not a chain in the source state. By contradiction let us assume such e does exist. Then, by definition of morphism, $h_{\text{cf}}^{-1}(e)$ is a pure chain in $\gamma_{q''} \in \text{SExp}(\gamma_{q'})$. This implies (in particular using 3m) that also $h \circ h_{\text{cf}}^{-1}(e)$ is a pure chain in $\gamma_{q'}$. Hence, since the chain does not exist in the source state, then it must have been created by the rearrangement of pointers due to the execution of the assignment. In the context of Proposition 5.6.20 this corresponds to $\gamma_q \xrightarrow{id} \gamma_{q'}$. Since the chain is created by manipulation of a link by the assignment there must exist two entities $e_1, e_2 \in h \circ h_{\text{cf}}^{-1}(e)$ such that $e_1 \not\prec_{\gamma_q} e_2$, before id and $e_1 \prec_{\gamma_{q'}} e_2$ after. However, γ_q is L -compact and therefore PV -reachable, then there exists $e_3 \neq e_1$ such that $e_3 \prec_{\gamma_q} e_2$ (e_3 could be either in PV or reachable from an entity in PV). But then in $\gamma_{q'}$ — after the link between e_1 and e_2 has been set — we have $\text{indegree}(e_2) > 1$, this implies that e_1, e_2 is not a pure chain contradicting our initial hypothesis. Therefore, precondition (a) is fulfilled.

The condition (b) is straightforward since in the assignment no new entities are created. \square

Theorem 5.6.27 For all program $p : id(\mathcal{A}_p) \sqsubseteq \mathcal{H}_p$.

Proof. We prove the theorem by defining a relation between $id(\mathcal{A}_p)$ and \mathcal{H}_p and proving that it is a simulation $id(\mathcal{A}_p) \sqsubseteq \mathcal{H}_p$. Let

$$\mathcal{R} = \{((r, \gamma^c), (r, \gamma^s), h) \in Q_{\mathcal{A}_p} \times Q_{\mathcal{H}_p} \times (Ent \rightarrow Ent) \mid r \in Par, \gamma^s = \text{cf}(\gamma^c), h = h_{\text{cf}}\}. \quad (\text{B.15})$$

First of all we prove that \mathcal{R} is a simulation, by showing that it satisfies conditions 1.a and 1.b of Definition 5.4.1. To this end, assume $((r, \gamma^c), (r, \gamma^s), h_0) \in \mathcal{R}$

- 1.a) Straightforward since by definition of \mathcal{R} we have $\gamma^c \xrightarrow{h_0} \gamma^s$.
- 1.b) We prove by induction on the structure of the statement r that the target state of every possible (r, γ^c) satisfies (i), (ii). We distinguish several cases.

Base of induction.

- **case** $r = \text{new}(\alpha.a)$. By the rules of the concrete transition system the only possible transition is:

$$\text{new}(\alpha.a), \gamma^c \rightarrow_{id} \text{skip}, \text{add}(\gamma^c, \alpha).$$

On the symbolic level we have:

$$\text{new}(\alpha.a), \gamma^s \rightarrow_{\lambda} \text{skip}, \text{cf}(\text{add}(\gamma^s, \alpha)).$$

with λ defined according to the operational rule. By Lemma B.2.2, $add(\gamma^c, \alpha) \succ^{h_1} add(\gamma^s, \alpha)$ where $h_1 \upharpoonright (E_{\gamma^c} \cap E_{add(\gamma^c, \alpha)}) = h_0$. Therefore if $h_2 : add(\gamma^s, \alpha) \succ \text{cf}(add(\gamma^s, \alpha))$ then it follows that

$$h_3 = h_2 \circ h_1 : add(\gamma^c, \alpha) \succ \text{cf}(add(\gamma^s, \alpha))$$

because of composition of morphisms. moreover, since $\text{cf}(add(\gamma^s, \alpha))$ is L -canonical by definition, then it is the canonical form of $add(\gamma^c, \alpha)$ via h_3 which is unique up to isomorphism (cf. Theorem 5.6.16). But then by definition of \mathcal{R} we have:

$$(\text{skip}, add(\gamma^c, \alpha)), (\text{skip}, \text{cf}(add(\gamma^s, \alpha))), h_3 \in \mathcal{R}$$

which proves (i).

For condition (ii) we must show that $id \triangleright \lambda$ via h_1 and h_3 . The first condition of Definition 5.3.17 has been already proved. For the second condition, let $e_1 \in E'$ then we have:

$$\begin{aligned} & \lambda(e_1, h_{cf}(e_1)) \\ = & \mathcal{C}(e_1) && \text{[by def } \lambda \text{ in (NEW-s)]} \\ = & [|h_0^{-1}(e_1)|] && \text{[by morphism def.]} \\ = & \sum_{\tilde{e} \in h_0^{-1}(e_1)} id(\tilde{e}, \tilde{e}) && \text{[by def. } id \text{ and } \mathcal{C}(\tilde{e}) = 1] \\ = & \sum_{(e_1, h_2(e_1)) = (h_0(\tilde{e}), h_3(\tilde{e}))} id(\tilde{e}, \tilde{e}) && [h_0(\tilde{e}) = e_1 \Rightarrow h_3(\tilde{e}) = h_2(e_1)] \end{aligned}$$

The property (No-Cross) is straightforward since the execution of the statement boils down to the application of $add(\gamma^c, \alpha)$ that does not perform rearrangement of links between entities (cf. (5.13) page 158) and because id does not produce any permutation of the entities. Also condition 4 of \triangleright is straightforward. In fact the only new entity in the concrete level is mapped by the morphism of the canonical form onto the new entity of the symbolic level (that is concrete). To see this, let $E_{add(\gamma^c, \alpha)} \setminus \text{cod}(id) = \{e\}$ and $(E_{add(\gamma^s, \alpha)} \setminus \text{cod}(\lambda) = \{e'\})$. By Lemma B.2.2 it must be $(h_2 \circ h_1)^{-1}(e') = e$ otherwise $h_1 \upharpoonright (E_{\gamma^c} \cap E_{add(\gamma^c, \alpha)}) = h_0$ would be contradicted (see also the proof of the lemma).

- **case** $r = \text{del}(\alpha.a)$. In this case, on the concrete level the only possible transition is:

$$\text{del}(\alpha.a), \gamma^c \xrightarrow{id} \text{skip}, \text{cancel}(\gamma^c, \alpha.a)$$

On the symbolic level:

$$\text{del}(\alpha.a), \gamma^s \xrightarrow{\lambda} \text{skip}, \text{cf}(\text{cancel}(\gamma^s, \alpha.a))$$

where λ is defined as the operational rule. Again by Lemma B.2.2 we have

$$\text{cancel}(\gamma^c, \alpha.a) \succ^{h_1} \text{cancel}(\gamma^s, \alpha.a)$$

and if $\text{cancel}(\gamma^s, \alpha.a) \xrightarrow{h_2} \text{cf}(\text{cancel}(\gamma^s, \alpha.a))$ then $\text{cf}(\text{cancel}(\gamma^s, \alpha.a))$ is also the canonical form of $\text{cancel}(\gamma^c, \alpha.a)$ by the morphism $h_3 = h_2 \circ h_1$. This implies by definition of \mathcal{R} :

$$((\text{skip}, \text{cancel}(\gamma^c, \alpha.a)), (\text{skip}, \text{cf}(\text{cancel}(\gamma^s, \alpha.a))), h_3) \in \mathcal{R}.$$

Moreover, λ defined in the `del` rule is the same as the one in the `new` rule, therefore, we have already shown in the case `new`($\alpha.a$) that conditions 1, 2, 3, of \triangleright definition hold. For condition 4 it is enough to observe that it is trivially true since the set of new entities is empty both at the concrete as well as at the symbolic level. Hence we can conclude that also for the case of `del` we have $id \triangleright \lambda$.

- **case** $r = \alpha_1.a := \alpha_2.a$. By the rules of the concrete transition system, the only possible transition is:

$$\alpha_1.a := \alpha_2.a, \gamma^c \xrightarrow{id} \text{skip}, \text{modify}(\gamma^c, \alpha_1.a, \alpha_2.a)$$

Let us indicate by γ_{trg}^c the target state (of the previous concrete transition). On the symbolic level we have:

$$\alpha_1.a := \alpha_2.a, \gamma^s \xrightarrow{id} \text{skip}, \text{cf}(\gamma_x)$$

for $(\gamma_x, h_x) \in \text{SExp}(\gamma')$ where $\gamma' = \text{modify}(\gamma^s, \alpha_1.a, \alpha_2.a)$ and $\lambda = h_{\text{cf}} \circ h_x^{-1}(\gamma^s \xrightarrow{id} \gamma')$. As above, let us refer to the target configuration of the previous symbolic transition by γ_{trg}^s .

Since this rule is nondeterministic, i.e., there can be more than one $\text{cf}(\gamma_x)$, we must show that there exists a morphism h_2 for *one* of the possible $\text{cf}(\gamma_x)$ resulting as target state of the rule.

Lemma B.2.2 allows us to conclude that there exists a morphism from γ_{trg}^c and γ' and this is precisely h_0 (cf. Figure B.3 left lower sub-diagram 1). Lemma B.2.1 then ensures the existence of a morphism h_x^c for one $\gamma_x \in \text{SExp}(\gamma')$. Thus, we can define $h_2 : \gamma_{trg}^c \xrightarrow{\quad} \text{cf}(\gamma_x)$ as composition of morphisms $h_2 = h_{\text{cf}}(\gamma_x) \circ h_x^c$ that by Proposition 5.3.7 is indeed a morphism (cf. Figure B.3, sub-diagram 3). But then since γ_{trg}^s is L -canonical, h_2 is the (unique) morphism that relate γ_{trg}^c to its normal form, that implies

$$((\text{skip}, \gamma_{trg}^c), (\text{skip}, \gamma_{trg}^s), h_2) \in \mathcal{R}.$$

Now, we show that $id \triangleright \lambda$. As for the previous statement the first condition of \triangleright is straightforward. In order to prove the the second, first of all observe the interdependence of morphisms and reallocation depicted in Figure B.3, and in particular, as we have already seen, by Lemma B.2.2, the morphism between γ_{trg}^c and γ' is in fact

h_0 . Hence, let $e_1 \in E_{\gamma'}$ and $e_2 \in E_{\gamma_{trg}^s}$, then we have:

$$\begin{aligned}
\lambda : (e_1, e_2) &= \sum_{\tilde{e} \in h_x^{-1}(e_1) \cap h_{cf(\gamma_x)}^{-1}(e_2)} \mathcal{C}(\tilde{e}) && [\text{def } \lambda \text{ in (ASGN-s)}] \\
&= |(h_x^c)^{-1}(h_x^{-1}(e_1) \cap h_{cf(\gamma_x)}^{-1}(e_2))| && [\text{by Figure B.3}] \\
&= |h_0^{-1}(e_1) \cap h_2^{-1}(e_2)| && [\text{by Figure B.3}] \\
&= \sum_{\tilde{e} \in h_0^{-1}(e_1) \cap h_2^{-1}(e_2)} id(\tilde{e}, \tilde{e}) \\
&= \sum_{(e_1, e_2) = (h_0(\tilde{e}), h_2(\tilde{e}))} id(\tilde{e}, \tilde{e}).
\end{aligned}$$

Finally, the reallocation id does not allow crossings, in particular because link updates occur only on entities with cardinality 1. Therefore the condition (No-Cross) is satisfied. Condition 4 of \triangleright is trivially fulfilled since no new entities are created in the concrete and symbolic transition. We conclude that $id \triangleright \lambda$.

Inductive step.

We show now only one case related to inductive step, namely the sequential composition, since the others can be shown in the same way. Assume a concrete state

$$q^c \equiv s_1; s_2, \gamma_1^c$$

and a symbolic

$$q^s \equiv s_1; s_2, \gamma_1^s.$$

According to the concrete transition system, state q^c can only perform a single transition

$$s_1; s_2, \gamma_1^c \rightarrow_{id} s'_1; s_2, \gamma_2^c$$

corresponding to a transition

$$s_1, \gamma_1^c \rightarrow_{id} s'_1, \gamma_2^c$$

for some statement s'_1 completely determined by the structure of s_1 . On the symbolic level to the transition performed by q^s corresponds to:

$$s_1; s_2, \gamma_1^s \rightarrow_{\lambda} s'_1; s_2, \gamma_2^s$$

for some λ determined by the transition

$$s_1, \gamma_1^s \rightarrow_{\lambda} s'_1, \gamma_2^s.$$

By induction hypothesis we have $((s_1, \gamma_1^c), (s_1, \gamma_1^s), h_1) \in \mathcal{R}$. Therefore:

- (a) there exists h_2 such that $((s'_1, \gamma_2^c), (s'_1, \gamma_2^s), h_2) \in \mathcal{R}$ and
 (b) $id \triangleright \lambda$.

But then by (a), (b) we have that:

- * $h_1 : q^c \xrightarrow{\triangleright} q^s$
- * $((s'_1; s_2, \gamma_2^c), (s'_1; s_2, \gamma_2^s), h_2) \in \mathcal{R}$
- * $id \triangleright \lambda$.

Then by definition, it is $(q^c, q^s, h_1) \in \mathcal{R}$.

We conclude that the relation \mathcal{R} is a simulation.

Now in order to prove that $\mathcal{A}_p \sqsubseteq \mathcal{H}_p$ we show that condition 2 of Definition 5.4.1 is satisfied by the simulation \mathcal{R} .

- 2.a) Both automata $id(\mathcal{A}_p)$ and \mathcal{H}_p have only one single initial state with the same configuration, namely $(PV, \emptyset, \mathbf{1}_{PV})$ that is also L -canonical. But the we have:

$$((r, (PV, \emptyset, \mathbf{1}_{PV})), (r, (PV, \emptyset, \mathbf{1}_{PV})), id_{(PV, \emptyset, \mathbf{1}_{PV})}) \text{ in } \mathcal{R}.$$

where r is the initial statement of the program p .

The second condition 2.a) that relates the set of new entities N and the initial valuation θ of the initial states is trivial since — as we have seen — both $id(\mathcal{A}_p)$ and \mathcal{H}_p have the same single initial state.

- 2.b) According to the definition of accept state for $id(\mathcal{A}_p)$ and \mathcal{H}_p have the following sets:

$$\begin{aligned} \mathcal{F}_{id(\mathcal{A}_p)} &= \{\hat{F}_1^c, \dots, \hat{F}_k^c, \tilde{F}_1^c, \dots, \tilde{F}_k^c\} \\ \mathcal{F}_{\mathcal{H}_p} &= \{\hat{F}_1^s, \dots, \hat{F}_k^s, \tilde{F}_1^s, \dots, \tilde{F}_k^s\} \end{aligned}$$

therefore as $\psi : \mathcal{F}_{id(\mathcal{A}_p)} \rightarrow \mathcal{F}_{\mathcal{H}_p}$ we naturally take:

$$\begin{aligned} \psi(\hat{F}_i^c) &= \hat{F}_i^s \\ \psi(\tilde{F}_i^c) &= \tilde{F}_i^s \end{aligned}$$

for $1 \leq i \leq k$. It is straightforward to see that ψ is bijective. Now, let $F \in \mathcal{F}_{id(\mathcal{A}_p)}$ and $q = (s, \gamma) \in F$, we must prove that there exists $q' \in \psi(F)$ and a morphism $h : q \xrightarrow{\triangleright} q'$ such that $(q, q', h) \in \mathcal{R}$. Let $q' = (s, \text{cf}(\gamma))$, the existence of $\text{cf}(\gamma)$ is ensured by Theorem 5.6.16 since q — being concrete — is L -safe by definition as well as PV -reachable by Proposition 5.6.10. Having a L -canonical and well-formed configuration implies that $q' \in Q_{\mathcal{H}_p}$. Moreover, $q' \in \psi(F)$ since q and q' have the same statement s . But then by definition of \mathcal{R} we have $(q, q', h) \in \mathcal{R}$.

Since condition 2.a) and 2.b) of Definition 5.4.1 hold we finally conclude that $id(\mathcal{A}_p) \sqsubseteq \mathcal{H}_p$. \square

Lemma B.2.3. Let $E_{\max} \geq 0$ be a fixed constant. If for all $q \in Q_{\mathcal{H}_p} : |E_q| \leq E_{\max}$ then $|Q_{\mathcal{H}_p}|$ is finite.

Proof. \mathcal{H}_p states are of the form $q = (s, E, \mu, \mathcal{C})$. Let s_{\max} be the longest sequential component in s and let m be the number of sequential components. The number of possibilities for s is bound by $|s_{\max}|^m$. Because of the use of reallocations we can consider states with entities up to renaming, i.e., we can always use entities within a finite set with E_{\max} elements (by hypothesis $|E| \leq E_{\max}$)². Furthermore, the component μ ranges over the set of all possible applications from E onto E^\perp . Therefore, fixing E we have $(|E|+1)^{|E|}$ different possible μ . Finally the component \mathcal{C} is the set of all mappings from E to the set $\{1, \dots, M\} \cup \{*\}$ therefore, we have $(M+1)^{|E_{\max}|}$ possibilities. Summarising we have:

$$\begin{aligned} |Q_{\mathcal{H}_p}| &\leq |s_{\max}|^m \cdot 2^{E_{\max}} \cdot \sum_{n=0}^{E_{\max}} (n+1)^n \cdot \sum_{n=0}^{E_{\max}} (M+1)^n \\ &\leq |s_{\max}|^m \cdot 2^{E_{\max}} \cdot \frac{(E_{\max}+1)^{E_{\max}} - 1}{E_{\max}} \cdot \frac{(M+1)^{E_{\max}} - 1}{M} \end{aligned}$$

Hence, $Q_{\mathcal{H}_p}$ is finite. \square

Theorem 5.6.29 For all programs p , \mathcal{H}_p is finite-state.

Proof. By contradiction, assume \mathcal{H}_p is infinite-state. Therefore, by Lemma B.2.3 there does not exist a bound on the number of entities in the states, i.e., there does not exist a constant, say $E_{\max} \geq 0$ such that for all $q \in Q_{\mathcal{H}_p} : |E_q| \leq E_{\max}$. Since every state has a canonical and therefore PV -reachable (cf. Proposition 5.6.14) configurations, this can be the case only for two reasons:

- either the number of PV is infinite,
- or in the state there are unbounded chains of entities.

The first possibility is impossible because, by definition, PV is finite. The second is also impossible. In fact, taking an unbounded chain, the number of entities with indegree greater than one is either unbounded or bounded. On the one hand, if we assume that the number of such entities is unbounded, then since the state is PV -reachable this again would imply that PV is infinite. On the other hand, if there is a bounded number of entities with indegree greater than one, then since the chain is unbound it must be that after a certain point we have an unbounded number of entities with indegree equal one, i.e., the chain becomes pure. Again this is not possible since states are in canonical form. \square

²As we have seen, this of course does not prevent to express unbounded number of entity creations.

B.3 Proofs of Section 5.7

Theorem 5.7.5 For all HABA \mathcal{H} such that $\mathcal{C}(\mathcal{H}) = M < \hat{M}$: $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{H} \uparrow \hat{M})$.

Proof. [$\mathcal{L}(\mathcal{H}) \subseteq \mathcal{L}(\mathcal{H} \uparrow \hat{M})$] Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$, with $\sigma = E_0 \lambda_0 E_1 \lambda_1 \dots$. Let $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ be the run generating (σ, N, θ) by some generator $(h_i)_{i \in \mathbb{N}}$. We define a run ρ' of $\mathcal{H} \uparrow \hat{M}$ that generates (σ, N, θ) in the following way. Let $\rho' = q'_0 \lambda_0 q'_1 \lambda'_1 \dots$ where for all $i \geq 0$, we take $q'_i \in S_{q_i}$ such that:

$$\forall e \in E_{q'_i} : [|h_i^{-1}(e)|]_{\hat{M}} = \mathcal{C}_{q'_i}(e).$$

By construction of the states in ρ' and by the definition of $\mathcal{H} \uparrow \hat{M}$ for all $i \geq 0$ there exists a transition $q'_i \xrightarrow{\lambda_i} q'_{i+1}$. As q'_0 is an initial state of $\mathcal{H} \uparrow \hat{M}$ and for every accept state q_i visited infinitely often, also the corresponding accept state q'_i is visited infinitely often, we conclude that $\rho' \in \text{runs}(\mathcal{H} \uparrow \hat{M})$. Finally, it is possible to show that generator $(h_i)_{i \in \mathbb{N}}$ generates (σ, N, θ) also from run ρ' . Hence we conclude that $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H} \uparrow \hat{M})$.

[$\mathcal{L}(\mathcal{H} \uparrow \hat{M}) \subseteq \mathcal{L}(\mathcal{H})$] Let $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H} \uparrow \hat{M})$, with $\sigma = E_0 \lambda_0 E_1 \lambda_1 \dots$, and let $\rho = q'_0 \lambda_0 q'_1 \lambda_1 \dots$ be the run generating (σ, N, θ) . We define $\rho = q_0 \lambda_0 q_1 \lambda_1 \dots$ such that for all $i > 0$, where $q'_i \in S_{q_i}$. Since $\rho' \in \text{runs}(\mathcal{H} \uparrow \hat{M})$ implies $\rho \in \text{runs}(\mathcal{H})$ and ρ' generates (σ, N, θ) by a generator $(h_i)_{i \in \mathbb{N}}$ implies ρ generates (σ, N, θ) by the same $(h_i)_{i \in \mathbb{N}}$, we conclude that $(\sigma, N, \theta) \in \mathcal{L}(\mathcal{H})$. \square

Proposition 5.7.7 For all γ -valuation (ψ, Θ, δ) :

- (a) $(\exists j > 0 : \mu^j \circ \Theta(x) = \Theta(y) \neq \perp) \Rightarrow (x, y) \in \text{dom}(\delta)$
- (b) $\Theta(x) = \Theta(y) \neq \perp \Rightarrow (x, y) \in \text{dom}(\delta) \vee (y, x) \in \text{dom}(\delta)$

Proof.

- (a) If there exists $j > 0$ such that $\mu^j \circ \Theta(x) = \Theta(y)$ then by (DELTA GAMMA 2) it can be proved that for all $0 < i \leq j$ we have $\delta(\mu^{i-1} \circ \Theta(x)^+, \mu^i \circ \Theta(x)^-) = 1$ that in turn implies $(\mu^{i-1} \circ \Theta(x)^+, \mu^i \circ \Theta(x)^-) \in \text{dom}(\delta)$. Similarly by (DELTA GAMMA 1), it follows that for all $0 < i \leq j$: $(\mu^i \circ \Theta(x)^-, \mu^i \circ \Theta(x)^+) \in \text{dom}(\delta)$. Combining these facts, we have $(\Theta(x)^-, \mu^j \circ \Theta(x)^+) = (\Theta(x)^-, \Theta(y)^+) \in \text{dom}(\delta)$ and moreover, $j > 0$ implies $(\Theta(x)^+, y) \in \text{dom}(\delta)$. By (DELTA THETA 2) $(x, \Theta(x)^+) \in \text{dom}(\delta)$. Therefore, we can conclude using (DELTA MET 2) that $(x, y) \in \text{dom}(\delta)$.

- (b) Since $\Theta(x) = \Theta(y) \neq \perp$ we have by (DELTA THETA 2):

$$\begin{aligned} \delta(\Theta(x)^-, x) \geq 0 \wedge \delta(\Theta(x)^-, y) \geq 0 \\ \delta(x, \Theta(x)^+) \geq 0 \wedge \delta(y, \Theta(x)^+) \geq 0 \end{aligned}$$

Therefore by (DELTA MET 3) we have either $\delta(\Theta(x)^-, x) \oplus \delta(x, y) = \delta(\Theta(x)^-, y)$ or $\delta(\Theta(x)^-, y) \oplus \delta(y, x) = \delta(\Theta(x)^-, x)$. Hence we have either $(x, y) \in \text{dom}(\delta)$ (corresponding to the first case) or $(y, x) \in \text{dom}(\delta)$ (corresponding to the second case). \square

Proposition 5.7.19 For $q_1 \xrightarrow{\lambda} q_2$, if E is a maximal weakly connected subgraph of $(E_{q_1} \uplus E_{q_2}, \lambda)$ such that $\perp \notin E$ then $\mathcal{C}_{q_1}(\iota_1(E)) = \mathcal{C}_{q_2}(\iota_2(E))$.

Proof. Without loss of generality let

$$\iota_1(E) = \{e_1, \dots, e_n\} \quad \text{and} \quad \iota_2(E) = \{e'_1, \dots, e'_k\}$$

for $k, n \geq 1$. Since E is a maximal weakly connected subgraph, we have the property:

- $\forall e \in \iota_1(E) : \lambda(e) \in \iota_2(E)$
- $\forall e \in \iota_2(E) : \lambda^{-1}(e) \in \iota_1(E)$.

Thus we can construct a $n \times k$ matrix R where $R(r, c) = \lambda(e_r, e_c)$ (with $1 \leq r \leq n$ and $1 \leq c \leq k$). Hence we have:

$$\begin{aligned} & \mathcal{C}_1(\iota_1(E)) \\ &= \mathcal{C}_1(e_1) + \dots + \mathcal{C}_1(e_n) && \text{[by definition]} \\ &= \sum_{j=1}^k R(1, j) + \dots + \sum_{j=1}^k R(n, j) && \text{[by def. of } R \text{ and Def 5.3.12 con. 1]} \\ &= \sum_{j=1}^n R(j, 1) + \dots + \sum_{j=1}^n R(j, k) && \text{[by def. of } R \text{ and Def 5.3.12 con. 2]} \\ &= \mathcal{C}_2(e'_1) + \dots + \mathcal{C}_2(e'_k) \\ &= \mathcal{C}_2(\iota_2(E)). \end{aligned}$$

\square

We will now prove few lemmas that that will be used in the proof of Proposition 5.7.25. These proofs use the following notion. Let γ' and γ be a concrete and an abstract configuration, respectively. Let $\theta : fv(\psi) \rightarrow E_{\gamma'}$ be an interpretation for the logical variables over γ' . We extend θ to the set of special logical variables E_{γ}^{\pm} as follows. Let $h : \gamma' \succ \rightarrow \gamma$ and $e \in E_{\gamma}$, then:

$$\theta^{\pm}(x) = \begin{cases} \theta(x) & \text{if } x \in fv(\psi) \\ \text{first}(h^{-1}(e)) & \text{if } x = e^- \\ \text{last}(h^{-1}(e)) & \text{if } x = e^+ \end{cases} \quad (\text{B.16})$$

The next definition provides us with a consistency notion between a concrete interpretation for logical variables θ (in γ') w.r.t. a symbolic interpretation given by Θ, δ in a γ -valuation when $\gamma' \xrightarrow{h} \gamma$.

Definition B.3.1. Let $h : \gamma' \succ \rightarrow \gamma$ be a morphism where $\mathcal{C}_{\gamma'} = \mathbf{1}$, (ψ, Θ, δ) a γ -valuation and $\theta : \text{LVAR} \rightarrow E_{\gamma'}$. Then, (θ, h) is consistent with (Θ, δ) (written $(\theta, h) \approx (\Theta, \delta)$) if

- (a) $h \circ \theta = \Theta$
- (b) $\forall x, y \in fv(\psi) \cup E_{\gamma}^{\pm} : \delta(x, y) = \min \{ \lceil n \rceil \mid \mu_{\gamma'}^n \circ \theta^{\pm}(x) = \theta^{\pm}(y) \}$ where $\min \emptyset = \perp$.

We use the previous definition in order to keep consistency, from state to state, between the interpretation given by the valuations of a path π and the interpretation of an allocation sequence σ generated by π .

Lemma B.3.2. Let $\pi = (q_0, D_0)\lambda_0(q_1, D_1)\lambda_1 \cdots$ be a path and let σ be an allocation sequence generated by the underlying run of π with generator $(h_j)_{j \in \mathbb{N}}$ and $\psi \in CL(\phi)$. Then for all $i \geq 0$

$$(\theta, h_i) \simeq (h_i \circ \theta, \delta) \Rightarrow (\psi, h_{i+1} \circ \lambda_i^{\sigma} \circ \theta, \delta') \in [\lambda_i \circ (\psi, h_i \circ \theta, \delta)].$$

for any δ' such that $(\lambda_i^{\sigma} \circ \theta, h_{i+1}) \simeq (h_{i+1} \circ \lambda_i^{\sigma} \circ \theta, \delta')$.

Proof. First of all, we recall that a δ' with the property $(\lambda_i^{\sigma} \circ \theta, h_{i+1}) \simeq (h_{i+1} \circ \lambda_i^{\sigma} \circ \theta, \delta')$ is defined (according to Definition B.3.1) as

$$\delta'(x, y) = \min \{ \lceil n \rceil \mid \mu^n \circ (\lambda_i^{\sigma} \circ \theta)^{\pm}(x) = (\lambda_i^{\sigma} \circ \theta)^{\pm}(y) \}$$

for $x, y \in fv(\psi) \cup E_{q_{i+1}}^{\pm}$. Now, for brevity, let $\Theta = h_i \circ \theta$ and $\Theta' = h_{i+1} \circ \lambda_i^{\sigma} \circ \theta$. By definition of generator (Def. 5.3.24) we have that $\lambda_i^{\sigma} \triangleright \lambda_i$ and therefore

$$\lambda_i(\Theta(x), \Theta'(x)) \neq 0. \quad (\text{B.17})$$

Hence, condition 1 of Definition 5.7.20 is fulfilled. We now prove condition 2.

Let E be a maximal (weakly) connected subgraph of $(E_{q_i} \uplus E_{q_{i+1}}, \lambda_i)$. Proposition 5.7.19 and the property (No-Cross) of \triangleright (Def. 5.3.17) imply:

$$|h_i^{-1}(\iota_1(E))| = |h_{i+1}^{-1}(\iota_2(E))|. \quad (\text{B.18})$$

That is: the number of entities on the concrete level that correspond to the first and the second projection of the maximal connected subgraph is the same³. For brevity, let $C = |h_i^{-1}(\iota_1(E))|$ and let

$$\begin{aligned} h_i^{-1}(\iota_1(E)) &= e_1^i, \quad \dots, \quad e_C^i \\ h_{i+1}^{-1}(\iota_2(E)) &= e_1^{i+1}, \quad \dots, \quad e_C^{i+1}. \end{aligned}$$

Another consequence of (No-Cross) is that for all $1 \leq m \leq C$:

$$\lambda_i^{\sigma}(e_m^i, e_m^{i+1}) \neq 0 \quad (\text{B.19})$$

³Note that (No-Cross) is necessary since if the cardinalities on the abstract level correspond, in case of unbounded entities, this does not necessarily imply that also at the concrete level the corresponding cardinalities are the same. In fact, some entity may during the transition but this may not be revealed on the abstract level if the cardinality is $*$.

that is, the entity at position m in the chain corresponding to $\iota_1(E)$ is re-allocated in the entity at position m in the chain corresponding to $\iota_2(E)$. Therefore, it is straightforward to see that the interpretation of a variable x on $\iota_1(E)$ will have the same position in $\iota_2(E)$, in particular, the distance between $first(\iota_2(E))^-$ and $last(\iota_2(E))^+$ is unchanged w.r.t. $first(\iota_1(E))^-$ and $last(\iota_1(E))^+$. Hence, conditions 2(a)-(b) of Definition 5.7.20 hold.

For the other free variables y with interpretation both on $\iota_1(E)$ and $\iota_2(E)$ we have:

$$\mu_i^n(\theta(x)) = \theta(y) \Rightarrow \mu_{i+1}^n(\lambda_i^\sigma \circ \theta(x)) = \lambda_i^\sigma \circ \theta(y) \Rightarrow \delta'(x, y) = [n] \quad (\text{B.20})$$

Moreover, since $(\theta, h_i) \approx (h_i \circ \theta, \delta)$ we have:

$$\mu_i^n(\theta(x)) = \theta(y) \Rightarrow \delta(x, y) = [n]. \quad (\text{B.21})$$

Thus, from (B.20) and (B.21) it follows that δ' satisfies condition 2(c) of Definition 5.7.20. We conclude that $(\psi, h_{i+1} \circ \lambda_i^\sigma \circ \theta, \delta') \in [\lambda_i \circ (\psi, h_i \circ \theta, \delta)]$. \square

Lemma B.3.3. Let $\pi = (q_0, D_0)\lambda_0(q_1, D_1)\lambda_1 \cdots$ be a path and let σ be an allocation sequence generated by the underlying run of π with generator $(h_j)_{j \in \mathbb{N}}$. Then for all $j > i \geq 0$,

a) if $(\lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, h_j) \approx (h_j \circ \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta)$ then

$$(\psi, h_{j+1} \circ \lambda_j^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta') \in [\lambda_j \circ (h_j \circ \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta)]$$

for any δ' such that $(\lambda_j^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, h_{j+1}) \approx (h_{j+1} \circ \lambda_j^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta')$.

b) if $(\theta, h_i) \approx (h_i \circ \theta, \delta)$ then

$$(\psi, h_j \circ \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta') \in [\lambda_{j-1} \circ \cdots \circ \lambda_i \circ (\psi, h_i \circ \theta, \delta)]$$

for any δ' such that $(\lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, h_j) \approx (h_j \circ \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta')$.

Proof.

a) Straightforward. In fact, let $\theta' = \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta$. Then by Lemma B.3.2 we have

$$(\psi, h_{j+1} \circ \lambda_j^\sigma \circ \theta', \delta') \in [\lambda_j \circ (\psi, h_j \circ \theta', \delta)]$$

and therefore: $(\psi, h_{j+1} \circ \lambda_j^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta') \in [\lambda_j \circ (h_j \circ \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma \circ \theta, \delta)]$ that in fact is what we wanted to prove.

b) We prove this part of the lemma by induction on $j \geq i$. The base step corresponds to Lemma B.3.2 therefore we show here the inductive step. For the sake of readability, let $\overline{\lambda_{j-1,i}} = \lambda_{j-1} \circ \cdots \circ \lambda_i$ and $\overline{\lambda_{j-1,i}^\sigma} = \lambda_{j-1}^\sigma \circ \cdots \circ \lambda_i^\sigma$. First of all we prove that

$$[\lambda_j \circ (\psi, h_j \circ \overline{\lambda_{j-1,i}^\sigma} \circ \theta, \delta)] \subseteq [\lambda_j \circ \overline{\lambda_{j-1,i}} \circ (\psi, h_i \circ \theta, \delta)]. \quad (\text{B.22})$$

The set inclusion (B.22) can be proved using the induction hypothesis:

$$(\psi, h_j \circ \overline{\lambda_{j-1,i}^\sigma} \circ \theta, \delta') \in [\lambda_{j-1} \circ \overline{\lambda_{j-2,i}} \circ (\psi, h_i \circ \theta, \delta)]$$

as follows:

$$\begin{aligned} & [\lambda_j \circ \overline{\lambda_{j-1,i}} \circ (\psi, h_i \circ \theta, \delta)] \\ &= \bigcup_{v \in [\overline{\lambda_{j-1,i}} \circ (\psi, h_i \circ \theta, \delta)]} \lambda_j \circ v \\ &= \left(\bigcup_{v \in [\overline{\lambda_{j-1,i}} \circ (\psi, h_i \circ \theta, \delta)] \setminus \{(\psi, h_j \circ \overline{\lambda_{j-1,i}^\sigma} \circ \theta, \delta)\}} \lambda_j \circ v \right) \\ & \quad \cup [\lambda_j \circ (\psi, h_j \circ \overline{\lambda_{j-1,i}^\sigma} \circ \theta, \delta)] \end{aligned}$$

Hence combining (B.22) and part a) of this lemma we conclude $(\psi, h_{j+1} \circ \lambda_j^\sigma \circ \dots \circ \lambda_i^\sigma \circ \theta, \delta) \in [\lambda_j \circ \dots \circ \lambda_i \circ (\psi, h_i \circ \theta, \delta)]$. \square

Lemma B.3.4. Let $\gamma = (E, \mu, \mathbf{1})$ and γ' be two configurations such that $\gamma \xrightarrow{h} \gamma'$ and $v = (\psi, h \circ \theta, \delta)$ a γ' -valuation. Then

$$(h, \theta) \simeq (h \circ \theta, \delta) \Rightarrow h \circ \mu^n(\theta(x)) = \llbracket x.a^n \rrbracket_{\gamma,v}^1$$

Proof. The intuition of this lemma is that the entity on the concrete level representing $x.a^n$ is mapped precisely on the entity on the abstract level that corresponds to the (first components of the) semantics of $x.a^n$.

We prove this lemma by induction on n .

- Base case $n = 0$. If $\theta(x) = \perp$ then $h \circ \theta(x) = \perp$ (is undefined) and by definition $\llbracket x \rrbracket_{\gamma,v}^1 = \perp$. If $\theta(x) \neq \perp$ then we have $h \circ \mu^0(\theta(x)) = h \circ \theta(x) = \llbracket x.a^n \rrbracket_{\gamma,v}^1$ by Definition 5.7.13.
- Inductive step. Assume the statement holds for n . We prove it for $n+1$. If $\theta(x) = \perp$ then $\mu^n(\theta(x)) = \mu^{n+1}(\theta(x)) = \perp$, that implies $h \circ \mu^n(\theta(x))$ and $h \circ \mu^{n+1}(\theta(x))$ are undefined. Because of the hypothesis of consistency, we have immediately that also $h \circ \theta(x) = \perp$ therefore by definition $\llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket x.a^{n+1} \rrbracket_{\gamma,v}^1 = \perp$. Therefore, assume $\theta(x) \neq \perp$ and let $h^{-1}(h \circ \theta(x)) = e_1, \dots, e_k$ and $\theta(x) = e_i$. We distinguish two cases.
 1. If $i + n + 1 \leq k$ then $h \circ \mu^n(\theta(x)) = h \circ \mu^{n+1}(\theta(x)) = h \circ \theta(x)$. Since $(h, \theta) \simeq (h \circ \theta, \delta)$, then $\delta(x, \Theta(x)^+) = \lceil k - i \rceil \geq n$ therefore by Definition 5.7.13 we have $\llbracket x.a^n \rrbracket_{\gamma,v}^1 = h \circ \theta(x)$.
 2. If $i + n + 1 > k$ again we distinguish two cases:
 - (a) Assume $h \circ \mu^n(\theta(x)) = h \circ \mu^{n+1}(\theta(x))$, that is $\mu^n(\theta(x))$ and $\mu^{n+1}(\theta(x))$ are mapped by the morphism h onto the same abstract entity and let $\Theta = h_i \circ \theta$. By the induction hypothesis we

have that $h \circ \mu^n(\theta(x)) = \llbracket x.a^n \rrbracket_{\gamma',v}^1$. Moreover, if $\llbracket x.a^n \rrbracket_{\gamma',v}^1 = \mu_{\gamma'}^j \circ \Theta(x)$ then

$$\mathcal{C}_v(x, j) > n$$

where the inequality is strict since $x.a^n$ is not the last instance of $\llbracket x.a^n \rrbracket_{\gamma',v}^1$ otherwise $h \circ \mu^n(\theta(x))$ and $h \circ \mu^{n+1}(\theta(x))$ would not be the same contradicting the initial hypothesis. But then $\mathcal{C}_v(x, j) \geq n + 1$ and by Definition 5.7.13 we have $\llbracket x.a^n \rrbracket_{\gamma',v}^1 = \llbracket x.a^{n+1} \rrbracket_{\gamma',v}^1$. Summarising in one line:

$$h \circ \mu^{n+1}(\theta(x)) = h \circ \mu^n(\theta(x)) = \llbracket x.a^n \rrbracket_{\gamma',v}^1 = \llbracket x.a^{n+1} \rrbracket_{\gamma',v}^1$$

that is precisely what we wanted to prove.

(b) As a second possibility, assume

$$h \circ \mu^{n+1}(\theta(x)) = \mu_{\gamma'}(h \circ \mu^n(\theta(x))) \quad (\text{B.23})$$

that is $\mu^n(\theta(x))$ and $\mu^{n+1}(\theta(x))$ are mapped by the morphism h onto consecutive abstract entities. This initial hypothesis implies that $\mu^n(\theta(x))$ denotes precisely the *last* entity in the chain of concrete entities corresponding to $h^{-1}(h \circ \mu^n(\theta(x)))$. If this would not be the case then also $\mu^{n+1}(\theta(x))$ would be mapped by h on the same entity (since h is a morphism) therefore contradicting the initial hypothesis. Let $\Theta = h_i \circ \theta$ and assume that

$$\llbracket x.a^n \rrbracket_{\gamma',\Theta,\delta}^1 = \mu_{\gamma'}^{j-1} \circ \Theta(x) \quad (\text{B.24})$$

for some $j > 0$. Then, by Definition 5.7.13 we have:

$$n \leq \mathcal{C}_v(x, j-1) < n+1 \quad (\text{B.25})$$

where the last inequality is true otherwise $\llbracket x.a^{n+1} \rrbracket_{\gamma',v}^1 = \mu_{\gamma'}^{j-1} \circ \Theta(x)$ that would imply that $h \circ \mu^n \circ \theta(x)$ is not the last entity of $h^{-1}(h \circ \mu^n(\theta(x)))$.

Now, by (B.25) it is straightforward to see that $\mathcal{C}_v(x, j-1) = n$ and therefore $\mathcal{C}_v(x, j) \geq n+1$. By Definition 5.7.13 this implies $\llbracket x.a^{n+1} \rrbracket_{\gamma',v}^1 = \mu_{\gamma'}^j \circ \Theta(x)$. Note that the entity $\mu_{\gamma'}^j \circ \Theta(x)$ does exist since it is the image (via h) of $\mu^{n+1} \circ \theta(x)$. Finally, we have:

$$\begin{aligned} \llbracket x.a^{n+1} \rrbracket_{\gamma',v}^1 &= \mu_{\gamma'}^j \circ \Theta(x) && \text{[by definition]} \\ &= \mu_{\gamma'} \circ \mu_{\gamma'}^{j-1} \circ \Theta(x) && \text{[by (B.24)]} \\ &= \mu_{\gamma'}(\llbracket x.a^n \rrbracket_{\gamma',v}^1) && \text{[by induction]} \\ &= \mu_{\gamma'}(h \circ \mu^n \circ \theta(x)) && \text{[by (B.23)]} \\ &= h \circ \mu^{n+1} \circ \theta(x). \end{aligned}$$

□

Lemma B.3.5. Let $\rho = q_0\lambda_0q_1\lambda_1\cdots$ be a run generating a triple (σ, N, θ) with generator $(h_i)_{i \in \mathbb{N}}$ such that $\sigma, N, \theta \models \phi$. Let

$$\begin{aligned} D_i &= \{(\psi, h_i \circ \theta, \delta_i) \mid \psi \in CL(\phi), \sigma^i, N_i^\sigma, \theta \models \psi\} \\ \forall x, y \in fv(\psi) \cup E_{q_i}^\pm : \delta_i(x, y) &= \min \{[n] \mid \mu_i^n \circ \theta^\pm(x) = \theta^\pm(y)\} \end{aligned}$$

where θ^\pm is the extension of θ to the set $E_{q_i}^\pm$ as defined by (B.16) (cf. page 283). Then (q_i, D_i) is an atom for all $i \in \mathbb{N}$.

Proof. We show by induction on the structure of ψ that D_i satisfies the conditions of Definition 5.7.17.

- **Atomic propositions.** Give a valuation $v = (\psi, \Theta, \delta)$, where ψ is an atomic propositions, we have to prove the following implication:

$$v \in AV_{q_i} \Rightarrow v \in D_i. \quad (\text{B.26})$$

To this end we need to prove the existence of a suitable (concrete) interpretation θ for $fv(\psi)$ and extend it according to (B.16) (cf. page 283). Suitable in this context means that the choice of $\theta(x)$ should be such that ψ is valid and — at the same time — $\Theta = h_i \circ \theta$ and $\delta = \delta_i$. We now single out the three possible cases occurring for the atomic proposition.

- case $\psi = x.a^n \text{ new}$. Let $v = (x.a^n \text{ new}, \Theta, \delta)$ and let $h_i^{-1}(\Theta(x)) = e_1, \dots, e_k$ ($k \geq 1$) and $\theta(x) = e_{i_x}$ where the index $1 \leq i_x \leq k$ is chosen according to the following rules:

$$i_x = \begin{cases} \delta(\Theta(x)^-, x) & \text{if } \delta(\Theta(x)^-, x) \neq * \wedge \delta(\Theta(x)^+, x) \neq * \\ k - \delta(x, \Theta(x)^+) & \text{if } \delta(\Theta(x)^-, x) = * \wedge \delta(\Theta(x)^+, x) \neq * \\ \delta(\Theta(x)^-, x) & \text{if } \delta(\Theta(x)^-, x) \neq * \wedge \delta(\Theta(x)^+, x) = * \end{cases} \quad (\text{B.27})$$

Note furthermore, for the current case $x.a^n \text{ new}$, we cannot have

$$\delta(\Theta(x)^-, x) = \delta(x, \Theta(x)^+) = *$$

otherwise this would imply $\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v} = \Theta(x)$ and $\mathcal{C}(\Theta(x)) = *$ that is impossible since new entities cannot be unbounded. Hence the choice of i_x is deterministic. Given such θ , let as usual θ^\pm be its extension, by construction we have:

$$\begin{aligned} h_i \circ \theta^\pm &= \Theta \\ \delta(x, y) &= \delta_i(x, y) \end{aligned} \quad (\text{B.28})$$

for all $x, y \in fv(\psi) \cup E_{q_i}^\pm$. Hence, we can conclude that:

$$v = (x.a^n \text{ new}, \Theta, \delta) = (x.a^n \text{ new}, h_i \circ \theta, \delta_i). \quad (\text{B.29})$$

A consequence of (B.28) is that $(h_i, \theta) \approx (\Theta, \delta)$ and by Lemma B.3.4 we have that $\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^1 = h_i \circ \mu^n(\theta(x))$. From $v \in AV_{q_i}$ it follows $\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^1 \in N_{q_i}$. Thus, by the properties of the generator (Def. 5.3.24) — in particular as stated by Lemma 5.3.19 — we have that new entities on the abstract level are related by the generator to the new entities on the concrete level, i.e., $\mu^n(\theta(x)) \in N_i^\sigma$ and therefore $\sigma^i, N_i^\sigma, \theta \models x.a^n$ new. Hence, by (B.29) and by definition of D_i , we conclude that $v \in D_i$ which proves (B.26).

– case $\psi = (x.a^n = y.a^m)$. Give a valuation $v = (x.a^n = y.a^m, \Theta, \delta) \in AV_{q_i}$, by Definition 5.7.15 we have $\Delta_{\gamma_{q_i}, v}(x.a^n, y.a^m) = 0$ that by definition implies

- a) $\llbracket x.a^n \rrbracket_{\gamma, v}^1 = \llbracket y.a^m \rrbracket_{\gamma, v}^1 = \perp$
- b) $\llbracket x.a^n \rrbracket_{\gamma, v}^1 = \llbracket y.a^m \rrbracket_{\gamma, v}^1 \neq \perp$ and $\llbracket x.a^n \rrbracket_{\gamma, v}^2 \neq *$
- c) $\llbracket x.a^n \rrbracket_{\gamma, v}^1 = \llbracket y.a^m \rrbracket_{\gamma, v}^1 \neq \perp$ and $\llbracket x.a^n \rrbracket_{\gamma, v}^2 = *$ and $(\delta(x, y) \oplus m = n \vee \delta(y, x) \oplus n = m)$

Let us discuss here b) and c) only. We need to show the existence of a suitable θ . Assume:

$$\begin{aligned} h_i^{-1}(\Theta(x)) &= e_1, \dots, e_{k_x} \\ h_i^{-1}(\Theta(y)) &= e_1, \dots, e_{k_y} \end{aligned}$$

with $k_x, k_y \geq 1$. For the selection of $\theta(x)$ and $\theta(y)$ we apply the same strategy employed for $x.a^n$ new. Namely, let $\theta(x) = e_{i_x}$ and $\theta(y) = e_{i_y}$ where i_x, i_y are chosen according to (B.30) and (B.31), respectively.

$$i_x = \begin{cases} \delta(\Theta(x)^-, x) & \text{if } \delta(\Theta(x)^-, x) \neq * \wedge \delta(\Theta(x)^+, x) \neq * \\ k - \delta(x, \Theta(x)^+) & \text{if } \delta(\Theta(x)^-, x) = * \wedge \delta(\Theta(x)^+, x) \neq * \\ \delta(\Theta(x)^-, x) & \text{if } \delta(\Theta(x)^-, x) \neq * \wedge \delta(\Theta(x)^+, x) = * \\ M < j < k_x - M & \text{if } \delta(\Theta(x)^-, x) = * \wedge \delta(\Theta(x)^+, x) = * \end{cases} \quad (\text{B.30})$$

$$i_y = \begin{cases} \delta(\Theta(y)^-, y) & \text{if } \delta(\Theta(y)^-, y) \neq * \wedge \delta(\Theta(y)^+, y) \neq * \\ k - \delta(y, \Theta(y)^+) & \text{if } \delta(\Theta(y)^-, y) = * \wedge \delta(\Theta(y)^+, y) \neq * \\ \delta(\Theta(y)^-, y) & \text{if } \delta(\Theta(y)^-, y) \neq * \wedge \delta(\Theta(y)^+, y) = * \\ M < j < k_y - M & \text{if } \delta(\Theta(y)^-, y) = * \wedge \delta(\Theta(y)^+, y) = * \end{cases} \quad (\text{B.31})$$

Since when both $\delta(\Theta(y)^-, x) = \delta(\Theta(y)^+, x) = *$ and $\delta(\Theta(y)^-, y) = \delta(\Theta(y)^+, y) = *$ the indexes i_x, i_y are still not completely determined, then we select them such that the following conditions are satisfied:

$$\begin{aligned} i_x - i_y &= n - m & \text{if } \delta(y, x) \oplus m = n \\ i_y - i_x &= m - n & \text{if } \delta(y, x) \oplus n = m. \end{aligned}$$

By construction we have that $h_i \circ \theta^\pm = \Theta$ and $\delta = \delta_i$, or in other words, $(h_i, \theta) \approx (\Theta, \delta)$. To prove that $v \in D_i$ it remains to show that by the previous choice of $\theta(x)$ and $\theta(y)$ we indeed have: $\sigma^i, N_i^\sigma, \theta \models x.a^n = y.a^m$.

Assume case b) and let

$$\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^2 = l \neq * \text{ and } e = \text{first}(h_i^{-1}(\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^1)).$$

Depending whether x is interpreted onto $\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}$ or not, by definition we have two possibilities, namely:

- (i) if $(x, \llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^-) \in \text{dom}(\delta)$ then by Definition 5.7.13 $\mathcal{C}_{\gamma_{q_i}}(x, j - 1) = n - l - 1$ and therefore:

$$\mu_i^{n-l}(\theta(x)) = e \quad (\text{B.32})$$

- (ii) otherwise $(\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^-, x) \in \text{dom}(\delta)$ in which case we have (again by Definition 5.7.13) $\delta(\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^-, x) = l - n$ and by consistency of δ we have

$$\mu_i^{l-n}(e) = \theta(x). \quad (\text{B.33})$$

Following the same lines, it is possible to show that depending on the interpretation of y w.r.t. $\llbracket y.a^m \rrbracket_{\gamma_{q_i}, v}$ either

$$\mu_i^{m-l}(\theta(y)) = e \quad (\text{B.34})$$

or

$$\mu_i^{l-m}(e) = \theta(y) \quad (\text{B.35})$$

From the previous case distinction we can determine the precise interpretation on the concrete level for $\mu_i^n(\theta(x))$ and $\mu_i^m(\theta(y))$. In particular, from (B.32) and (B.33) we obtain:

$$\mu_i^n(\theta(x)) = \mu_i^l(\mu_i^{n-l}(\theta(x))) = \mu_i^l(e) \quad (\text{B.36})$$

$$\mu_i^n(\theta(x)) = \mu_i^n(\mu_i^{l-n}(e)) = \mu_i^l(e) \quad (\text{B.37})$$

For the variable y , from (B.34) and (B.35) we have

$$\mu_i^m(\theta(y)) = \mu_i^l(\mu_i^{m-l}(\theta(y))) = \mu_i^l(e) \quad (\text{B.38})$$

$$\mu_i^m(\theta(y)) = \mu_i^m(\mu_i^{l-m}(e)) = \mu_i^l(e) \quad (\text{B.39})$$

from which we can conclude that in case (b), in any possibility,

$$\mu_i^n(\theta(x)) = \mu_i^m(\theta(y)) = \mu_i^l(e).$$

Now, assume case (c). Let $\delta(x, y) = l$ (the case $\delta(y, x) = l$ is symmetrical). By hypothesis, we have $\delta(x, y) \oplus m = n$ that implies

$l < K(\phi)$ (by assumption on stretching see page 5.7.1) and therefore, since δ is consistent we obtain $\mu_i^l(\theta(x)) = \theta(y)$. But then

$$\mu_i^n(\theta(x)) = \mu_i^{l+m}(\theta(x)) = \mu_i^m(\mu_i^l(\theta(x))) = \mu_i^m(\theta(y)).$$

Hence also in case (c) $\mu_i^n(\theta(x)) = \mu_i^m(\theta(y))$. Thus, by definition of the semantics of $\mathcal{N}\ell\ell\text{TL}$ for both cases (b) and (c) we conclude $\sigma^i, N_i^\sigma, \theta \models x.a^n = y.a^m$ and thus $v \in D_i$.

– case $x.a^n \rightsquigarrow y.a^m$.

Suppose $v = (x.a^n \rightsquigarrow y.a^m, \Theta, \delta) \in AV_{q_i}$. By definition of atomic valuation $\Delta(x.a^n, y.a^m) = \top$ or $\Delta(x.a^n, y.a^m) = 0$ and therefore by definition of Δ we have the following possibilities:

- a) $\llbracket x.a^n \rrbracket_{\gamma,v}^1 \neq \llbracket y.a^m \rrbracket_{\gamma,v}^1$ and $\exists j > 0 : \mu^j(\llbracket x.a^n \rrbracket_{\gamma,v}^1) = \llbracket y.a^m \rrbracket_{\gamma,v}^1$
- b) $\llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp$ and $\llbracket x.a^n \rrbracket_{\gamma,v}^2 < \llbracket y.a^m \rrbracket_{\gamma,v}^2$
- c) $\llbracket x.a^n \rrbracket_{\gamma,v}^1 = \llbracket y.a^m \rrbracket_{\gamma,v}^1 \neq \perp$ and $\llbracket x.a^n \rrbracket_{\gamma,v}^2 = \llbracket y.a^m \rrbracket_{\gamma,v}^2 = *$ and $(\delta(x, y) \oplus m > n \vee \delta(y, x) \oplus n < m)$.

We define $\theta(x)$ and $\theta(y)$ according to (B.30) and (B.31) as we have done for the case of equality. Therefore we have by construction $\Theta = h_i \circ \theta$ and $\delta = \delta_i$. In order to have $v \in D_i$ it remains to show that $\sigma^i, N_i, \theta \models x.a^n \rightsquigarrow y.a^m$ for both a), b) as well as c).

- a) It is straightforward that at the concrete level, there exists $j' \geq j$ such that $\mu_i^{n+j'}(\theta(x)) = \mu_i^m(\theta(y))$ and therefore by definition of the semantics of $\mathcal{N}\ell\ell\text{TL}$ we can conclude $\sigma^i, N_i, \theta \models x.a^n \rightsquigarrow y.a^m$.
- b) As we have done for case $x.a^n = y.a^m$, let $\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^2 = l$ and $e = \text{first}(h_i^{-1}(\llbracket x.a^n \rrbracket_{\gamma_{q_i}, v}^2))$ then by (B.36) and (B.37) we have $\mu_i^n(\theta(x)) = \mu_i^l(e)$. By hypothesis there exists $l' \geq l$ such that $\mu_i^m(\theta(y)) = \mu_i^{l'}(e)$. But this implies $\mu_i^m(\theta(y)) = \mu_i^{l'-1}(\mu_i^n(\theta(x)))$ and by definition $\sigma^i, N_i, \theta \models x.a^n \rightsquigarrow y.a^m$.
- c) if $\delta(x, y) \oplus m \geq n$ then $(x, y) \in \text{dom}(\delta)$. We can have either $\delta(x, y) = *$, then on the concrete level since δ is consistent there must exist $l' > n$ such that $\mu_i^{l'}(\theta(x)) = \theta(y)$ that is enough to conclude that $x.a^n$ reaches $y.a^m$. Otherwise, $\delta(x, y) = c \neq *$ and since by hypothesis $c + m \geq n$ then, for the consistency of δ , at the concrete level $\mu_i^{c+m-n}(\mu_i^n(\theta(x))) = \theta(y)$. So we are done.

Finally if $\delta(y, x) \oplus n \leq m$ it is straightforward to verify that on the concrete level $x.a^n \rightsquigarrow y.a^m$.

Inductive step

- case $\psi = \neg\psi'$. According to the definition of atom we have to prove:

$$v = (\neg\psi', h_i \circ \theta, \delta) \in D_i \Leftrightarrow (\psi', h_i \circ \theta, \delta) \notin D_i.$$

Straightforward from the definition of D_i , and semantics of $\mathcal{N}allTL$, in fact:

$$\begin{aligned} v = (\neg\psi', h_i \circ \theta, \delta) \in D_i &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models \neg\psi' \\ &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \not\models \psi' \\ &\Leftrightarrow (\psi', h_i \circ \theta, \delta) \notin D_i \end{aligned}$$

- **case** $\psi = \psi_1 \vee \psi_2$. As the previous case, straightforward from the definition of D_i , and semantics of $\mathcal{N}allTL$, in fact:

$$\begin{aligned} (\psi_1 \vee \psi_2, h_i \circ \theta, \delta) \in D_i &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models \psi_1 \vee \psi_2 \\ &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \upharpoonright \psi_1 \models \psi_1 \text{ or } \sigma^i, N_i^\sigma, \theta \upharpoonright \psi_2 \models \psi_2 \\ &\Leftrightarrow (\psi_1, h_i \circ \theta \upharpoonright \psi_1, \delta \upharpoonright \psi_1) \in D_i \\ &\quad \text{or } (\psi_2, h_i \circ \theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2) \in D_i. \end{aligned}$$

- **case** $\psi = \exists x.\psi'$.

[\Rightarrow] Suppose $(\exists x.\psi', h_i \circ \theta, \delta) \in D_i$ then $\sigma^i, N_i^\sigma, \theta \models \exists x.\psi'$. It follows that there exists an $e \in E_i^\sigma$ such that $\sigma^i, N_i^\sigma, \theta\{e/x\} \models \psi'$. By general assumption (see Page 178), $x \in fv(\psi')$ and hence $\text{dom}(\theta\{e/x\}) \subseteq fv(\psi')$. Thus,

$$(\psi', h_i \circ \theta\{e/x\}, \delta'_i) \in D_i$$

where $\delta'_i(z, y) = \min \{[n] \mid \mu_i^n \circ \theta\{x/e\}^\pm(z) = \theta\{x/e\}^\pm(y)\}$.

[\Leftarrow] If $(\psi', h_i \circ \theta\{e/x\}, \delta'_i) \in D_i$ where

$$\delta'_i(z, y) = \min \{[n] \mid \mu_i^n \circ \theta\{x/e\}^\pm(z) = \theta\{x/e\}^\pm(y)\}$$

then $\sigma^i, N_i^\sigma, \theta\{e/x\} \models \psi'$. And therefore $\sigma^i, N_i^\sigma, \theta \models \exists x.\psi'$ and finally $(\exists x.\psi', h_i \circ \theta, \delta) \in D_i$ where $\delta = \delta'_i \upharpoonright \psi$.

- **case** $\psi = \neg X\psi'$. We have:

$$\begin{aligned} (\neg X\psi', h_i \circ \theta, \delta_i) \in D_i &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models \neg X\psi' \\ &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \not\models X\psi' \\ &\Leftrightarrow \sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \not\models \psi' \\ &\Leftrightarrow \sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \models \neg\psi' \\ &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models X\neg\psi' \\ &\Leftrightarrow (X\neg\psi', h_i \circ \theta, \delta_i) \in D_i \end{aligned}$$

- **case** $\psi = \psi_1 \cup \psi_2$. To prove this case we use the following known equivalence:

$$\sigma, N, \theta \models \psi_1 \cup \psi_2 \Leftrightarrow \sigma, N, \theta \models \psi_2 \vee (\psi_1 \wedge X(\psi_1 \cup \psi_2))$$

for all allocation triple (σ, N, θ) . Therefore we have:

$$\begin{aligned}
(\psi_1 \mathbf{U} \psi_2, h_i \circ \theta, \delta) \in D_i &\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models \psi_1 \mathbf{U} \psi_2 \\
&\Leftrightarrow \sigma^i, N_i^\sigma, \theta \models \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2)) \\
&\Leftrightarrow \sigma^i, N_i^\sigma, \theta \upharpoonright \psi_2 \models \psi_2 \text{ or} \\
&\quad (\sigma^i, N_i^\sigma, \theta \upharpoonright \psi_1 \models \psi_1 \text{ and} \\
&\quad \quad \sigma^i, N_i^\sigma, \theta \models \mathbf{X}(\psi_1 \mathbf{U} \psi_2)) \\
&\Leftrightarrow (\psi_2, h_i \circ \theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2) \in D_i \text{ or} \\
&\quad (\psi_1, h_i \circ \theta \upharpoonright \psi_1, \delta \upharpoonright \psi_1) \in D_i \text{ and} \\
&\quad (\mathbf{X}(\psi_1 \mathbf{U} \psi_2), h_i \circ \theta, \delta) \in D_i
\end{aligned}$$

□

Proposition 5.7.25. ϕ is \mathcal{H} -satisfiable if and only if there exists a path in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ .

Proof.

[\Leftarrow] If there exists π in $G_{\mathcal{H}}(\phi)$ that fulfils ϕ , by definition this implies that ϕ is satisfied by an allocation triple (σ, N, θ) generated by the underlying run of π .

[\Rightarrow] Now assume that ϕ is \mathcal{H} -satisfiable, and let $\rho = q_0 \lambda_0 q_1 \lambda_1 \cdots$ be a run generating a triple (σ, N, θ) with generator $(h_i)_{i \in \mathbb{N}}$ such that $\sigma, N, \theta \models \phi$. We construct a path $\pi = (q_0, D_0) \lambda_0 (q_1, D_1) \lambda_1 \cdots$ that fulfils ϕ as follows:

$$D_i = \{(\psi, h_i \circ \theta, \delta_i) \mid \psi \in CL(\phi), \sigma^i, N_i^\sigma, \theta \models \psi\}$$

where let θ^\pm be the extension of θ to the set $E_{q_i}^\pm$ as defined by (B.16) (cf. page 283) then δ_i is defined for all $x, y \in \text{fv}(\psi) \cup E_{q_i}^\pm$ by:

$$\delta_i(x, y) = \min \{[n] \mid \mu_i^n \circ \theta^\pm(x) = \theta^\pm(y)\}.$$

First of all, by Lemma B.3.5 (q_i, D_i) is an atom for all $i \in \mathbb{N}$. Secondly, we show that π is indeed a path by proving that it satisfies the conditions of Definition 5.7.23.

1. By the construction of π .
2. By contradiction. Assume that there exists an $i \geq 0$ such that $(q_i, D_i) \rightarrow_{\lambda_i} (q_{i+1}, D_{i+1})$ is not a transition of G . Take the minimal such i . Then one of the following must hold:
 - i) $q_i \rightarrow_{\lambda_i} q_{i+1}$ is not a transition in \mathcal{H} . But this contradicts the fact that ρ is a run of \mathcal{H} .

ii) By contradiction assume that there exists $(X\psi, h_i \circ \theta, \delta_i) \in D_i$ such that $(\psi, \Theta', \delta') \notin D_{i+1}$ for all $(\psi, \Theta', \delta') \in [\lambda_i \circ (\psi, h_i \circ \theta, \delta_i)]$. But then also $(\psi, h_{i+1} \circ \lambda_i^\sigma \circ \theta, \delta_{i+1}) \notin D_{i+1}$ since $(\psi, h_{i+1} \circ \lambda_i^\sigma \circ \theta, \delta_{i+1}) \in [\lambda_i \circ (\psi, h_i \circ \theta, \delta_i)]$ (see Lemma B.3.3); hence this would imply $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta \not\models \psi$. But then also $\sigma^i, N_i^\sigma, \theta \not\models X\psi$, contradicting the construction of D_i .

Assume $(\psi, \Theta', \delta') \in [\lambda_i \circ (\psi, \Theta, \delta)] \cap D_{i+1}$, but $(X\psi, \Theta, \delta) \notin D_i$. By construction of D_{i+1} , the tuple (ψ, Θ', δ') must be of the form $(\psi, h_{i+1} \circ \theta, \delta_{i+1})$ and $\sigma^{i+1}, N_{i+1}^\sigma, \theta \models \psi$. Condition 1 of Definition 5.7.20 implies that $\text{cod}(h_{i+1} \circ \theta) \subseteq \text{cod}(\lambda_i)$. Then, since $\lambda_i^\sigma \triangleright \lambda_i$, by Lemma 5.3.19 we have $\text{cod}(\theta) \cap N_{i+1}^\sigma = \emptyset$. Therefore there exists $\theta' : \text{fv}(\psi) \rightarrow \text{Ent}$ such that

$$\lambda_i^\sigma \circ \theta' = \theta.$$

Now by the semantics of $\mathcal{A}llTL$, $\sigma^{i+1}, N_{i+1}^\sigma, \lambda_i^\sigma \circ \theta' \models \psi$ implies $\sigma^i, N_i^\sigma, \theta' \models X\psi$. Thus by definition of D_i we must have $(X\psi, h_i \circ \theta', \delta_i) \in D_i$. Contradiction.

3. Assume $(\psi_1 \cup \psi_2, h_i \circ \theta, \delta_i) \in D_i$. By the construction of D_i we have $\sigma^i, N_i^\sigma, \theta \models \psi_1 \cup \psi_2$. Therefore $\sigma^j, N_j^\sigma, \lambda_{j-1}^\sigma \circ \dots \circ \lambda_i^\sigma \circ \theta \upharpoonright \text{fv}(\psi_2) \models \psi_2$ for some $j \geq i$. Due to the construction of D_j , it follows that $(\psi_2, h_j \circ \lambda_{j-1}^\sigma \circ \dots \circ \lambda_i^\sigma \circ \theta \upharpoonright \psi_2, \delta_j \upharpoonright \psi_2) \in D_j$ and Lemma B.3.3, $(\psi_2, h_j \circ \lambda_{j-1}^\sigma \circ \dots \circ \lambda_i^\sigma \circ \theta \upharpoonright \psi_2, \delta_j \upharpoonright \psi_2) \in [\lambda_{j-1} \circ \dots \circ \lambda_i \circ (\psi_2, h_i \circ \theta \upharpoonright \psi_2, \delta_i \upharpoonright \psi_2)]_{q_i, q_j}$.

Thus, π is a path and it is fulfilling since it satisfies ϕ by construction. \square

Proposition 5.7.27. If π is fulfilling path in $G_{\mathcal{H}}(\phi)$, then $\text{Inf}(\pi)$ is a self-fulfilling SCS of $G_{\mathcal{H}}(\phi)$.

Proof. Let $G' = \text{Inf}(\pi)$. G' is strongly connected. From the definition of infinite set it follows that there exists $i \geq 0$ such that the atoms in $\pi^i = (q_i, D_i)\lambda_i(q_{i+1}, D_{i+1})\lambda_{i+1} \dots$ are precisely those in G' . Furthermore, if there is an atom $A \in G'$ such that $(\psi_1 \cup \psi_2, \Theta, \delta) \in D_A$, then there exists $j \geq i$: $(q_j, D_j) = A$. By condition 3 of Def. 5.7.23 we have that there exists $n \geq j$ such that $(\psi_2, \Theta', \delta') \in D_n$ where $(\psi_2, \Theta', \delta') \in [\lambda_{n-1} \circ \dots \circ \lambda_j \circ (\psi_2, \Theta \upharpoonright \psi_2, \delta \upharpoonright \psi_2)]$. But then $(q_n, D_n) \in G'$, hence G' is self-fulfilling. \square

C

Proofs of Chapter 6

The next proof uses the following facts implied by the definition of Ω :

$$\forall a \in n(P) : \forall Q : \varepsilon(a) \uplus \Omega(a, Q, k, \text{tt}) = \Omega(a, Q, k, \text{tt}) \quad (\text{C.1})$$

$$\forall Q : \varepsilon(n(Q)) \uplus \Omega(@, Q, k, \text{tt}) = \Omega(@, Q, k, \text{tt}). \quad (\text{C.2})$$

Moreover the definition of union of states implies that:

$$\forall a : \varepsilon(a) \uplus \varepsilon(a) = \varepsilon(a)$$

Conjecture 6.3.29. Let \tilde{P} be a well-indexed process. If $\text{noi}(\tilde{P}) \rightarrow Q$ then there exists λ and a well-indexed process \tilde{Q}' such that $\mathcal{D}(\tilde{P}) \rightarrow_\lambda (\mathcal{D}(\tilde{Q}') \uplus n(\tilde{P}))$ and $\text{noi}(\tilde{Q}') \equiv Q$.

Proof. (Sketch) We give a sketch proof of this conjecture by induction on the depth of the derivation of $P \rightarrow Q$. The last rule used in the transition $P \rightarrow Q$ can be any of those listed in Table 6.2.2. We distinguish the possible cases¹

- case (Red In). We have that \tilde{P} is of the form $\tilde{P} = n_i[\text{in } m.\tilde{P}'|\tilde{Q}''|]m_j[\tilde{R}']$ and $P = n[\text{in } m.P'|Q''|]m[R']$. Therefore by the (Red In) we have $Q = m[n[P'|Q''|]R']$. Let $\tilde{Q}' = m_j[n_i[\tilde{P}'|\tilde{Q}''|]\tilde{R}'$, we have obviously $\text{noi}(\tilde{Q}') \equiv Q$.

¹In this proof, we write indexed processes with a tilde. The same process without tilde is the result of the application of the function noi . That is, for any indexed process \tilde{R} , $\text{noi}(\tilde{R}) = R$.

By definition of $\Omega(@, \tilde{P}, 1, \text{tt})$ we have:

$$\begin{aligned}
\Omega(@, \tilde{P}, 1, \text{tt}) &= \Omega(@, n_i[\text{in } m. \tilde{P}' | \tilde{Q}''], 1, \text{tt}) \cup \Omega(@, m_j[\tilde{R}'], 1, \text{tt}) \\
&= \varepsilon(@) \uplus \varepsilon(n_i) \uplus \Omega(n_i, \text{in } m. \tilde{P}' | \tilde{Q}'', 1, \text{ff}) \uplus \\
&\quad \Omega(m_j, \tilde{R}', 1, \text{tt}) \uplus \\
&\quad \langle \{e_{n_i}^c, \text{ho}(@)\}, \{e_{n_i}^c, \text{ho}(@)\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(@)\}} \rangle, \\
&\quad \quad \quad \langle \{e_{n_i}^c, \rho(\text{in } m. \tilde{P}' | \tilde{Q}'')\}, \text{ho}(@), \mathbf{0} \rangle \rangle \uplus \\
&\quad \langle \{e_{m_j}^c, \text{ho}(@)\}, \{e_{m_j}^c, \text{ho}(@)\}, \mathbf{1}_{\{e_{m_j}^c, \text{ho}(@)\}} \rangle, \\
&\quad \quad \quad \langle \{e_{m_j}^c, \rho(\tilde{R}')\}, \text{ho}(@), \mathbf{0} \rangle \rangle
\end{aligned}$$

Symmetrically,

$$\begin{aligned}
\Omega(@, \tilde{Q}', 1) &= \varepsilon(@) \uplus \varepsilon(m_j) \uplus \varepsilon(n_i) \uplus \\
&\quad \Omega(n_i, \tilde{P}' | \tilde{Q}'', 1, \text{tt}) \uplus \Omega(m_j, \tilde{R}', 1, \text{tt}) \uplus \\
&\quad \langle \{\text{ho}(@), e_{m_j}^c\}, \{e_{m_j}^c, \text{ho}(@)\}, \mathbf{1}_{\{e_{m_j}^c, \text{ho}(@)\}} \rangle \\
&\quad \quad \quad \langle \{e_{m_j}^c, \rho(\tilde{R}')\}, \text{ho}(@), \mathbf{0} \rangle \rangle \uplus \\
&\quad \langle \{\text{ho}(m_j), e_{n_i}^c\}, \{e_{n_i}^c, \text{ho}(m_j)\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(m_j)\}} \rangle, \\
&\quad \quad \quad \langle \{\text{ho}(m_j), \mathbf{0}\}, \{e_{n_i}^c, \rho(\tilde{P}' | \tilde{Q}'')\} \rangle
\end{aligned}$$

Let $\mathcal{D}(\tilde{P}) = (h_{\text{cf}} \circ h_1^{-1}(\gamma_{\Omega(@, \tilde{P}, 1, \text{tt})}), P_{\Omega(@, \tilde{P}, 1, \text{tt})}\{\mathbf{0}/\text{is}(@)\})$ where the morphism h_1 is given by the definition of \mathcal{D} . The configuration $\gamma_{\mathcal{D}(\tilde{P})}$ — even after the transformation of representation produced by the application of $h_{\text{cf}} \circ h_1^{-1}$ — can be considered as the union of several configurations. For the concrete part of $\gamma_{\Omega(@, \tilde{P}, 1, \text{tt})}$, for example the following must be sub-configurations of $\gamma_{\mathcal{D}(\tilde{P})}$:

$$\begin{aligned}
\gamma' &= (\{e_{n_i}^c, \text{ho}(@)\}, \{e_{n_i}^c, \text{ho}(@)\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(@)\}}) \\
\gamma'' &= (\{e_{m_j}^c, \text{ho}(@)\}, \{e_{m_j}^c, \text{ho}(@)\}, \mathbf{1}_{\{e_{m_j}^c, \text{ho}(@)\}}).
\end{aligned}$$

In fact, we can safely assume that even after the application of $h_{\text{cf}} \circ h_1^{-1}$, there is a concrete entity of type m_j and a concrete one of type n_i that have a reference to $\text{ho}(@)$ (by L -canonicity). Since the actual entity does not matter, we can assume for simplicity that these are precisely $e_{n_i}^c, e_{m_j}^c$. Hence, up to isomorphism, $\gamma_{\mathcal{D}(\tilde{P})}$ is of the form

$$\gamma_{\mathcal{D}(\tilde{P})} = \gamma' \uplus \gamma'' \uplus \gamma'''$$

for some γ''' .

Rule **In** of Table 6.3.10 starts by the application of

$$\text{IOUp}(\mathcal{D}(\tilde{P}), \text{in } m. \tilde{P}' | \tilde{Q}'', e_{n_i}^c \mapsto \text{ho}(m_j))$$

that in turn applies first $move(\gamma_{\mathcal{D}(\tilde{P})}, e_{n_i}^c \mapsto \mathbf{ho}(m_j))$. The latter substitute γ' by the appropriate pointer update dictated by its definition. To the resulting configuration, activation applies:

$$\begin{aligned}
 & act_{\tilde{P}'|\tilde{Q}'', n_i}(\gamma_{\varepsilon(@)} \uplus (\{e_{n_i}^c, \mathbf{ho}(m_j)\}, \{(e_{n_i}^c, \mathbf{ho}(m_j))\}) \mathbf{1}_{\{e_{n_i}^c, \mathbf{ho}(m_j)\}}) \uplus \\
 & \quad (\{e_{m_j}^c, \mathbf{ho}(@)\}, \{(e_{m_j}^c, \mathbf{ho}(@))\}, \mathbf{1}_{\{e_{m_j}^c, \mathbf{ho}(@)\}}) \uplus \\
 & \quad \gamma_{\Omega(n_i, \text{in } m. \tilde{P}'|\tilde{Q}'', 1, \text{ff})} \uplus \gamma_{\Omega(m_i, \tilde{R}', 1, \text{tt})} \\
 = & \gamma_{\varepsilon(@)} \uplus (\{e_{n_i}^c, \mathbf{ho}(m_j)\}, \{(e_{n_i}^c, \mathbf{ho}(m_j))\}, \mathbf{1}_{\{e_{n_i}^c, \mathbf{ho}(m_j)\}}) \uplus \\
 & (\{e_{m_j}^c, \mathbf{ho}(@)\}, \{(e_{m_j}^c, \mathbf{ho}(@))\}, \mathbf{1}_{\{e_{m_j}^c, \mathbf{ho}(@)\}}) \uplus \\
 & \gamma_{\Omega(n_i, \tilde{P}'|\tilde{Q}'', 1, \text{tt})} \uplus \gamma_{\Omega(m_i, \tilde{R}', 1, \text{tt})}
 \end{aligned}$$

Therefore, we have (also using (C.1)) that the configuration of

$$\text{IOUp}(\mathcal{D}(\tilde{P}), \text{in } m. \tilde{P}'|\tilde{Q}, e_{n_i}^c \mapsto \mathbf{ho}(m_j))$$

is $\gamma_{\Omega(@, \tilde{Q}', 1)}$. For the P component we have:

$$\text{P}_{\mathcal{D}(\tilde{P})}\{\text{P}_{\mathcal{D}(\tilde{P})}(e) \setminus \{\text{in } m. \tilde{P}'|\tilde{Q}''\} \cup \rho(\tilde{P}'|\tilde{Q}'')/e_{n_i}^c\} = \text{P}_{\mathcal{D}(\tilde{P})}\{\rho(\tilde{P}'|\tilde{Q}'')/e_{n_i}^c\}$$

which this is precisely: $\text{P}_{\Omega(@, \tilde{Q}', 1, \text{tt})}$. Thus we have:

$$\begin{aligned}
 \text{IOUp}(\mathcal{D}(\tilde{P}), \text{in } m. \tilde{P}'|\tilde{Q}, e_{n_i}^c \mapsto \mathbf{ho}(m_j)) &= \langle \gamma_{\Omega(@, \tilde{Q}', 1, \text{tt})}, \text{P}_{\Omega(@, \tilde{Q}', 1, \text{tt})} \rangle \\
 &= \Omega(@, \tilde{Q}', 1, \text{tt})
 \end{aligned}$$

Note that since $n(\tilde{Q}') = n(\tilde{P})$, and by (C.2) we have

$$\Omega(@, \tilde{Q}', 1, \text{tt}) \uplus \varepsilon(n(\tilde{P})) = \Omega(@, \tilde{Q}', 1, \text{tt})$$

Now, according to the **In** rule of Table 6.3.10, the target state is the canonical form of a state in the safe expansion, together with the updated set of capabilities we have just computed. Thus:

$$\mathcal{D}(\tilde{P}) \rightarrow_{\lambda} (h_{\text{cf}} \circ h^{-1}(\gamma'), \text{P}_{\Omega(@, \tilde{Q}', 1, \text{tt})}).$$

for all $(\gamma', h) \in \text{SExp}(\gamma_{\Omega(@, \tilde{Q}', 1, \text{tt})})$. Among the canonical configurations (for the properties of the canonical form) there must be one that correspond to $\text{cf}(\gamma'')$ where γ'' is a safe configuration expanded by a morphism as described by the definition of $\mathcal{D}(\tilde{Q}')$. This is precisely the one that expands the unbounded entities by a chain of L concrete entities followed by an unbounded entity.

- case (Red Out). Similar to the (In Rule) case.
- case (Red Open). We have $\tilde{P} = \text{openn}.\tilde{P}'|n_i[\tilde{R}]$ and $\text{noi}(\tilde{P}) = \text{openn}.\tilde{P}'|n[R]$ and $Q = \tilde{P}'|R$. Let $\tilde{Q}' = \tilde{P}'|\tilde{R}$ then $\text{noi}(\tilde{Q}') \equiv Q$.

By definition of $\Omega(@, \tilde{P}, 1, \text{tt})$ we have:

$$\begin{aligned} \Omega(@, \tilde{P}, 1, \text{tt}) &= \Omega(@, \text{open } n.\tilde{P}', 1, \text{tt}) \uplus \Omega(@, n[\tilde{R}], 1, \text{tt}) \\ &= \varepsilon(@) \uplus \Omega(@, \tilde{P}', 1, \text{ff}) \uplus \Omega(n_i, \tilde{R}, 1, \text{tt}) \uplus \varepsilon(n_i) \uplus \\ &\quad \langle \{e_{n_i}^c, \text{ho}(@)\}, \{(e_{n_i}^c, \text{ho}(@))\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(@)\}} \rangle, \\ &\quad \{(e_{n_i}^c, \rho(\tilde{R})), (\text{ho}(@), \mathbf{0})\} \end{aligned}$$

For the process \tilde{Q}' we have:

$$\Omega(@, \tilde{P}'|\tilde{R}, 1, \text{tt}) = \Omega(@, \tilde{P}', 1, \text{tt}) \uplus \Omega(@, \tilde{R}, 1, \text{tt})$$

Let $\mathcal{D}(\tilde{P}) = (h_{\text{cf}} \circ h_1^{-1}(\gamma_{\Omega(@, \tilde{P}, 1, \text{tt})}), \mathbf{P}_{\Omega(@, \tilde{P}, 1, \text{tt})})$ where h_1 is the morphism determined by the definition of \mathcal{D} . As observed for the case (Red In), because of the canonical form, we can separate the part of the configuration $\gamma_{\mathcal{D}(\tilde{P})}$ related to the execution of the rule from the rest of the configuration. Therefore, the first step of the application of OpenUp result in:

$$\begin{aligned} &\text{OpenUp}(\mathcal{D}(\tilde{P}), \text{is}(@), e_{n_i}^c, \tilde{P}') \\ &= \langle \text{act}_{@, \tilde{P}'}(\text{dissolve}(\gamma_{\mathcal{D}(\tilde{P})}, \text{ho}(@), e_{n_i}^c)), \\ &\quad \mathbf{P}_{\mathcal{D}(\tilde{P})} \{ \mathbf{P}_{\mathcal{D}(\tilde{P})}(\text{is}(@)) \setminus \{ \text{open } n.\tilde{P}' \} \cup \rho(\tilde{P}') \cup \mathbf{P}_{\mathcal{D}(\tilde{P})}(e_{n_i}^c) / \text{is}(@) \} \rangle \\ &= \langle \text{act}_{@, \tilde{P}'}(\text{dissolve}(\gamma_{\mathcal{D}(\tilde{P})}, \text{ho}(@), e_{n_i}^c)), \mathbf{P}_{\mathcal{D}(\tilde{P})} \{ \rho(\tilde{P}'|\tilde{R}) / \text{is}(@) \} \rangle \end{aligned}$$

Let's now concentrate only on the configuration part:

$$\text{act}_{@, \tilde{P}'}(\text{dissolve}(\gamma_{\mathcal{D}(\tilde{P})}, \text{ho}(@), e_{n_i}^c))$$

where

$$\begin{aligned} \gamma_{\mathcal{D}(\tilde{P})} &= \gamma_{\varepsilon(@)} \uplus \gamma_{\Omega(@, \tilde{P}', 1, \text{ff})} \uplus \gamma_{\Omega(n_i, \tilde{R}, 1, \text{tt})} \uplus \gamma_{\varepsilon(n_i)} \\ &\quad \uplus \langle \{e_{n_i}^c, \text{ho}(@)\}, \{(e_{n_i}^c, \text{ho}(@))\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(@)\}} \rangle. \end{aligned}$$

Moreover, according to the definition of *dissolve* (see page 221), after the deletion of $e_{n_i}^c$ that cancel the subset

$$\langle \{e_{n_i}^c, \text{ho}(@)\}, \{(e_{n_i}^c, \text{ho}(@))\}, \mathbf{1}_{\{e_{n_i}^c, \text{ho}(@)\}} \rangle$$

the pointer structure must be modified according to the following schema:

$$\mu' = \mu_{\gamma_q} \{ \text{ho}(@) / \mu_{\gamma_q}(e_{n_i}^c), \text{ho}(@) / \mu_{\gamma_q}^{-1}(\text{ho}(n_i)) \}.$$

The first kind of update that links other possible copies of n_i to $\text{ho}(@)$ is implicit in the representation of the configuration we are using. Namely: if there would be other entities linked to $e_{n_i}^c$, since we consider the configuration as the union of the part related to $e_{n_i}^c$ and the rest of the configuration, for the very nature of the union operator, the other copies of n_i are

already linked to $\text{ho}(\textcircled{a})$ (and represented in other part of the union). For the second update, note that the entities in $\mu_{\gamma_q}^{-1}(\text{ho}(n_i))$ that are those inside n_i , in $\gamma_{\mathcal{D}(\tilde{P})}$ are contained in the part of the configuration build by $\Omega(n_i, \tilde{R}, 1, \text{tt})$. In fact this is the only part of the configuration that embeds ambients inside n_i . Note furthermore, that these ambients must be precisely the set $\text{enab}(\tilde{R})$. Therefore, moving these ambients from n_i into \textcircled{a} , corresponds to replace the part of the configuration $\Omega(n_i, \tilde{R}, 1, \text{tt})$ by $\Omega(\textcircled{a}, \tilde{R}, 1, \text{tt})$. We conclude that after the pointer update dictated by *dissolve* to the configuration $\gamma_{\mathcal{D}(\tilde{P})}$ we have:

$$\text{act}_{\textcircled{a}, \tilde{P}'}(\gamma_{\varepsilon(\textcircled{a})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{P}', 1, \text{ff})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{R}, 1)} \uplus \gamma_{\varepsilon(n_i)})$$

and by the application of *act* that replace $\Omega(\textcircled{a}, \tilde{P}', 1, \text{ff})$ by $\Omega(\textcircled{a}, \tilde{P}', 1, \text{tt})$ we obtain:

$$\begin{aligned} & \text{act}_{\textcircled{a}, \tilde{P}'}(\gamma_{\varepsilon(\textcircled{a})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{P}', 1, \text{ff})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{R}, 1, \text{tt})} \uplus \varepsilon(n_i)) \\ = & \gamma_{\varepsilon(\textcircled{a})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{P}', 1, \text{tt})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{R}, 1, \text{tt})} \uplus \gamma_{\varepsilon(n_i)} \\ = & \gamma_{\varepsilon(\textcircled{a})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{P}' | \tilde{R}, 1, \text{tt})} \uplus \gamma_{\varepsilon(n_i)} \\ = & \gamma_{\varepsilon(\textcircled{a})} \uplus \gamma_{\Omega(\textcircled{a}, \tilde{Q}', 1, \text{tt})} \uplus \gamma_{\varepsilon(n_i)} \\ = & \gamma_{\Omega(\textcircled{a}, \tilde{Q}', 1, \text{tt})} \uplus \gamma_{\varepsilon(n_i)}. \end{aligned}$$

Now, according to the **Open** rule of Table 6.3.10, the target states are the canonical form of the state in the safe expansion, among which there is h one such that $(h_{\text{cf}} \circ h^{-1}(\gamma_{\Omega(\textcircled{a}, \tilde{Q}', 1)} \uplus \varepsilon(n_i)), \mathbf{P}_{\Omega(\textcircled{a}, \tilde{Q}', 1)}) = \mathcal{D}(\tilde{Q}') \uplus \varepsilon(n_i)$ using the same observation we applied for **In**. Moreover, since $n(\tilde{P}) = \{n_i\}$, also in this case we conclude:

$$\mathcal{D}(\tilde{P}) \rightarrow_{\lambda} \mathcal{D}(\tilde{Q}') \uplus \varepsilon(n(\tilde{P})).$$

The other cases (Red Res), (Red Amb), (Red Par), (Red \equiv) follow by induction and are not reported here. \square

Notations

General and Miscellaneous

\rightharpoonup	partial function
\cong	isomorphism
$\text{dom}(f)$	domain of the function f
$\text{cod}(f)$	codomain of the function f
\circ	function composition
$f \upharpoonright E$	restriction of the domain of f to E
B_k	k -th Bell number
$fv(\phi)$	free variables of ϕ
$fbv(\phi)$	free and bounded variable of ϕ
$[\cdot]$	semantics function
ι	projection function
\uplus	disjoint union
\mathcal{M}	multi-set
Ent	countable set of entities
LVAR	set of logical variables
PVAR	set of program variables
σ	(folded) allocation sequence
E_i^σ	set of entities at state i -th of σ
N_i^σ	set of new entities at i state of σ
θ_i^σ	valuation at state i of σ
N	set of entities initially new
θ	partial valuation of free logical variables
\models_u	satisfaction relation on unfolded allocation sequences
\models_f	satisfaction relation on folded allocation sequences
\mathcal{A}	ABA
\mathcal{H}	HABA
\mathcal{A}_p	concrete semantics of program p
\mathcal{H}_p	symbolic semantics of program p

\mathcal{F}	set of sets of accept states
ρ	run
$\text{runs}(\mathcal{H})$	set of \mathcal{H} runs
$\text{Gen}(\rho)$	set of allocation sequences generated by ρ
$\mathcal{L}(\mathcal{H})$	language accepted by \mathcal{H}
$CL(\phi)$	closure of ϕ
$A_{\mathcal{H}}$	set of atoms for ϕ
$G_{\mathcal{H}}(\phi)$	tableau graph for ϕ and \mathcal{H}
AV_q	atomic proposition valuation of q
π	allocation path
$\text{Inf}(\pi)$	infinite set of π
SCS	strong connected subgraph
Par	set of compound statements
\mathcal{V}	semantics of boolean expressions
\perp	undefined value

Chapter 3

OID	universe of object identities
EVT	universe of events
MNAME	universe of class names
CNAME	universe of class names
VNAME	universe of variable names
MDECL	universe of method declarations
CDECL	universe of class declarations
VDECL	universe of variable declarations
TYPE	universe of types
VAL	universe of values
S_{BOTL}	BOTL static expressions
T_{BOTL}	BOTL temporal expressions
$Conf$	set of configurations of a BOTL model
ς	local state of a BOTL model
γ	valuation of formal parameter/return value in method invocations
η	path of configurations
$\{\cdot\}$	bag
\uplus	union of bags
δ	translation function for OCL expressions
Δ	translation function for OCL constraints

Chapter 4

λ	reallocation
\sqsubseteq^{fold}	fold relation between allocation triples or ABA/HABA languages
∞	black hole
$[q]$	bounded HABA state q
$\bar{[q]}$	unbounded HABA state q
$Exp(\mathcal{H})$	an expansion of \mathcal{H}
\mathcal{H}_d	duplication of \mathcal{H}
$\bar{\Theta}$	flattening of Θ
$K(\phi)$	upper bound on the number of variables for constructing $G_{\mathcal{H}}(\phi)$
$\Omega(f)$	black number of f

Chapter 5

Nav	set of navigation expressions
α	navigation expression
$\dots a^n$	abbreviation for $\underbrace{\dots a \dots a}_{n \text{ times}}$
μ	reference function
M	upper bound to the precision of cardinality functions
\mathbb{M}	$\{1, \dots, M\}$
$*$	unbounded cardinality value
\mathcal{C}_E	cardinality function on E
$\mathbf{1}_E$	unitary cardinality function on E
\oplus	bounded sum operation on cardinalities
$\sum_{e \in E}$	bounded sum over E
$\lceil n \rceil_M$	cut-off of n at M
E^*	set of unbounded entities in E
γ	configuration
CONF	set of all configurations
\prec	relation induced by μ
\preceq	reflexive closure of \prec
$indegree(e)$	number of references pointing to e
$first(E)$	first entity of the chain E
$last(E)$	last entity of the chain E
\succrightarrow	morphism
\Rightarrow	reallocation
id	identity morphism or reallocation

\circ	composition of morphism
\triangleright	concretion
\sqsubseteq	HABA simulation
nil	syntactic constant for null value
ϑ	standard interpretation for program variables
PV	set of entities modelling program variables
$\mu^*(e)$	remainder of the chain starting at e
$\langle \gamma \rangle_{PV}$	PV -reachable part of γ
$h \downarrow$	contractive morphism h
$h \downarrow^C$	contractive morphism h with shrink factor bounded by C
error	error state
$\text{SExp}(\gamma)$	safe expansion of γ
$\text{cf}(\gamma)$	canonical form of γ
L_p	bound on the longest navigation expression in p
$K(\phi)$	bound on stretching
$\mathcal{H} \uparrow \hat{M}$	stretching of \mathcal{H} up to \hat{M}
E^-, E^+	special set of logical variables for E
δ	distance function between interpretation of variables
Δ	distance function between navigation expressions
$[- \circ -]$	valuation reallocator

Chapter 6

\mathcal{N}	universe of ambient names
$\tilde{\mathcal{N}}$	universe of indexed names
Proc	universe of processes
\mathcal{J}	set of indexes
$n(P)$	names of P
$fn(P)$	free names of P
bnP	bound names of P
$noi(P)$	non-indexed version of P
$idx(P)$	set of indexes in P
\rightarrow	reduction relation
\rightarrow^*	reflexive and transitive closure of \rightarrow
\equiv	structural congruence relation
$\text{ho}(a)$	host entity of a
$\text{is}(a)$	inactive site entity of a
$\text{A}(e)$	ambient represented by e
$e:n$	shorthand for $\text{A}(e) = n$
E^{ho}	subset of (fixed) host entities in E
E^c	subset of ambient copies in E
E^m	subset of mobile entities in E

$E_n^{\text{is}}(P)$	set of inactive sites in P
$E_n^{\text{ho}}(P)$	set of hosts in P
$P(e)$	set of processes to be executed by e
@	outer-most ambient
$\ell^a(e)$	longest pure chain of a copies leading to e
\uplus	configuration union
$\rho(P)$	set of subprocesses to be executed by the ambient containing P
$\text{enab}(P)$	enabled ambients in P
$\varepsilon(a)$	empty (fixed) state for ambient a
Ω	encoding function for processes
\mathcal{D}	encoding function for processes
q_{pre}	pre-initial state
q_{in}	initial state
λ_{pre}	pre-initial reallocation
act	activation function
move	move function
dissolve	dissolve function
IOUp	Input/output update of the state
OpenUp	Open update of the state
$\text{siblings}(e)$	set of sibling ambients of e
$\text{parents}(b)$	set of parents of b
$\text{son}(a)$	set of son entities of a

Index

Symbols

L -compactness, 175
 L -compactness for MA, 226
 L -safety, 170
 L -safety for MA, 226
 Inf , 38
 PV -reachable, 169
 \mathcal{A} -satisfiability, 103, 163
 \mathcal{A} -validity, 103, 163
 \mathcal{H} -satisfiable, 103, 163
 \mathcal{H} -valid, 103, 163
 ∞ -reallocation, 94
F eventually operator, 29
G globally operator, 29
U until operator, 29
X next operator, 28
(Common Birth), 154
(Common Death), 154
(No-Cross), 155

A

accepted language
 Büchi automata, 26
 HABA, 98, 160
active ambient, 225, 229
allocation path
 $\mathcal{A}llTL$, 129
 $\mathcal{N}allTL$, 206
allocation sequence
 $\mathcal{A}llTL$, 86
 $\mathcal{N}allTL$, 144
Allocational Büchi Automaton, 93
 with references, 150

Allocational Temporal Logic, 85

atom, 36
 $\mathcal{A}llTL$, 121
 $\mathcal{N}allTL$, 200
atomic proposition valuations
 $\mathcal{A}llTL$, 119
 $\mathcal{N}allTL$, 199

B

Büchi automata, 26
 generalised, 27
Bandera Specification Language, 79
black hole, 94
 abstraction, 95
black number, 120
BOTL operational model, 59
 configuration, 59
bounded entities, 146

C

canonical form, 176
canonical form for MA, 226
cardinality function, 145
class, 42
closure, 36
 $\mathcal{A}llTL$, 120
 $\mathcal{N}allTL$, 200
common fate, 156
Computation Tree Logic (CTL), 30
concrete automaton \mathcal{A}_p
 $\mathcal{A}llTL$, 108
 with references, 171
configuration
 $\mathcal{N}allTL$, 146

- BOTL operational model, 59
- union, 227
- connected graph, 201
- D**
- distance, 193
- duplication, 116
- dynamic aliasing, 48
- E**
- enabled ambient, 229
- entity
 - concrete, 146
 - mobile, 239
 - multiple, 146
 - unbounded, 146
- event, 53
- F**
- flattening, 69
- folded allocation sequence
 - All*TL, 88
 - Nall*TL, 159
- fulfilling path
 - All*TL, 129
 - Nall*TL, 206
- G**
- garbage collection, 48
- generalised Büchi automaton, 27
- generator
 - All*TL, 96
 - Nall*TL, 160
- H**
- HABA emptiness problem, 209
- HABA expansion, 101
- HD-automaton, 40
- High-level ABA, 94
 - with references, 158
- host, 219
- I**
- identity morphism, 149
- identity of object, 42
- identity reallocation, 164
- impartiality, 154
- inactive
 - ambient, 225
 - site, 222
- indexed process, 221
- initial valuation, 158
- initial-state, 224
- invariants, 65
- isomorphism
 - All*TL, 89
 - Nall*TL, 150
- K**
- Kripke structure, 24
 - fair, 27
- L**
- leads-to operator, 143
- LFSA, 26
- Linear Temporal Logic (LTL), 28
- M**
- MA process encoding, 232
- maximal connected subgraph, 201
- metempsychosis metaphor, 91
- method occurrence, 53
- metric space, 195
- Mobile Ambients, 214
- mobile entities, 239
- model checker, 23
- model checking, 23
- model checking problem, 32
- modelling, 24
- morphism, 148
 - composition, 149
 - contractive, 170
 - identity, 149
- multiplicity, 152
- multiset, 152
- N**
- navigation expressions, 143
- non-resurrection condition, 87
- O**
- object, 42

- Object Constraint Language, 65
- Object Constraint Language (OCL)
 - constraints, 66
 - expressions, 66
 - iterate, 70
- P**
- parents, 240
- path, 25
- postconditions, 65
- Pre-initial state, 223
- preconditions, 65
- properties specification, 24
- pure chain, 148
- R**
- reallocation
 - AllTL*, 88
 - NallTL*, 153
- references, 47
- S**
- safe expansions, 177
- satisfiability problem, 31
- self-fulfilling SCS, 38
 - AllTL*, 131
 - NallTL*, 206
- shrink factor, 170
- siblings, 240
- simulation, 161
- son, 240
- stretching, 191
- strongly connected subgraph (SCS),
 - 38
- structural rules, 216
- symbolic automaton \mathcal{H}_p
 - AllTL*, 111
 - NallTL*, 182
- synchronous product of LFSA, 27
- T**
- tableau graph, 36
 - AllTL*, 127
 - NallTL*, 205
- U**
- UML, 43
- unitary cardinality function, 146
- V**
- validity problem, 32
- valuation
 - AllTL*, 118
 - NallTL*, 193
- valuation reallocator, 203
- verification, 24
- W**
- weakly connected graph, 201
- well-formed configuration, 169
- well-indexed, 221

Samenvatting

Dit proefschrift beschrijft een onderzoek op het gebied van *software verificatie*, met name naar automatische technieken voor het vaststellen van de correctheid van objectgebaseerde programma's. We kijken daarbij vooral naar de dynamische aspecten van dergelijke programma's en beschouwen verificatietechnieken gebaseerd op *model checking*. Het grootste probleem bij het ontwerpen van algoritmen voor model checking is de *explosie van de oneindige toestandsruimte* als gevolg van de dynamische constructies van objectgebaseerde programmeertalen. Enerzijds zorgen de onbegrensde *allocatie* en *deallocatie* (geboorte en sterfte) van zowel objecten als threads voor deze oneindige toestandsruimte. Anderzijds verwijzen objecten naar elkaar middels referenties (pointers) waardoor een 'heap' ontstaat – waar de objecten gealloceerd worden – waarvan de dynamische topologische structuur op een onvoorspelbare en complexe manier evolueert.

Ten einde de gesignaleerde problemen op te lossen, definieert dit proefschrift allereerst een temporele logica, gericht op de specificatie van een grote verscheidenheid van eigenschappen van objectgebaseerde systemen. Vervolgens worden twee belangrijke deelverzamelingen van deze logica onder de loep genomen. De eerste deelverzameling is een basislogica om te kunnen redeneren over allocatie en deallocatie van objecten. De tweede deelverzameling voegt daar de mogelijkheid aan toe om over dynamische referenties te redeneren. Deze temporele logica's worden geïnterpreteerd op geschikte, op (Büchi) automaten gebaseerde, modellen die gebruikt worden als eindige abstracties van systemen met een oneindige toestandsruimte. Dergelijke automaten worden ook gebruikt voor de definitie van de operationele semantiek van programmeertalen en vormen de basis van onze algoritmen voor model checking. Voor de basislogica voor allocatie en deallocatie van objecten presenteren we een correct en volledig algoritme. Voor de logica met dynamische referenties laten we een correct (maar niet volledig) algoritme zien. Dit laatste algoritme kan derhalve incorrecte tegenvoorbeelden opleveren.

Daarnaast laten we in dit proefschrift enkele voorbeelden zien hoe eindige automaten automatisch kunnen worden afgeleid uit een programma. Tenslotte demonstreren we het gebruik van de ontwikkelde theorie en algoritmen aan de

hand van een voorbeeld waarin veiligheidseigenschappen van 'Mobile Ambients' worden geverifieerd.

The author

Dino Distefano was born on July 20th, 1973, in Catania, Italy. He started studying computer science at the University of Pisa in 1992. He graduated in 1997 after an internship at the Free University of Amsterdam where he carried out the research for his master thesis on formal semantics of concurrent object-oriented languages. In 1999, he joined the Formal Methods and Tools group at the Department of Computer Science of the University of Twente working on the research described in this thesis.

Titles in the IPA Dissertation Series

- J.O. Blanco.** *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-01
- A.M. Geerling.** *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-02
- P.M. Achten.** *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-03
- M.G.A. Verhoeven.** *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-04
- M.H.G.K. Kessler.** *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-05
- D. Alstein.** *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-06
- J.H. Hoepman.** *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-07
- H. Doornbos.** *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-08
- D. Turi.** *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-09
- A.M.G. Peeters.** *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10
- N.W.A. Arends.** *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11
- P. Severi de Santiago.** *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12
- D.R. Dams.** *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13
- M.M. Bonsangue.** *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14
- B.L.E. de Fluiter.** *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01
- W.T.M. Kars.** *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02
- P.F. Hoogendijk.** *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03
- T.D.L. Laan.** *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04
- C.J. Bloo.** *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05
- J.J. Vereijken.** *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06
- F.A.M. van den Beuken.** *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07
- A.W. Heerink.** *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01
- G. Naumoski and W. Alberts.** *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02
- J. Verriet.** *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03
- J.S.H. van Gageldonk.** *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04
- A.A. Basten.** *In Terms of Nets: System Design with Petri Nets and Process Algebra.*

- Faculty of Mathematics and Computing Science, TUE. 1998-05
- E. Voermans.** *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01
- H. ter Doest.** *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02
- J.P.L. Segers.** *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03
- C.H.M. van Kemenade.** *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, UL. 1999-04
- E.I. Barakova.** *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05
- M.P. Bodlaender.** *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06
- M.A. Reniers.** *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07
- J.P. Warners.** *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08
- J.M.T. Romijn.** *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09
- P.R. D'Argenio.** *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10
- G. Fábíán.** *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11
- J. Zwanenburg.** *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12
- R.S. Venema.** *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13
- J. Saraiva.** *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14
- R. Schiefer.** *Viper, A Visualisation Tool for Parallel Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15
- K.M.M. de Leeuw.** *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01
- T.E.J. Vos.** *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02
- W. Mallon.** *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03
- W.O.D. Griffioen.** *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04
- P.H.F.M. Verhoeven.** *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05
- J. Fey.** *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06
- M. Franssen.** *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07
- P.A. Olivier.** *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08
- E. Saaman.** *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10
- M. Jelasity.** *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01
- R. Ahn.** *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

- M. Huisman.** *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03
- I.M.M.J. Reymen.** *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04
- S.C.C. Blom.** *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05
- R. van Liere.** *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06
- A.G. Engels.** *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07
- J. Hage.** *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08
- M.H. Lamers.** *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09
- T.C. Ruys.** *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10
- D. Chkhaev.** *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11
- M.D. Oostdijk.** *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12
- A.T. Hofkamp.** *Reactive machine control: A simulation approach using χ .* Faculty of Mechanical Engineering, TU/e. 2001-13
- D. Bošnački.** *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14
- M.C. van Wezel.** *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01
- V. Bos and J.J.T. Kleijn.** *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02
- T. Kuipers.** *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03
- S.P. Luttik.** *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04
- R.J. Willemsen.** *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05
- M.I.A. Stoelinga.** *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06
- N. van Vugt.** *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07
- A. Fehnker.** *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08
- R. van Stee.** *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09
- D. Tauritz.** *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10
- M.B. van der Zwaag.** *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11
- J.I. den Hartog.** *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12
- L. Moonen.** *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13
- J.I. van Hemert.** *Applying Evolutionary Computation to Constraint Satisfaction and*

Data Mining. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μ CRL.* Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedeia. *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On model checking the dynamics of object-based software: a foundational approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09