

An Executable Theory of Multi-Agent Systems Refinement

Lăcrămioara Aștefănoaei

Promotion committee:

Promotores:	Prof. dr. F.S. de Boer	Leiden University
	Prof. dr. J.J.Ch. Meyer	University of Utrecht
Co-promotor:	dr. M. Dastani	University of Utrecht
Other members:	Prof. dr. F. Arbab	Leiden University
	dr. K. Hindriks	Delft University of Technology
	Prof. dr. W. van der Hoek	University of Liverpool, UK
	Prof. dr. J.N. Kok	Leiden University

Copyright © 2011, Lăcrămioara Aștefănoaei, Amsterdam. All rights reserved.

ISBN 978-90-6464-450-4

IPA dissertation series 2011-4

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

Printed by Ponsen & Looijen

Cover designed by Lăcrămioara Aștefănoaei with the help of GIMP (gimp.org) and Harmony (mrdoob.com/projects/harmony).

The four images on the cover are from the film “Le Mystère Picasso” directed by Henri Clouzot. They depict instances of the making of a painting. There is no intention of explaining creativity by means of refinement.



The work reported in this dissertation has been carried out at the *Centrum voor Wiskunde en Informatica* (CWI) in Amsterdam, under the auspices of the *Instituut voor Programmatuurkunde en Algoritmiek* (IPA). The research has been supported by the CoCoMas (Coordination and Composition in Multi-Agent Systems) project funded by *Nederlandse Organisatie voor Wetenschappelijk Onderzoek* (NWO).

An Executable Theory of Multi-Agent Systems Refinement

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus prof.mr.P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdedigen op woensdag 19 januari 2011
klokke 13:45 uur

door

Lăcrămioara Aștefănoaei
geboren te Iași, Romania
in 1982

“Il faut confronter les idées vagues avec des images claires.”
(La Chinoise, Jean-Luc Godard)

Contents

Acknowledgments	v
1 Introduction	1
1.1 Thesis Contributions and Structure	3
1.1.1 Part I: “Refinement of Single Agents”	3
1.1.2 Part II: “Refinement of Multi-Agent Systems”	4
1.1.3 Part III: “Implementation”	7
I Refinement of Single Agents	9
2 From Agent Specification to Implementation	11
2.1 Formalising Basic Concepts of Agent Languages	12
2.1.1 Mental States	13
2.1.2 Actions	14
2.1.3 Queries	15
2.2 A Specification Agent Language: BUnity	18
2.2.1 Syntax	18
2.2.2 Operational Semantics	19
2.2.3 Fair Executions of BUnity Agents	20
2.3 From BUnity to BUpL: Refining Control	22
2.3.1 Syntax	22
2.3.2 Operational Semantics	23
2.3.3 Fair Executions of BUpL Agents	24
2.4 From BUnity to BUnity ^K : Refining Data	26
2.5 An Implicit Modelling of Goals	29
2.6 Prototyping BUpL as Rewrite Theories	30
2.6.1 Executing Agents by Rewriting	35
2.6.2 A Taste of Rewriting as a Metalanguage	36
2.6.3 A Strategy Language	39

2.6.4	Controlling Executions with Strategies	41
2.7	A Theory of Agent Refinement	42
2.7.1	Control Refinement	42
3	Verification Techniques	47
3.1	Model-Checking Control Refinement	48
3.2	Undecidability results	50
3.3	A Weakest Precondition Calculus for BUnity	55
3.3.1	Assertions	56
3.3.2	Action Correctness	56
3.3.3	The Predicate Transformer wp	57
3.3.4	Invariants	60
3.3.5	Leads-to Properties	61
3.4	Testing BUpL Using Strategies	66
3.4.1	Methodology	66
3.4.2	Formalising Test Cases	68
3.4.3	Using Rewrite Strategies to Define Test Drivers	69
II	Refinement of Multi-Agent Systems	75
4	From Agents to Multi-Agent Systems	77
4.1	Classifying Coordination	77
4.2	Action-Based Coordination	79
4.2.1	Choreographies	79
4.3	Timed Choreographies	82
4.3.1	Timed BUnity	84
4.3.2	Timed BUpL	85
4.3.3	Timed Multi-Agent Systems	86
4.4	A Normative Language	87
4.4.1	Syntax	88
4.4.2	Operational Semantics	91
4.4.3	Normative Properties	93
4.4.4	From Totalism to Liberalism in Operational Semantics	93
4.4.5	A Short Note on Computing Closures	96
4.5	Timed Normative Artifacts	97
4.6	Executable Timed Choreographed Normative Systems	100
4.6.1	Combining Timed Choreographies and Norms	101
4.6.2	Execution by Real-Time Rewriting	101
4.7	Multi-Agents Systems Refinement	109
4.7.1	A Finer Notion of Refinement	110
4.7.2	A Short Note on TCNMAS Refinement	114

III	Implementation	117
5	Maude at Practise	119
5.1	Prototyping	119
5.1.1	Introduction to Maude	120
5.1.2	Implementing BUpL: Syntax	122
5.1.3	Example BUpL Program	125
5.1.4	Implementing BUpL: Semantics	126
5.1.5	Executing an Agent Program	129
5.2	Model-Checking	130
5.2.1	Connecting BUpL Agents and Model-Checker	130
5.2.2	Examples	132
5.2.3	Fairness	134
5.3	Testing	136
5.3.1	Searching	137
5.3.2	Rewrite Strategies in Maude	138
5.3.3	Using Maude Strategies for Implementing Test Cases	139
5.4	Executable Normative Multi-Agent Systems	141
6	Coordinating 2APL with Reo Artifacts	145
6.1	A Short Overview of Reo	145
6.2	A Short Overview of 2APL	147
6.3	Integrating Reo Connectors into the 2APL platform	150
7	Conclusions and Perspectives	157
	Bibliography	161
	Samenvatting	173
	Abstract	175

Acknowledgements

First and most importantly I want to thank Frank de Boer. He guided me with patience and humour during the last four years, generously sharing his ideas, while also giving me freedom to choose for myself. I thank him for his gentle encouragements and for reminding me where the focus was whenever i was off-topic for too long. I owe this thesis mainly to him. Secondly, I want to thank Mehdi Dastani, who has always answered my questions promptly and objectively. I thank him for being enthusiastic about our research and for his vivid discussions, scientific and not only. Thirdly, i want to thank John-Jules Meyer for carefully reading the thesis, for his adequate suggestions and for wisely anticipating possible criticisms in a positive manner. Further, I thank Farhad Arbab, Koen Hindriks, Wiebe van der Hoek, and Joost Kok for willing to take place in the thesis committee. I thank Farhad Arbab and Jan Rutten for their contribution in the process of hiring me. I thank the Centre for Mathematics and Computer Science (CWI) for being a research institute with splendid people, a most beautiful library, and a stimulating atmosphere for young minds.

I also thank my co-authors, and especially Birna van Riemsdijk for her instructive feedback. I thank Nick Tinnemeier for his part in the CoCoMas project. Many thanks to Vlad Rusu for making our collaboration possible and for the pleasant and useful discussions we had. I want to thank Herman Geuvers, Koen Hindriks, and the Rascal-ists for thorough explanations and interesting research ideas. I thank the organisers and the lecturers of Marktoberdorf and rewriting summer-schools which covered a great amount of exciting topics. I owe a many of thanks to the people that offered me scientific guidance prior to my PhD studies, indirectly shaping possible research trajectories: Valerica Bența, Gabriel Ciobanu, Liviu Ciortuz, Cornel Croitoru, Dorel Lucanu, Vlad Rădulescu, Laurent Dupont, and Sylvain Petitjean. They have been models and sources of inspiration for me as a student.

A special thanks to the following people: Irina Brudaru, Svetlana Dubinkina, Mohammad Mahdi Jaghoori, Sung-Shik Jongmans, Alexandra Silva, Yanjing Wang, Eugen Zălinescu for their practical help. A double special thanks to Eugen Zălinescu for his relevant suggestions and explanations. I warmly thank Ronald de Wolf for his inexhaustible source of bits of sage thoughts. I thank the whole SEN3 for the joyful, creative and dynamic atmosphere.

Finally, i thank Filmmuseum et al. (avec leur frère dans la nouvelle vague Sergiu Bursuc), friends and family for all that harmoniously complemented the thesis. I am most grateful to mother and father for their unconditioned effort to provide me with the proper conditions to focus on my studies.

Lăcrămioara Aștefănoaei
December 2010

Abstraction may well be an indispensable element in any attempt to formalise real systems. As N. Dershowitz and Y. Gurevich phrased it ([DG08]), “science in general is impossible without model[li]ng.” To stress this point, it might be adequate to retrospectively think of the following lines which belong to A.M. Turing ([Tur54]): “If one wants to treat the problem seriously and systematically one has to replace the physical puzzle by its mathematical equivalent”. In the context of this thesis, the puzzles are to be understood as complex applications such as incident management, social simulations, manufacturing applications, electronic auctions, e-institutions, and business to business applications. Their mathematical equivalent is obtained by adapting an advance in abstraction which is the *agent-oriented methodology*.

To illustrate key concepts in the agent-oriented paradigm we consider the typical example of incident management. In incident management, different *organisations* are involved, such as police, fire, and ambulance departments, in order to handle calamity situations. Each of these organisations can in turn consist in coordinating other organisations or *individual agents*. For example, a police department consists in coordinating subdepartments such as the detective and the administration department, or individual agents such as police officers. At the level of individual agents, a police officer, for example, needs to coordinate different *actions* such as street patrolling and house searching to find criminal evidence. The behaviours of the organisations are *coordinated* by means of *social/normative* structures, which are themselves defined in terms of a variety of social concepts and relations like norm, trust, power, delegation of tasks, responsibilities, access to resources, and communication protocols. In what follows, we shortly describe the key concepts and we highlight, whenever the case, the problems that might arise and which we dealt with.

Agents An agent is commonly seen as an encapsulated computer system that is situated in some environment and that is capable of flexible, *autonomous action* in that environment in order to meet its design objectives [Woo97]. An important line of research in the agent systems field is the design of agent languages where the guiding idea is that agent-specific concepts such as beliefs (representing the environment and possibly other data the agent has to store), goals (representing desired states), and plans (specifying which sequences of actions, eventually compositions of other plans, to execute in order to reach the goals) facilitate the programming of agents. Much of the underlying theory suggesting this particular view is

based on the work on *rational agency*, the so-called Belief Desire Intention (BDI) paradigm [Bra87]. Following this paradigm, several agent programming languages have been developed with an emphasis on the use of *formal methods*. In particular, *operational semantics* [Plo81] is often used for formally defining the semantics of languages. One feature which is advantageous to exploit is that operational semantics is easily prototyped. This facilitates, on the one hand, building an interpreter for the languages, and on the other hand, verification.

Agents and Coordination A multi-agent system (MAS) can be roughly seen as a collection of interacting agents. One of the challenges in the design and development of multi-agent systems is to basically define and implement “interaction”. In this respect, two approaches which aim at achieving interaction are communication and coordination. Communication, in turn, can be implemented by message passing. Or it can be implemented by channel-based mechanisms. This latter can be seen as the basis of implementing coordination artifacts. Such artifacts are usually built in terms of resource access relation in [RVO07], or in terms of action synchronisation [AAdB⁺08]. Another reference worth mentioning is the language Linda [GZ97]. The proposed coordination paradigm has not yet been applied in a multi-agent setting but to service oriented services, where the notion of *data* plays a more important role than *synchrony*.

Agents and Norms The design of programming languages that support the implementation of normative systems is an important issue. However, it is still conceptually problematic. Norms are essential for artificial agents that are to display behaviour comparable to human intelligent behaviour or collaborate with humans. Norms play a central role in many social phenomena such as coordination, cooperation, decision-making, etc. There is an increasing interest in the role of norms in societies, both inside and outside the agent community. Normative multi-agent systems combine theories and frameworks for normative systems with multi-agent systems. Thus, these systems provide a promising model for human and artificial agent coordination, because they integrate norms and individual intelligence. They are a prime example of the use of sociological theories in multi-agent systems, and therefore of the relation between agent theory and the social sciences, e.g., sociology, philosophy, economics, legal science, etc. Regarding penalties, rewards, regulated norms, deontic logic aspects, the interested reader is invited to check [BvdTV07, BvdT08, BBvdT08, GR08a, GR08b] for comprehensive discussions. For a game-theoretic approach we refer to the works from [BvdT07, GGvdT08].

Verification To ensure that the developed multi-agent systems achieve their overall design objectives and satisfy some global desirable properties, one has to verify both agents and the organisation artifacts that constitute the coordination and control part of the multi-agent systems. With respect to verification, some results are as follows: [BFVW06] discusses model-checking AgentSpeak systems, [BJvdM⁺06] proposes Temporal Trace Language for analysing dynamics between agents, [RL07] refers to verifying deontic interpreted systems.

1.1 Thesis Contributions and Structure

The main contribution of this thesis consists of introducing an executable theory for the refinement of multi-agents systems. The effort of introducing the formalism of multi-agent system refinement is motivated by the need to perform verification. Multi-agent systems are more complex structures, and their verification tends to become harder. The refinement relation we aim at is such that the verification of a concrete MAS system reduces to the verification of the corresponding abstract system.

Technically, the construction of the proposed theory makes use of and integrates different concepts and results from process theory [vG90], rewriting logic [BM03], and the theory of timed automata [Alu99]. These concepts will be explained in the sections where they will appear.

The thesis is organised in three parts. The first part develops a theory of agent refinement. This is extended in the second part to systems of agents where the notion of coordination plays the main role. Implementation issues are discussed in the last part.

1.1.1 Part I: “Refinement of Single Agents”

Part I consists of Chapter 2 and 3 and develops an executable theory of agent refinement.

Chapter 2 describes a top-down methodology of agent design by introducing simple but expressive agent languages, BUnity, BUPL and BUnity^K. These languages are inspired by the already standard GOAL [dBHvdHM07] and 3APL [HdBvdHM99] languages. It is important to stress from the beginning that the focus is on *modelling* and not programming agent languages. The semantics of the languages being introduced is operational. This makes it easy to *prototype* the agent languages as *rewrite theories*. The motto which should become transparent throughout the chapter advocates prototyping before implementing (complex agent platforms) because prototyping is a quick method for proving that the language definitions fulfill the initial requirements or have desired properties. The choice to prototype in a rewrite-based framework is motivated by the fact that rewriting logic (RL) is *executable*. More precisely, RL has the advantage that not only is prototyping a straightforward process but it further makes it possible to *execute* the prototypes since “computation in rewriting logic is execution”.

The proposed languages represent different levels of abstraction with respect to data and control. They are meant to illustrate refinement relations which are a natural choice when defining *correctness* properties between agents at different specification levels. By correctness it is meant the following statement: “a given implementation is correct with respect to a given specification when the implementation refines the specification”. Though there are some general ideas discussed about *data refinement* between BUnity and BUnity^K agents, the main focus is on *control refinement* between BUnity and BUPL agents. More precisely, in this latter case, BUnity is considered as a specification and BUPL as an implementation agent language. BUPL refines the nondeterminism in the choice of actions inherently present in BUnity. Control refinement is defined as trace inclusion, namely a BUPL agent is correct with respect to a BUnity specification if any possible BUPL behaviour (trace) is also a BUnity one. Control refinement can be encoded in a natural manner as a simulation relation. This means that the problem of deciding refinement can be reduced to a model-checking problem.

The main result is a proof-technique for refinement which is extended such that it is suitable for the analysis of agents with infinite behaviours. Since some of these behaviours are unlikely to occur in practice, we provide a declarative approach to modelling fairness as formulae in linear temporal logic (LTL) such that refinement under fairness conditions reduces again to a model-checking problem.

Chapter 3 focuses on verification techniques. One of these is model-checking, as mentioned above. The notable difference with respect to related works (on model-checking agents) is that here the focus is on refinement, not on arbitrary properties. It is known that model-checking works well for finite systems, i.e., in an agent-based framework, for agents with a finite, better said small, number of configurations (observe that this does not imply that the behaviours are finite). For infinite state agents, however, model-checking is a semi-decision procedure. Since the languages dealt with are, as one might expect, Turing complete, a general solution for reachability problems is not possible. Thus one needs to look for other approaches which would turn adequate to reason about agent programs. One approach is to define, in the style of [Dij76] a *weakest precondition calculus* for abstract agent languages, for instance BUnity, to allow symbolic reasoning about various properties of agent programs. It is shown that refinement preserves certain properties and this result brings the benefit that the weakest precondition calculus can be used as an indirect way of reasoning about the properties of more concrete agent programs, as BUpL agents. For the reference, there is a variety of weakest precondition calculi designed for imperative and logical languages [CN00, Jac02, PR98]. A weakest precondition calculus for agent languages as proposed in this thesis is new.

Another technique being considered in Chapter 3 is to *test* BUpL agents. It is explained how to define test cases and how to implement them by means of rewriting strategies. In the literature, the very basic idea behind testing is that it aims at showing that the intended and the actual behaviour of a system differ by generating and checking individual executions. Testing object-oriented software has been extensively researched and there are many pointers in the literature with respect to manual and automated, partition and random testing, test case generation, criteria for test selection. A broad overview can be found in [Mey08]. In an agent-oriented setup, there are less references. A few pointers are [ZTP08, NPT07] for developing test units from different agent methodologies, however the direction is orthogonal to the one we consider.

The material from this first part of the thesis is published in [AdB08, AdBvR09]. The work on refinement appears also in an invited chapter [AB10].

1.1.2 Part II: “Refinement of Multi-Agent Systems”

Part II consists of Chapter 4. It aims at showing how the refinement methodology from Part I can be extended at a multi-agent level. It begins by proposing different coordination mechanisms and by discussing their integration. The essential difference between the proposed coordination mechanisms is with respect to the dichotomy “action/state”: action-based coordination artifacts are meant to force the synchronisation of action execution, while normative

artifacts¹ are meant to enforce the system to be in a non-violating state.

Action-based coordination is represented by choreographies. Choreographies are seen as protocols describing when either orderings or synchronisations of the actions executed by the agents. The concept of choreography and the somehow related concept of orchestration have already been introduced to web services (to the paradigm Service-oriented Computing), see [MA07, BBM⁺07, BBC⁺09, Mis04] for different approaches. Though in this thesis the same terminology of *choreography* is being used, the understanding of this notion is essentially different from the one in the related work since we deliberately ignore communication issues. The choreography model being defined in Chapter 4 is explicit whereas in the other works choreography is implicit in the communication protocol. One consequence is that additional care needs to be taken with regards to deadlock situations that may appear because of “mall-formed” choreographies. Being external, the choreography represents, in fact, contexts while in the other approaches there is a distinction between the modularity and the contextuality of the communication operator.

To have a more expressive framework, it is proposed an extension of choreographies by incorporating a notion of time as a real value. The extension allows, for example, to impose timing constraints on action execution, or to force independent multiple delays between actions. Furthermore, the extension is such that there is a clear “separation of concerns”: actions, as the main building block in timed multi-agent systems, are *untimed*, while time constraints are *application-specific*. Actions have a natural definition as belief base transformers, thus having an untimed ontology of actions allows reusability. The main source of inspiration for timed choreographies is the theory of timed automata. Timed automata has been applied to testing real-time systems specifications [HLM⁺08], to scheduling problems [BBL08], and to web-services [LPSS09]. The use of timed automata in a normative multi-agent setting is new. One of the advantages of using timed automata is the fact that there exists a suite of verification tools like UPPAAL [BDL⁺06], KRONOS [BDM⁺98]. There is also Real-Time Maude [ÖM08] which is used as a natural extension to prototype timed-multi-agent systems as real-time rewrite theories.

The work on normative artifacts has as a starting point a programming language which was first introduced in [DGMT08] and further extended in [TDM09]. The normative language was designed to facilitate the implementation of norm-based organisation artifacts. Such artifacts refer to norms as a way to signal when violations take place and sanctions as a way to respond (by means of punishments) in the case of violations. Basically, a norm-based artifact observes the actions performed by the individual agents, determines their effects in the environment (which is shared by all individual agents), determines the violations caused by performing the actions, and possibly, imposes sanctions. The literature on normative concepts is rich and a few pointers are as follows. The early works in [Dig03, FGM03] focus on social and organisational concepts like norms, roles, groups, responsibility, and mechanisms like monitoring agents’ actions and sanctioning in order to organise and control the behaviour of individual agents. Other approaches propose organisation-based coordination artifacts, i.e., coordination artifacts that are designed and developed in terms of social and organisational concepts [DGMT08, TDM08, BvdT08, ERRAA04, HSB02]. The purpose in this thesis is

¹For simplicity we refer to artifacts based on counts-as and sanction rules as normative artifacts.

not to provide more sophisticated normative constructions (i.e., penalties, rewards, regulated norms, deontic logic aspects). Rather, the normative language in [DGMT08] is being used as a basic framework which provides the minimal necessary constructions to enable the analysis of possible operational semantics which can derive from different scheduling strategies for the application of norms. Each such semantics characterises a specific type of normative system. For instance, in the extreme case of an “autocratic agent society” each action an agent performs is followed by an inspection of the normative rules which might be applicable. At the other extreme, in a “most liberal society” the monitoring mechanism runs as a separate thread, independent of the executions of the agents. Furthermore, while in autocratic societies certain correctness (in terms of safety) properties are modelled by definition, this is no longer the case in liberal societies with infinite executions. This implies that it is necessary to consider additional fairness constraints in order to ensure the well-behaviour of the systems. Such technicalities are discussed in more detail in Chapter 4.

The only extension proposed for the normative language is along the same lines as for choreographies, that is, a timed extension realised by means of timed automata. The main benefit is that it makes it possible to time constrain the application of norms, for example, to signal new violations if certain deadlines have passed, or to cancel sanctions when certain expiration dates are met. This particular approach is orthogonal on other existing works which focus on temporal aspects of normative structures like the ones from [ASP09, VFC05]. The difference lies in the understanding of the notion of “time” in the sense that the clocks, as real-time variables, make time explicit while the related approaches deal with time implicitly.

Thinking of both action and state coordination mechanisms, these are considered as being orthogonal, each with its own expressiveness power and specific use. Thus, in this view, it makes little sense to imagine choreographies which play the role of normative rules and vice-versa. This, however, does not mean that both approaches cannot be integrated in the same framework. On the contrary, it is thought as being convenient to have different coordination mechanisms as alternatives provided in a general framework. This is better advocated in the last part of Chapter 4 where it is also shown, more importantly, that the whole resulting models, called timed choreographed normative systems, are executable.

The last part of Chapter 4 generalises the theory of agent refinement to multi-agent systems where coordination is achieved by means of timed choreographies and norms. The generalisation is such that coordination artifacts do not introduce deadlocks when plugged in concrete multi-agent systems when they do not cause deadlocks in the abstract systems. This means that the extension of simulation as a proof-technique for multi-agent systems must be treated with care. More precisely, it is needed to substitute simulation by a finer notion, the so-called *ready simulation*. The next step is to think of agents as elements in an “algebra” of multi-agent systems where coordination artifacts are operators, acting as contexts precisely in the following way. Given a coordination artifact c as a context with a hole $c[_]$ and two agents in a refinement relation, $Impl \subseteq Spec$, a multi-agent refinement relation \subseteq_{MAS} is defined such that $c[Impl] \subseteq_{MAS} c[Spec]$, that is after filling the hole with $Impl$, resp. $Spec$, refinement is preserved. In other words, multi-agent refinement is a preorder relation compatible with the coordination artifact or simply put, a *precongruence*. Since precongruence is closely related to compositionality, one advantage of the proposed approach is that refinement at the individual level implies refinement at the multi-agent system level. Compositionality is an

important property since it reduces heavily the verification process: knowing that a property P holds for $Spec$, $Spec \models P$, or equally that $c[Spec] \models P$ and having that $Impl$ refines $Spec$ one can deduce that P holds also for $c[Impl]$. Thus having a theory of refinement is a crucial step towards modular verification of multi-agent systems' correctness. The material in this part of the thesis is published in [AdBD09, ADMdB09, AdBD10].

1.1.3 Part III: "Implementation"

The last part of the thesis is divided in Chapter 5 and 6. The motivation is to provide an illustration that the formalism described in the first two parts is implementable.

Chapter 5 presents a natural and intuitive encoding as executable rewrite theories in Maude, a rewriting logic software. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Namely, being a rewrite-based framework, it makes it is easy to prototype modelling languages with an operational semantics by means of rewrite theories [MOM00a], and it provides mechanisms for verifying programs and language definitions by means of LTL model-checking [EMS02]. Furthermore, the inherent reflective feature of rewriting logic (and of Maude, in particular) offers an alternative to model-checking by means of rewrite strategies. With respect to related work, it is worth mentioning that Maude has already been used for prototyping executable semantics, and the work presented in [SRM09] is an extensive survey. As for agent-oriented languages, the initiative of modelling agents (a propositional variant of 3APL [HdBvdHM99]) in Maude is taken in [vRdBDM06] and, in fact, this reference provided the basis of the current implementation.

Chapter 6 provides a more practical experiment by illustrating the way low-level coordination effectively works within an existing agent platform called 2APL [Das08]. Specifically, the coordination mechanism in question is realised by means of channel-based connectors designed in the language Reo [Arb04, BSAR06]. Besides Reo, there are several kinds of channel-based communication models, MoCha [GSAdBB05], Nomadic Pict [WS00], to name a few. The middleware MoCha, for example, has the advantage of being a real infrastructure for distributed implementations. The choice for Reo is motivated by the fact that in Reo one can use channels not only for communication, but also for building complex connectors by composing channels. Such connectors impose specific coordination patterns on the execution of individual agents. The idea of using Reo as a coordination language for agents is not new, it appears first in [DAdB05]. However, what is provided in this thesis is an executable 2APL platform where it is possible to integrate Reo connectors as the underlying communication infrastructure and as coordination artifacts in 2APL systems. It is also shown that the benefits of the integration are in the wreath of existing tools, "The Eclipse Coordination Tools"², which facilitates the verification of interaction and communication properties.

The material in this last part of the thesis appears in [RAB10, AAdB⁺08].

²The Eclipse Coordination Tools are at <http://homepages.cwi.nl/~koehler/ect/>

Part I

Refinement of Single Agents

Chapter 2

From Agent Specification to Implementation

In this chapter we first propose a formalisation of what we think that are main concepts in agents languages, namely, mental states, actions and queries. We then describe a top-down methodology for the design of agent languages. Our source of inspiration is the classical design methodology UNITY [CM88] which emphasises the principles:

- “specify little in early stages of design” and
- “program design at early stages should not be based on considerations of control flow”.

Following the above design guidelines, we present a general modelling framework where we identify two abstraction levels of BDI agents. On the one hand, at a higher level of abstraction we introduce the language BUnity as a way to specify “what” an agent can execute. On the other hand, at a more concrete layer we introduce the language BUpL (**B**elief **U**ppdate **p**rogramming **L**anguage) as implementing not only what an agent can do but also “how” the agent executes. The agent specification language BUnity is meant to represent an agent in the first stage of design. One only needs to specify initial beliefs and actions (*what* an agent can do). The behaviour of a BUnity agent is given by the set of all possible infinite executions. These executions are highly nondeterministic because actions may be executed in any arbitrary order. Since BUnity abstracts away from scheduling policies we have that indeed BUnity “specifies little”. The agent implementation language BUpL is meant to represent an agent in the last stage of design. BUpL enriches BUnity constructions with the notions of *plans* and *repair rules*. These refine the early stage nondeterminism by specifying *how* and *when* actions are executed. To illustrate another (orthogonal) refinement we introduce BUnity^K where the accent is on data rather than control.

The semantics of all agent languages we design is operational, given as usually by means of transition systems. Transition systems have a natural encoding as rewrite systems and this implies that we can easily prototype BUnity and BUpL as rewrite theories. The main advantage of doing so is that we obtain executable semantics for free, meaning that we can execute agent programs by rewriting the corresponding rewrite system. Furthermore, since rewriting logic is reflective, we can control the agents’ executions by means of strategies. Strategies work at the meta-level and thus there is a clear separation between the definition of the semantics and control. This is what we will show in Section 2.6.

2.1 Formalising Basic Concepts of Agent Languages

Agent languages have an underlying logical language which provides the necessary mechanisms for agents to reason. In this thesis the underlying logical language we consider is first-order. For the ease of reference we denote it by L . We recall that a first-order language L consists of a vocabulary and the set of all formulae defined over it. The vocabulary consists of logical and non-logical symbols. The logical symbols are the usual connectives, quantifiers and variables. The non-logical symbols are functions and relations. Functions and relations have a fixed arity, i.e., number of arguments. 0-arity function symbols are called constants. We assume the set of constant symbols of L is non-empty. We note that the logical symbols are fixed a priori while the non-logical ones vary depending on the agent languages and thus they are the ones *identifying* and distinguishing them. The non-logical symbols are specified in a *signature* denoted by Σ_L .

Given a signature Σ_L there are two classes of strings that can be defined over Σ_L . The first class represents *terms*. Terms are built on a vocabulary of function symbols and variable symbols. They have an inductive definition: (1) any variable and constant are terms; (2) if f is a function symbol of arity n and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term. Sorted terms are built from a sorted signature, i.e., an extended signature which contains also a set of *sorts*. We use sorts because they are suitable for handling partiality, polymorphism and errors. We will explain them in more detail when needed. For the rest of the thesis we make the implicit assumption that agent languages are sorted.

A special set of terms we will work with is the set of *ground* terms. A term is *ground* if and only if it does not have any variables. The set of ground terms over a signature Σ_L is denoted by \mathcal{U}_L , the *Herbrand universe* of L ¹.

Two standard operations on terms are matching and unification. Their definition depends on the notion of substitution. A substitution is a mapping from variables to terms. Given a substitution θ which maps a variable x to a term t we abbreviate $\theta(x) = t$ as $\theta = [x/t]$. The application of a substitution $\theta = [x/t]$ on a term t' , in symbols $t'\theta$, consists of replacing every appearance of x in t' by t . A term s matches a ground term t , $s \leq^? t$, if there exists a substitution (called *matcher*) such that $s\theta$ is syntactically equal to t , $s\theta = t$. Two terms s, t unify, $s =^? t$, if there is a substitution (called *unifier*) such that $s\theta$ is syntactically equal to $t\theta$, $s\theta = t\theta$.

The second class represents *atoms*. Atoms are built on predicate symbols and terms, that is, if P is a relation symbol of arity n and t_1, \dots, t_n are terms, then $P(t_1, \dots, t_n)$ is an atom. An atom is sorted (respectively ground) when so are the terms defining it. The set of ground atoms over a signature Σ_L is denoted by \mathcal{B}_L , the *Herbrand base* of L .

2.1.1. EXAMPLE. Given a language L with a constant c , a function f and a relation P , the Herbrand universe \mathcal{U}_L is the infinite set $\{f^i(c) \mid i \in \mathbb{N}\}$ and the Herbrand base \mathcal{B}_L is the infinite set $\{P(f^i(c)) \mid i \in \mathbb{N}\}$. ♠

Formulae are constructed inductively using the logical connectors in the language L . The application of substitutions to formulae is similar to the one on terms. An expression is either

¹In an algebraic setting the Herbrand universe corresponds to the notion of initial algebra.

a term or a formula. We use $Var(e)$ to denote the variables from an expression e . If e is a formula, $Var(e)$ denotes the free variables of e .

When the semantics of logical formulae in the language L is defined with respect to Herbrand interpretations it is called *Herbrand semantics*. This is detailed next following closely the presentation from [Apt88]. We first recall the standard Tarskian notion of interpretation. Given a first-order language L , an interpretation \mathcal{I} for L consists of (1) a non-empty domain, D , (2) an assignment for each function f of arity n in L of a mapping $f^{\mathcal{I}}$ from D^n to D (constants c are assigned elements of D , $c^{\mathcal{I}}$), (3) an assignment for each relation P of arity n in L of an n -ary predicate $P^{\mathcal{I}}$ on D , i.e., a subset of D^n . Herbrand interpretations are obtained by fixing the universe as being the Herbrand universe. A Herbrand interpretation is uniquely determined by a subset of the Herbrand base which fixes the assignment of relations. This allows us to refer to Herbrand interpretations as subsets of \mathfrak{B}_L .

Given a signature Σ_L , with its associated Herbrand universe \mathfrak{U}_L and Herbrand base \mathfrak{B}_L , and a Herbrand interpretation \mathcal{H} in \mathfrak{B}_L , Herbrand semantics of a closed formula is defined inductively on the structure of the formula.

$$\begin{array}{ll} \models_{\mathcal{H}} P(\bar{t}) & \text{iff } P(\bar{t}) \in \mathcal{H} \\ \models_{\mathcal{H}} \neg \psi & \text{iff } \not\models_{\mathcal{H}} \psi \\ \models_{\mathcal{H}} \psi_1 \vee \psi_2 & \text{iff } \models_{\mathcal{H}} \psi_1 \text{ or } \models_{\mathcal{H}} \psi_2 \\ \models_{\mathcal{H}} \exists \bar{x} \psi & \text{iff } \models_{\mathcal{H}} \psi[\bar{x}/\bar{t}] \text{ for some } \bar{t} \in \mathfrak{U}_L \end{array}$$

where the vector notation \bar{x} represents the set of free variables of ψ , i.e., $\bar{x} = Var(\psi)$ and $\bar{t} \in \mathfrak{U}_L$ denotes that any component t_i of \bar{t} is a ground term from \mathfrak{U}_L . The semantics for the other logical connectives and for the universal quantifier \forall follows from the combination of \vee , \neg , and \exists .

A Herbrand model for a formula ψ is a Herbrand interpretation which satisfies ψ . A formula ψ is Herbrand valid, in symbols $\models \psi$, iff it is satisfied in every Herbrand model. A formula ψ is a logical Herbrand consequence of ϕ , in symbols $\phi \models \psi$, if and only if any Herbrand model satisfying ϕ satisfies also ψ . Note that ψ is a logical Herbrand consequence of ϕ iff $\phi \rightarrow \psi$ is Herbrand valid.

The minimum set of agent-oriented concepts (thus common to all agent languages) should include a notion of a signature, mental state and action. The signature of an agent A , Σ_A , is a subset of Σ_L . Σ_A determines the Herbrand universe \mathfrak{U}_A and the Herbrand base \mathfrak{B}_A of A . In the next sections we focus on two main aspects which relate to mental states and actions: (1) *what* is the *data* agents manipulate? and (2) *how* is the data *transformed* by agents?

2.1.1 Mental States

Mental states contain data that agents manipulate. This data we refer to as *beliefs*. We model beliefs as *sorted ground atoms*, i.e., predicates on sorted ground terms. Beliefs are organised in so-called *belief bases* which we will denote by \mathcal{B} . Each belief base \mathcal{B} has a corresponding signature $\Sigma_{\mathcal{B}}$ which is a subset of Σ_A . Since belief bases are *sets* of ground atoms, they have a double interpretation: (1) as subsets of the Herbrand base \mathfrak{B}_A and (2) as conjunctions of ground atoms. This distinction will be useful in Section 2.1.3.

2.1.2 Actions

Mental states change by means of executing basic actions. Basic actions may be conditional to allow more expressive constructions. Basic conditional (bc-)actions have the form:

$$a(\bar{t}) = (\psi, \xi) \text{ if } Cond$$

with a being an action name, \bar{t} a list of parameters, ψ (resp. ξ) a pre- (resp. post-) condition and $Cond$ an *equality-based condition* which we detail later. The construction *if Cond* is not a required element. When this construction is missing, the action is simply called *basic*. The precondition ψ is a first-order formula where the free variables are interpreted existentially. The post-condition is a set of literals which we also understand as a conjunction when this notion is more adequate. For clarity, we denote ξ as a *set* and we will sometimes refer to ξ^+ (resp. ξ^-) as the set of positive (resp. negative) literals. For a bc-action to be *well-defined*, the following condition on variables must be fulfilled:

$$Vars(\xi) \cup Vars(Cond) \subseteq Vars(\psi) \subseteq Vars(\bar{t})$$

This requirement has the advantages that it is easy to be checked and that it ensures that the effects and the equality-based conditions of well-defined actions are *ground*. The fact that effects are *ground* terms is important because, as will be clear from the definition of updates, it guarantees that the belief bases themselves are collections of *ground* terms. The fact that equality-based conditions involve *ground* terms means that the conditions can be checked syntactically by *matching*. We notice that if we were to include $Cond$ in the precondition, i.e., in the form of $\psi \wedge Cond$, the groundness of equality-based conditions would not have been trivially ensured anymore.

The mechanism of a bc-action $a(\bar{t}) = (\psi, \xi) \text{ if } Cond$ is simple. Whenever enabled, i.e.:

1. the precondition ψ *matches*² the current belief base with a *matcher* (a ground substitution) θ
2. the ground condition $Cond\theta$ is true
3. the ground effect $\xi\theta$ is consistent, that is, it does not contain both l and $\neg l$, where l is a ground literal

the belief base is updated with the ground effect of the bc-action:

$$\mathcal{B} \uplus \xi\theta = \mathcal{B} \cup \{l \mid l \in \xi\theta\} \setminus \{l \mid \neg l \in \xi\theta\}$$

where \uplus denotes the update operation. The result of the update is automatically guaranteed to be consistent since we add only positive literals.

Both requirements 1 and 2 involve that particular forms of formulae “hold”. We discuss this in the next section. Requirement 3 is meant to overcome peculiar situations like

²This follows from the fact that the belief bases are ground. It is important to notice that what we solve is a *matching* (and not *unification*) problem. From a complexity point of view, this is important since it is easier to implement a linear algorithm for matching than for unification.

the following one. Assume an agent has an initial belief base $\{P(c), Q(c)\}$ and a basic action $a(x, y) = (P(x) \wedge Q(y), \{\neg S(x), S(y)\})$. Given that the only matcher is the substitution $[x/c][y/c]$ the agent is in difficulty: if it first tries to delete $S(c)$ and then to add $S(c)$ the resulting belief base is $\{P(c), Q(c), S(c)\}$, however, if it proceeds the other way around, then the belief base remains unchanged. One solution is to fix a priority, for example first add and then delete. However, we find such solutions rather ad hoc and thus we avoid them by forbidding updates with *inconsistent* ground effects. For a ground effect $\xi\theta$, we use the notation $\perp \in \xi\theta$ to denote that $(\exists l)(\{l, \neg l\} \subseteq \xi\theta)$.

2.1.3 Queries

In what follows we discuss Requirements 1 and 2 by first referring to Requirement 1. Preconditions are understood as implicitly existentially quantifiers, that is, a precondition ψ represents the formula $\exists \bar{x}\psi$, where $\bar{x} = \text{Var}(\psi)$. Their purpose is to act as *queries* for belief bases. Given an agent A with its associated Herbrand universe \mathcal{U}_A , the semantics of queries is with respect to belief bases. Since one way to understand belief bases is that they are subsets of the Herbrand base \mathfrak{B}_A , we find it natural to define the semantics of queries by means of Herbrand models. We say that a query ψ holds in a belief base \mathcal{B} if and only if \mathcal{B} is a Herbrand model of ψ , in symbols $\models_{\mathcal{B}} \exists \bar{x}\psi$. We draw attention that the ground terms substituting \bar{x} are from \mathcal{U}_A . For convenience, we use the notation “ $\models_{\mathcal{B}} \psi\theta$ for some $\theta : \text{Var}(\psi) \rightarrow \mathcal{U}_A$ ” instead of “ $\models_{\mathcal{B}} \psi(\bar{t})$ for some $\bar{t} \in \mathcal{U}_A$ ” and “ $\models_{\mathcal{B}} \psi$ ” instead of “ $\models_{\mathcal{B}} \exists \bar{x}\psi$ ” when it is clear from the context that ψ is a query.

The definition of Herbrand satisfaction for existential formulae involves choosing “some” ground terms from the Herbrand universe. It is thus a vague statement, not satisfactory when the interest is in computability, i.e., when we want to determine precisely those substitutions θ such that $\models_{\mathcal{B}} \psi\theta$. In what follows, we address the same issue as in logic programming where $\models_{\mathcal{B}} \exists \bar{x}\psi$ has a *procedural* interpretation “given \mathcal{B} , solve ψ ” rather than a *denotational* one “given \mathcal{B} , ψ is true”. To stress the separation we will explicitly use the notation $\psi \leq^? \mathcal{B}$ to symbolically represent the procedural interpretation of $\models_{\mathcal{B}} \exists \bar{x}\psi$.

Solving Queries

In this section the interest is more on *how* to *compute*, i.e., on designing an algorithm for answering queries. Since this is an internal operation upon which the whole agent’s execution relies, the algorithm, besides being sound and complete, must terminate. Knowing that the satisfiability of formulae with unrestricted alternations of universal and existential quantifiers is undecidable, to have an effective algorithm for $\models_{\mathcal{B}} \exists \bar{x}\psi$, the format of ψ must be restricted. A natural solution is to think of queries as existential closures of *quantifier free* formulae. In such a setup, thinking of \mathcal{B} as a conjunction of atoms, $\mathcal{B} = \bigwedge_n a_n$, and of ψ as transformed into disjunctive normal form, i.e., $\psi = \bigvee_i \psi_i$ and $\psi_i = \bigwedge_j l_{ij}$, we have the following reasoning:

$$\begin{aligned}
& \models_{\mathcal{B}} \exists \bar{x} \psi \quad \text{iff} \quad \models \mathcal{B} \rightarrow \exists \bar{x} \psi \quad \text{iff} \\
& \neg(\mathcal{B} \rightarrow \exists \bar{x} \psi) \equiv \mathcal{B} \wedge_i \neg \psi_i (*) \equiv \\
& \mathcal{B} \wedge (\neg l_{i_1} \vee \dots \vee \neg l_{i_k}) \dots \wedge (\neg l_{i_1} \vee \dots \vee \neg l_{i_p}) \text{ is unsatisfiable}
\end{aligned}$$

where the latter formula is in negation normal form (because \mathcal{B} contains only atoms) and all its variables (i.e., those appearing in the literals) are universally quantified. We know that a formula in negation normal form containing only universal quantifiers is unsatisfiable iff it is unsatisfiable in every Herbrand interpretation and, if needed, we refer to [Gal85] for a proof. That is, if there is a solution, i.e., an *answer substitution*, for $\mathcal{B} \rightarrow \exists \bar{x} \psi$ then this can be found in the Herbrand universe. Naturally, there may be more solutions (because the belief base is a set and the goal may be a disjunction). To find one, it suffices to look for those indexes i for which the set $\mathcal{B} \cup \neg \psi_i$ is unsatisfiable. By the compactness theorem from first-order logic, we know that for some literals l of each ψ_i for which $a_1 \wedge \dots \wedge a_n \wedge \neg l$ is unsatisfiable.

2.1.2. EXAMPLE. Let \mathcal{B} be a belief base $\{on(1,0), on(2,1)\}$ and ψ_i a query given by the existential closure of $on(x,y) \wedge on(y,z) \wedge \neg on(z,x)$. We have that $\mathcal{B} \cup \neg \psi_i$ is unsatisfiable because we can choose $on(x,y)$ such that $on(2,1) \wedge \neg on(x,y)$ is unsatisfiable with the (partial) answer being the substitution $[x/2][y/1]$.

The second iteration returns an answer $[z/0]$. Of course, if in the first step the answer was $[x/1][y/0]$ (because also $on(1,0) \wedge \neg on(x,y)$ is unsatisfiable), then we had no answer in the second step, thus an unsuccessful computation.

The answer substitution is $[x/2][y/1][z/0]$ grounds the negative literal $\neg on(z,y)$ to $on(0,2)$ which is not in the belief base thus the answer substitution lead to a success. ♠

To draw a parallel with the standard computations in logic programming, solving ψ in \mathcal{B} is a *particular* case of SLDNF resolution (Selection rule driven Linear resolution for Definite clauses with Negation as Failure). It is particular because: (a) \mathcal{B} can be seen as a conjunction of atoms $\bigwedge_n a_n$, i.e., a logic program with only ground unit clauses, and (b) ψ is an unrestricted goal, a disjunction of *general goals*, with possible negative literals. We recall that the basic idea of resolution is to show that the logic program together with the negated goal is unsatisfiable. A resolution step consists in choosing for each ψ_i a literal l_{i_j} which unifies with the head of a clause in the program. This step is iterated until a resolvent $R(t)$ is found such that there exists a literal $\neg R(t')$ in $\neg \psi_i$ and t unifies with t' . In our case, thanks to (a), solving queries is simpler than resolution: only one iteration step is needed per ψ_i which consists of choosing an l_{i_j} and an a_n from \mathcal{B} . Moreover, since any a_n is ground, what we have is a *matching* and not a *unification* problem. This justifies our choice to overload $\leq^?$ in the notation $\psi \leq^? \mathcal{B}$ introduced above. On the other hand (b) makes implementation more difficult than basic (SLD) resolution because it needs negation as failure (NAF). To conclude, we can design an algorithm for gathering all possible solutions (*matchers*) for $\psi \leq^? \mathcal{B}$ in a set denoted by *Sols* and this is what we depict in Figure 2.1. There, ψ_i^+ (resp. ψ_i^-) denotes the positive (resp. negative) literals from ψ_i , $|l|$ denotes l stripped of negation and \square denotes an unsuccessful, or better said, failed computation. To compute the substitution from (**) we only need to compute one element in $\mathcal{U}_{\mathcal{B}}$ which is not in \mathcal{B} . In the line (***) “otherwise” explicitly means that the atom from l is an element of $\mathcal{B} \mid l \in \mathcal{B}$, or that l is not ground

$$\begin{aligned}
Sols(\mathcal{B}, \bigvee_i \psi_i) &= \bigcup_i Sols(\mathcal{B}, \psi_i) \\
Sols(\mathcal{B}, \psi_i) &= \{\theta \mid \theta \in Sols(\mathcal{B}, \psi_i^+) \text{ and } \square \notin Sols(\mathcal{B}, \psi_i^-)\} \\
Sols(\mathcal{B}, \psi_i^+) &= \bigcup_{l \in \psi_i^+} \{Sols(\mathcal{B}, (\psi_i^+ - l)\theta) \mid \theta \in Sols(\mathcal{B}, l)\} \\
Sols(\mathcal{B}, l) &= \{l \leq^? a_n \mid a_n \in \mathcal{B}\} \quad (*) \\
\\
Sols(\mathcal{B}, \psi_i^-) &= \begin{cases} \emptyset, & \forall l \in \psi_i^- l \text{ is ground and } l \notin \mathcal{B} \\ \theta, & \theta \in Sols(\mathfrak{B}_{\mathcal{B}} \setminus \mathcal{B}, \neg l) \quad (**) \\ \square, & \text{otherwise} \quad (***) \end{cases}
\end{aligned}$$

Figure 2.1: A Procedure for $\psi \leq^? \mathcal{B}$

and that $\mathfrak{U}_{\mathcal{B}}$ equals \mathcal{B}^3 . Further, in line (*) we use $l \leq^? a_n$ to denote the standard syntactic matching problem. For the sake of completeness, we present below an adaptation of a rule based algorithm, one of the earliest, due to Huet [Hue76]. The description is based on the one from [KK99, Chapter 3]. Though not difficult to observe, it is convenient to mention that the

$$\begin{aligned}
f(\bar{t}) \leq^? g(\bar{t}) &= \square, & \text{if } f \neq g & \quad [\text{SymbolClash}] \\
f(t_1, \dots, t_n) \leq^? f(t'_1, \dots, t'_n) &= \bigwedge_{i=1}^n t_i \leq^? t'_i & \quad [\text{Decomposition}] \\
t \leq^? t \wedge Eqs &= Eqs & \quad [\text{Delete}] \\
t \leq^? x \wedge Eqs &= \square, & \text{if } x \text{ is a variable} & \quad [\text{SymbolVariableClash}] \\
x \leq^? t \wedge x \leq^? t' \wedge Eqs &= \square, & \text{if } t \neq t' & \quad [\text{MergeClash}] \\
\bigwedge_{i=1}^n x_i \leq^? t_i &= [\bar{x}/\bar{t}], & \text{if } x_i \text{ are all different} & \quad [\text{Match}]
\end{aligned}$$

Figure 2.2: Syntactic Matching

rules from Figure 2.2 are proved in [KK99] to be sound, complete and terminating. This is also the case for $\psi \leq^? \mathcal{B}$. More precisely, it can be proved that $\models_{\mathcal{B}} \exists \bar{x} \psi$ if and only if $\mathcal{B} \vdash \psi$, where $\mathcal{B} \vdash \psi$ denotes that there is a successful computation of an answer solution, i.e., there exists a substitution in $Sols(\mathcal{B}, \psi)$. For instance, the proof follows by adapting the soundness and completeness⁴ for SLDNF. We also notice that, in case of the existence of a solution for $\models_{\mathcal{B}} \exists \bar{x} \psi$, \mathcal{B} is, in fact, the least Herbrand model for ψ , i.e., the smallest subset of the Herbrand base where ψ evaluates to true.

Returning to the requirements for updates, we recall that the separation between preconditions and equality-based conditions was intended to ensure that equality-based conditions are ground which is what Requirement 2 asks for. Thus these can be checked syntactically in

³We note that $\mathfrak{U}_{\mathcal{B}}$ equals \mathcal{B} iff there are no function symbols in the agent language L , otherwise $\mathfrak{U}_{\mathcal{B}}$ is infinite, as we mentioned in the beginning of Section 2.1

⁴In the general case, due to the so-called *floundering* programs, the completeness is restricted to *allowed* programs [Apt88]. However, this poses no problem in our particular situation where the belief bases are ground unit clauses because they represent, in fact, allowed programs.

the theory of equality, denoted by $T^=$, i.e., by means of the equality axioms:

$(\forall x)(x = x)$	[reflexivity]
$(\forall x, y)(x = y \rightarrow y = x)$	[symmetry]
$(\forall x, y, z)(x = y \wedge y = z \rightarrow x = z)$	[transitivity]
$(\forall \bar{x}, \bar{y})(\bigwedge x_i = y_i \rightarrow f(\bar{x}) = f(\bar{y}))$	[function congruence]
$(\forall \bar{x}, \bar{y})(\bigwedge x_i = y_i \rightarrow p(\bar{x}) = p(\bar{y}))$	[predicate congruence]

2.1.3. EXAMPLE. We consider a problem with blocks, where blocks are identified by natural numbers. An action for moving one block x from block y to block z if there is no block on neither x nor z can be designed as follows:

$$\begin{aligned} \text{move}(x, y, z) = & (\text{on}(x, y) \wedge \text{clear}(x) \wedge \text{clear}(z), \\ & \{ \text{on}(x, z), \neg \text{on}(x, y), \neg \text{clear}(z), \text{clear}(y) \}) \\ & \text{if } z \neq 0 \wedge y \neq z. \end{aligned}$$

with 0 being used to identify the floor where the blocks are situated. Given a belief base $\{\text{on}(2, 1), \text{clear}(2), \text{clear}(3)\}$, matching the precondition $\text{move}(x, y, z)$ returns the substitution $[x/2][y/1][z/3]$ which grounds the condition of move to $3 \neq 0 \wedge 1 \neq 3$. It is not difficult to see that the ground conditions reduce to true in $T^=$. ♠

To conclude, the logical framework for solving first-order queries and respectively ground equality-based conditions requires a first-order computation mechanism which bares resemblance with first-order resolution and respectively the axioms of equality. To simplify, for the rest of thesis we consider that the theory of equality is implicit in the underlying logical framework of the agent languages we work with. Also, to ease notation, we adopt the shorter $\theta \in \text{Sols}(\mathcal{B}, \psi \wedge \text{Cond})$ instead of “ $\theta \in \text{Sols}(\mathcal{B}, \psi)$ and $T^= \vdash \text{Cond}\theta$ ”.

2.2 A Specification Agent Language: BUnity

In this section we introduce an abstract agent language which we call BUnity. It is meant to represent abstract agent specifications and to model agents at a coarse level, using a minimal set of constructions on top of the notions of mental states and actions which were the subject of the previous section. A BUnity agent is abstract in the sense that it is oblivious with respect to any form of specific orderings (for example, action planning). As a consequence, the executions of a BUnity agent are highly nondeterministic.

2.2.1 Syntax

The mental state of a BUnity agent is simply a belief base. As for actions, besides the basic conditional ones, BUnity language allows a finer type of construction, *triggers*, organised in a set denoted by \mathcal{A}^t . Triggers have the form $\{\phi\} \triangleright \text{do}(a(\bar{t})) \text{ if } \text{Cond}$ where ϕ is a first-order formula with only free variables, Cond an equality-based condition, a is a bc-action name and \bar{t} the arguments of a such that $a(\bar{t})$ can be understood as an “action”-call. Intuitively, triggers

are like *await* statements in imperative languages: **await** ϕ **do** a , action a can be executed only when ϕ matches the current belief base.

Both bc-actions and triggers have enabling conditions, and this might raise confusion when distinguishing them. The intuition lying behind the need to consider them both is that they demand information at different levels. The precondition of a bc-action should be understood a built-in restriction which internally enables belief updates. It is independent of the particular choice of application. A trigger is executed in function of certain external requirements (reflected in the mental state). Thus it is application-dependent. The external requirements are meant to be independent of the built-in enabling mechanism. Whether is agent “A” or agent “B” executing an action a , the built-in condition should be the same for both of them. Nevertheless, each agent may have its own external trigger for a . This distinction allows re-usability of bc-actions. For example, the action *move* from the previous section is implemented in the same way regardless of initial configurations. On the contrary, triggers are meant to “trigger” bc-actions precisely in function of the initial configurations:

$$\{on(x,y) \wedge y \neq 0\} \triangleright do(move(x,y,0))$$

which is meant to allow an unstack move of x from y as long as y is not the floor. This will turn out to be quite useful later in this section.

A BUnity configuration is a tuple $(\mathcal{B}, \mathcal{A}^b, \mathcal{A}^t)$, where \mathcal{B} is a set of beliefs, \mathcal{A}^b is the set of bc-actions and \mathcal{A}^t is the set of triggers. A configuration is called *initial* when \mathcal{B} is a set of initial beliefs, usually denoted by \mathcal{B}_0 . A BUnity agent is defined as a pair consisting of a sorted signature Σ and an initial configuration. The sorted signature gives the sorts and ranges of operations on the sorts. The sorted signature we have used so far makes use of \mathbb{N} and has only one operation $on : \mathbb{N} \times \mathbb{N} \rightarrow Pred$ meaning that *on* is a predicate taking as arguments two naturals. We assume \mathbb{N} (respectively *Pred*) to be a pre-defined signature (respectively sort) accessible to any agent program.

2.2.2 Operational Semantics

Since the work of McCarthy [McC62], one long advocated approach to describing changes in programs is by means of operational semantics. This approach is appealing because it is intuitive (compared to denotational) and because it gives flexibility. The operational semantics works with labelled transition systems whose states represent data by closed terms over an algebraic structure, and whose transitions between states are obtained from the collection of so-called transition rules of the form $\frac{premises}{conclusion}$. This is also the approach we adopt to define the semantics of BUnity operationally in terms of labelled transition systems. The states are represented by belief bases, since only these have a dynamic structure and the only possible transition denotes the transformation of belief bases:

2.2.1. DEFINITION. [BUnity Semantics] Let $(\mathcal{B}_0, \mathcal{A}^b, \mathcal{A}^t)$ be a BUnity configuration. The associated LTS is $(\Sigma, \mathcal{B}_0, L, \rightarrow)$, where:

- Σ is a set of states (belief bases)
- \mathcal{B}_0 is the initial state (the initial belief base)

- L is a set of ground action terms
- \rightarrow is the transition relation, given by the rule in Figure 2.3.

$$\boxed{
 \frac{
 \begin{array}{l}
 \phi \triangleright do(a) \in \mathcal{A}^t \text{ if } Cond_1 \quad a = (\psi, \xi) \text{ if } Cond_2 \in \mathcal{A}^b \\
 \theta \in Sols(\mathcal{B}, (\phi \wedge \psi \wedge Cond_i)) \quad \perp \notin \xi \theta
 \end{array}
 }{
 \mathcal{B} \xrightarrow{a\theta} \mathcal{B} \uplus \xi \theta
 } \quad (act)$$

Figure 2.3: The Transition Rule for BUnity



We consider the meaning of a BUnity agent defined in terms of possible sequences of mental states, i.e., the set of all (maybe infinite) computations $\sigma = \mathcal{B}_0 \xrightarrow{a_1} \dots \xrightarrow{a_i} \mathcal{B}_i \xrightarrow{a_{i+1}} \dots$. The observable *behaviour*, $Tr(\mathcal{B}_0)$, is the set of all traces, i.e., the sequences of actions executed during computations, from the initial configuration. Our choice to observe actions (and not states) is motivated by the fact that, in studying simulation, we are interested in *what we see* and not *how the agent thinks*. We take the case of a robot: one simulates his physical actions, lifting or dropping a block, for example, and not the mental states of the robot.

The transition rule (*act*) captures the effects of performing the action a . It basically says that if there is a trigger $\phi \triangleright do(a)$ and the query ϕ has a solution in the current mental state then if the precondition of a matches the current belief base new beliefs are added/deleted with respect to the effects of a .

We take, as an illustration, the Hanoi towers problem. We assume blocks are identified by natural numbers. The initial arrangement is of three blocks 1, 2, 3 is given there: 1 and 2 are on the floor, and 3 is on top of 1. The goal⁵ of the agent is to rearrange them such that 1 is on the floor, 2 on top of 1 and 3 on top of 2. The only action an agent can execute is to move one block on the floor, or on top of another block, if the latter is clear. To allow moves on the floor, the floor is always clear. We deliberately separate between moving *to* the floor and moving on a different block in order to have a consistent description of the effects, more precisely, in order to avoid deleting *clear*(0).

The example from Figure 2.4 is meant to show another situation where the difference between bc-actions and triggers matters. On the one hand, it is possible to move a block x on top of another block z , if x and z are clear; on the other hand, given the goal of the agent, moves are allowed only when the configuration is different than the final one.

2.2.3 Fair Executions of BUnity Agents

State-transition systems describe only the “safe” behaviours of systems. This means that in nondeterministic systems that abstract from scheduling policies, some traces are improbable

⁵We do not explicitly model goals. Please check Section 2.5 for a discussion motivating our choice.

$$\begin{aligned}
\Sigma &= \{on : \mathbb{N} \times \mathbb{N} \rightarrow Pred\} \\
\mathcal{B}_0 &= \{ on(3,1), on(1,0), on(2,0), clear(2), clear(3) \} \\
\mathcal{A}^b &= \{ move(x,y,0) = (on(x,y) \wedge clear(x), \{ on(x,0), \neg on(x,y), clear(y) \}) \\
&\quad move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \\
&\quad \{ on(x,z), \neg on(x,y), \neg clear(z), clear(y) \}) \text{ if } z \neq 0 \wedge y \neq z \} \\
\mathcal{A}^t &= \{ \neg(on(2,1) \wedge on(3,2)) \triangleright do(move(x,y,z)) \}
\end{aligned}$$

Figure 2.4: A BUnity Toy Agent

to occur in real computations. In this sense, the operational semantics (given by transition rules) is too general in practise: if the actions an agent can execute are always enabled, it should not be the case that the agent always chooses the same action. Such executions are usually referred to as being *unfair*. In order to model live behaviours, we have to augment systems with fairness conditions, which partition the infinite computations of a system into fair and unfair computations [Fra86].

For example, we imagine a scenario illustrative for cases where modelling fairness constraints is a “must”. For this, we slightly complicate the “tower” problem from the previous section, by giving the agent described in Figure 2.4 an extra assignment to clean the floor, if it is dirty. Thus the agent have two alternatives: either to clean or to build. We add a basic action, $clean = (\neg cleaned, \{cleaned\})$. We enable the agent to execute this action at any time, by setting \top as the query of the trigger calling $clean$, i.e., $\top \triangleright do(clean)$.

We note that it is possible that the agent always prefers cleaning the floor instead of rearranging blocks, in the case that the floor is constantly getting dirty. We want to cast aside such traces and moreover, we want a declarative, and not imperative solution. Our option is to follow the approach from [MP92]: we constrain the traces by adding *fairness* conditions, modelled as LTL properties. Fairness is there expressed either as a weak, or as a strong constraint. They both express that actions which are “many times” enabled on infinite execution paths should be infinitely often taken. The difference between them is in the definition of “many times” which is continuously (resp. infinitely often). Due to the semantics of triggers, it follows that the *choice* of executing one action cannot disable the ones not chosen and thus BUnity agents only need weak fairness.

2.2.2. DEFINITION. [Justice [MP92]] A trace is just (weakly fair) with respect to a transition a if it is not the case that a is continually enabled beyond some position, but taken only a finite number of times. \blacklozenge

To model such a definition as LTL formulae we need only two future operators, \Diamond (*eventually*) and \Box (*always*). Their satisfaction relation is defined as follows:

$$\begin{aligned}
\sigma \models \Diamond \phi &\text{ iff } (\exists i > k)(s_i \models \phi) \\
\sigma \models \Box \phi &\text{ iff } (\forall i > k)(s_i \models \phi),
\end{aligned}$$

where s_0, \dots, s_k, \dots are the states of a computation σ . By means of these operators we define weak fairness for BUnity as:

$$just_1 = \bigwedge_{a \in Act} (\Diamond \Box \text{enabled}(\phi \triangleright do(a)) \rightarrow \Box \Diamond \text{taken}(a)) .$$

where *enabled* and *taken*, predicates on the states of BUnity agents, are defined as:

$$\begin{aligned} \mathcal{B} \models \text{enabled}(\phi \triangleright do(a))\theta & \quad \text{iff} \quad a = (\psi, \xi) \text{ if } Cond \wedge \theta \in Sols(\mathcal{B}, \phi \wedge \psi \wedge Cond) \\ \mathcal{B} \models \text{taken}(a)\theta & \quad \text{iff} \quad \mathcal{B} \xrightarrow{a\theta} \mathcal{B}' \end{aligned}$$

Such a fairness condition ensures that all fair BUnity traces are of the form $(clean^* (move\theta)^*)^\omega$, or equally $\{(clean^n (move\theta)^m)^k \mid \forall n, m \in \mathbb{N}, k \in \mathbb{N} \cup \{\infty\}\}$. We note that the advantage of a declarative approach to modelling fairness is the fact that we do not need to commit to a specific scheduling policy as it is the case when implementing fairness by means of a scheduling algorithm, for example Round-Robin. A scheduling policy would basically correspond to fixing the exponents n and m .

2.3 From BUnity to BUpL: Refining Control

The BUnity agent described in Figure 2.4 is highly nondeterministic. It is possible that the agent moves 3 on the floor, 2 on 1, and 3 on 2. This sequence represents, in fact, the shortest one to achieving the goal. However, it is also possible that the agent pointlessly move 3 from 1 to 2 and then back from 2 to 1.

2.3.1 Syntax

BUpL language allows the construction of *plans* as a way to order actions. We refer to \mathcal{P} as a set of plans, with a typical element p , and to Π as a set of plan names, with a typical element π . Syntactically, a plan is defined by the following BNF grammar:

$$p ::= a(t, \dots, t) \mid \pi(t, \dots, t) \mid a(t, \dots, t); p \mid p + p$$

with ';' being the *sequential composition* operator and '+' the *choice* operator, with a lower priority than ';'.¹

The construction $\pi(x_1, \dots, x_n)$ is called *abstract plan*. It is a function of arity n , defined as $\pi(x_1, \dots, x_n) = p$. Abstract plans should be understood as procedures in imperative languages: an abstract plan calls another abstract plan, as a procedure calls another procedure inside its body.

BUpL language provides a mechanism for handling the failures of actions in plans through constructions called *repair rules*. A plan fails when the current action cannot be executed. Repair rules replace such a plan with another. Syntactically, they have the form $\phi \leftarrow p$, and it means: if ϕ matches \mathcal{B} , then substitute the plan that failed for p .

A BUpL configuration is a tuple $(\mathcal{B}_0, \mathcal{A}^b, \mathcal{P}, \mathcal{R}, p_0)$, where $\mathcal{B}_0, \mathcal{A}^b$ are the same as for a BUnity agent, p_0 is the *initial plan*, \mathcal{P} is a set of plans and \mathcal{R} is a set of repair rules. A BUpL agent is a pair of a sorted signature and an initial configuration.

2.3.2 Operational Semantics

Plans, like belief bases, have a dynamic structure, and this is why the mental state of a BUpL agent incorporates both the current belief base and the plan in execution. The operational semantics for a BUpL agent is as follows:

2.3.1. DEFINITION. [BUpL Semantics] Let $(\mathcal{B}_0, \mathcal{A}^b, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL configuration. Then the associated LTS is $(\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)$, where:

- Σ is a set of states, tuples (\mathcal{B}, p)
- (\mathcal{B}_0, p_0) is the initial state
- L is a set of labels, either ground action terms or τ
- \rightarrow represents the transition rules given in Figure 2.5.

$$\begin{array}{c}
 \frac{a = (\psi, \xi) \text{ if } Cond \in \mathcal{A}^b \quad \theta \in Sols(\mathcal{B}, \psi \wedge Cond) \quad \perp \notin \xi \theta}{(\mathcal{B}, a; p') \xrightarrow{a\theta} (\mathcal{B} \uplus \xi \theta, p' \theta)} \quad (act) \\
 \\
 \frac{(\mathcal{B}, p_i) \xrightarrow{\mu} (\mathcal{B}', p')}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\mu} (\mathcal{B}', p')} \quad (sum_i) \\
 \\
 \frac{(\mathcal{B}, a; p) \not\xrightarrow{a} \quad \phi \leftarrow p' \in \mathcal{R} \quad \theta \in Sols(\mathcal{B}, \phi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p' \theta)} \quad (fail) \\
 \\
 \frac{\pi(x_1, \dots, x_n) := p}{(\mathcal{B}, \pi(t_1, \dots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \dots, t_n))} \quad (\pi)
 \end{array}$$

Figure 2.5: BUpL Rules



As it was the case for BUnity agents, we consider the meaning of a BUpL agent defined in terms of possible sequences of mental states, and its externally observable *behaviour*, $Tr((\mathcal{B}, p))$, as sequences of executed actions.

The transition rule *(act)* captures the effects of performing the action a which is the head of the current plan. It basically says that if θ is a solution to the matching problem $\mathcal{B} \models \psi$ where ψ is the precondition of action a then the current mental state changes to a new one,

where the current belief base is updated with the effects of a and the current plan becomes the “tail” of the previous one. The transition rule (*fail*) handles exceptions. If the head of the current plan is an action that cannot be executed (the set of solutions for the matching problem is empty) and if there is a repair rule $\phi \leftarrow p'$ such that the new matching problem $\phi \text{leq}^? \mathcal{B}$ has a solution θ then the plan is replaced by $p'\theta$. The transition rule (π) implements “plan calls”. If the abstract plan $\pi(x_1, \dots, x_n)$ defined as $p(x_1, \dots, x_n)$ is instantiated with the terms t_1, \dots, t_n then the current plan becomes $p(t_1, \dots, t_n)$ which stands for $p[x_1/t_1] \dots [x_n/t_n]$. In both transitions (*fail*) and (π) we use τ as a special label to denote internal changes in the agent. This is in order to distinguish such internal aspects from the *observable* behaviour represented by ground actions. The transition rule (sum_i) replaces a choice between two plans by either one of them. The label μ can be either a ground action name or a τ step, in which case $\mathcal{B}' = \mathcal{B}$, and p' is a valid repair plan (if any).

We take as an example a BUPL agent that solves the same Hanoi towers problem. It has the same initial belief base and the same basic action as the BUnity agent.

$$\begin{aligned}
\Sigma &= \{on : \mathbb{N} \times \mathbb{N} \rightarrow Pred\} \\
\mathcal{B}_0 &= \{ on(3, 1), on(1, 0), on(2, 0), \\
&\quad clear(2), clear(3) \} \\
\mathcal{A}^b &= \{ move(x, y, 0) = (on(x, y) \wedge clear(x), \{ on(x, 0), \neg on(x, y), clear(y) \}) \\
&\quad move(x, y, z) = (on(x, y) \wedge clear(x) \wedge clear(z), \\
&\quad \{ on(x, z), \neg on(x, y), \neg clear(z), clear(y) \}) \text{ if } z \neq 0 \wedge y \neq z \} \\
\mathcal{P} &= \{ p_0 = move(2, 0, 1); move(3, 0, 2) \} \\
\mathcal{R} &= \{ on(x, y) \leftarrow move(x, y, 0); p_0 \}
\end{aligned}$$

Figure 2.6: A BUPL Toy Agent

The BUPL agent from Figure 2.6 is modelled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move 2 on 1. Similar scenarios can easily arise in multi-agent systems: imagine that initially 3 is on the floor, and the agent decides to move 2 on 1; imagine also that another agent comes and moves 3 on top of 1, thus moving 2 on 1 will fail. The failure is handled by $on(x, y) \leftarrow move(x, y, 0); p_0$. Choosing $[x/1][y/3]$ as a matcher, enables the agent to move C on the floor and after the initial plan can be restarted.

2.3.3 Fair Executions of BUPL Agents

Though BUPL agents are meant to reduce the nondeterminism from BUnity agent specifications, unfair executions are not ruled out because of the nondeterminism in the choices between plans and/or repair rules. To illustrate this, we assign a *mission* plan to the BUPL agent described in Figure 2.6, $mission = cleanR + rearrange(2, 1, 3)$, where *cleanR* is a tail-

recursive plan, $cleanR = clean; cleanR$, with $clean$ being the action defined in Section 2.2.3. The plan $rearrange$ generalises the previously defined p_0 : $rearrange(x, y, z) = move(x, 0, y); move(z, 0, x)$. It consists of reorganising clear blocks placed on the floor, such that they form a tower. This plan fails if not all the blocks are on the floor, and the failure is handled by the already defined repair rule, which we call r_1 . We add a repair rule, r_2 , $\top \leftarrow mission$, which simply makes the agent restart the execution of the plan $mission$.

As it was the case with the BUnity agent from the Section 2.2.3, it is possible, in the above scenario, that the BUpL agent always prefers cleaning the floor instead of rearranging blocks, though this is useless when the floor has already been cleaned. Nevertheless, such cases are disregarded if one requires that executions are fair. The only difference from the fairness condition imposed on the executions of BUnity agents is that plans need not be continuously but infinitely often enabled.

We consider two scenarios for defining fairness with respect to choices in repair rules and plans. The execution of $rearrange$ has failed. Both repair rules r_1 and r_2 are enabled, and always choosing r_2 makes it impossible to make the rearrangement. This would not be the case if r_1 were triggered. It follows that the choice of repair rules should be weakly fair:

$$just_2 = \bigwedge_{p \in \mathcal{P}} (\Diamond \Box enabled(\phi \leftarrow p) \rightarrow \Box \Diamond taken(p)).$$

The repair rule r_1 has been applied, and all three blocks are on the floor. Returning to the initial mission and being in favour of cleaning leads again to a failure (the floor is already clean). The only applicable repair rule is r_2 which simply tells the agent to return to the mission. Thus, it can be the case that, though rearranging the blocks is enabled, it will never happen, since the choice goes for the plan $clean$ (which always fails). Therefore, because plans are not continuously enabled, their choice has to be strongly fair:

2.3.2. DEFINITION. [Compassion [MP92]] A trace is compassionate (strongly fair) with respect to a transition a if it is not the case that a is infinitely often enabled beyond some position, but taken only a finite number of times. \blacklozenge

As it was the case with justice, modelling the above definition as a linear temporal logic formula is straightforward, however we refer to *plans* instead of actions:

$$compassionate = \bigwedge_{p \in \mathcal{P}} (\Box \Diamond enabled(p) \rightarrow \Box \Diamond taken(p))$$

In the above scenarios *enabled* and *taken* are defined similarly as in the case of actions for the language BUnity. The important observation is that a plan are enabled either when (1) the precondition of its first action is satisfied or (2) it is contained in the expression of a repair rule which is applicable. A plan is taken when its first action is taken.

$$\begin{aligned} (\mathcal{B}, p) \models enabled(p\theta) \quad \text{iff} \quad & p \text{ is } a; p' \text{ and } a = (\psi, \xi) \text{ if } Cond \text{ and } \theta \in Sols(\mathcal{B}, \psi \wedge Cond) \quad (1) \\ & \text{or } \phi \leftarrow p \text{ is applicable, i.e., } \theta \in Sols(\mathcal{B}, \phi) \quad (2) \\ (\mathcal{B}, p) \models taken(p\theta) \quad \text{iff} \quad & p \text{ is } a; p' \text{ and } (\mathcal{B}, a; p') \xrightarrow{a\theta} (\mathcal{B}', p') \end{aligned}$$

The fairness conditions ensure that all fair BUpL traces are of the form $(\text{clean}^* (\text{move}\theta)^*)^\omega$ which is exactly the same as in the case of the BUnity agent. This is a positive result, since we are interested in the fair refinement between the BUpL and the BUnity agent.

2.4 From BUnity to BUnity^K: Refining Data

If in the previous section we have proposed BUpL as an extension of BUnity specifications by means of adding more “control” on the structures, in this section we illustrate a perpendicular extension of BUnity with respect to “data”. This extension is what we call *knowledge bases* and we denote them by \mathcal{K}_i with i an index. Knowledge bases can be seen as ontologies, i.e., *immutable* collections of verified facts which make it possible to express and solve more sophisticated queries on the *contingent* set of beliefs. A knowledge base is the minimal setting which can be added in a modular way, without breaking the semantics of updates, i.e., adding (deleting) positive (negative) literals.

The knowledge bases that we think of are first-order logical theories, i.e., pairs of signatures and axioms. Our motivation is practical: there are already standard first-order theories like Presburger, Integer, Rational, or Real arithmetic, and the theory of Arrays or Lists, for which there are known results about what is decidable. For an overview, we mention [BM07] which provides detailed references for results with respect to each theory. These standard theories facilitate the design of more expressive queries and are meant to provide algorithms for solving the queries in an efficient way.

We distinguish two main orthogonal directions of making use of knowledge bases. On the one hand, when the interest is in *abstraction*, we simply need to use axiomatic extensions of the theory of equality. For example, to say that two beliefs $p(a)$ and $p(b)$ are the same we can use a theory with an axiom $p(a) = p(b)$ as a knowledge base. If we were to synthesise in a few words, we could propose the slogan “abstraction is identification” to say that equations can be used to identify beliefs, or certain patterns in the belief base. This approach can be exploited especially in *verification by abstraction*. For example, equations can be used to finitely partition infinite domains of state spaces, like dividing naturals into odds and evens.

On the other hand, when the interest is in a more expressive representation, we see again two uses. Firstly, to define attributes, properties for given function or predicate symbols, we simply think of \mathcal{K} as an equational theory.

2.4.1. EXAMPLE. Let T be an extension of the theory of equality with the following set E of added axioms:

$$\begin{array}{ll} (\forall x, y)(f(x, y) = f(y, x)) & [\text{comm}] \\ (\forall x, y, z)(f(x, f(y, z)) = f(f(x, y), z)) & [\text{assoc}] \\ (\forall x)(f(0, x) = x) & [\text{l-id}]. \end{array}$$

Thus E contains the axioms for commutativity, associativity and left identity for the functional symbol f . We call it an ACI-theory. ♠

2.4.2. REMARK. Example 2.4.1 brings a relevant observation with respect to the problem of solving first-order queries. We notice that it suffices to think of logical operators as functions

fulfilling certain properties: \neg is a unary function, \wedge, \vee are binary functions satisfying the axioms of associativity, commutativity and identity (\top for \wedge , resp. \perp for \vee) to reduce $\psi \leq^? \mathcal{B}$ to a variant of an ACI-matching problem. Then we can benefit of the vast set of results from the theory of semantic matching which are also main blocks in automated reasoning and theorem proving. To name but a few algorithms for AC-matching we refer to [Eke02, HK99, Con04]. We only note that in our case we always match against a belief base, which is a flat (depth 1) variadic term, thus we have a particular ACI-matching problem. The variant is with respect to the operator corresponding to the logical negation. This requires to be handle with care because trivially $\neg(p(t))$ cannot match anything from \mathcal{B} because there is no term $\neg(_)$ in there. One way to deal with negative terms is to introduce additional rules for each term $p(t)$ representing beliefs: $p(t) \leq^? \neg(p(t)) \rightsquigarrow \square$ and to solve (with the help of the new rules) the matching problems with negative terms at the end, when these terms should already be grounded. ♣

2.4.3. EXAMPLE. As another example which will turn out to be useful in a later section, we consider a knowledge base \mathcal{K} consisting of a predefined associative operator $" , "$ for lists with an identity element 0 and a predicate symbol *tower* with the following axioms:

$$\begin{aligned} (\forall x : \mathbb{N}, y : \mathbb{N}, z : \text{List}) \text{tower}((x, y, z)) &= \text{on}(x, y) \wedge \text{tower}((y, z)). \\ (\forall x : \mathbb{N}) \text{tower}((x)) &= \text{on}(x, 0). \end{aligned}$$

where you used \mathbb{N} (respectively List) to denote the sort of natural numbers (respectively lists). This enables us to construct a trigger:

$$\{\neg \text{tower}((3, 2, 1))\} \triangleright \text{do}(\text{move}(x, y, z))$$

which says that the agent is allowed to perform any legal move as long as the configuration of the blocks is not (3, 2, 1). ♠

In this setting of first-order theories, extending the semantics of queries as described in Section 2.1.3 is easy. The main observation is that a belief base \mathcal{B} and a set of knowledge bases $\bigcup_i \mathcal{K}_i$ can no longer be seen as a Herbrand model since clearly the logical theories representing the knowledge bases may contain variables. This means that we need to define the satisfaction relation by the standard model theoretic semantics with completion in order to handle negative queries:

$$\mathcal{B} \cup \overline{\mathcal{B}} \cup \bigcup_i \mathcal{K}_i \models \exists \bar{x} \psi \quad \text{iff} \quad \mathcal{B} \cup \overline{\mathcal{B}} \cup \bigcup_i \mathcal{K}_i \models \psi \theta \text{ for some ground substitution } \theta \quad (1)$$

where ψ is a query. By logical consequence we can also read (1) as “any model of $\mathcal{B} \cup \overline{\mathcal{B}} \cup \bigcup_i \mathcal{K}_i$ is a model of $\exists \bar{x} \psi$ ” and write it as $\models \mathcal{B} \cup \overline{\mathcal{B}} \cup \bigcup_i \mathcal{K}_i \rightarrow \exists \bar{x} \psi$.

We assume each theory representing knowledge bases comes with its own validity procedure, i.e., a set of deduction rules⁶ which enable us to deduce the validity of a formula. As an illustration, in the ACI-theory from Example 2.4.1 we can prove that the formula $f(0, f(1, 2)) = f(2, 1)$ is valid, by using [assoc], then [l-id] for $f(0, f(1, 2))$ and [comm] for $f(2, 1)$.

⁶This set implicitly contains the rules $\vdash Ax$ for any axiom Ax.

We further assume that each formulae expressible in a theory has a normal form which can be reached by a finite number of applying the deduction rules. Recalling Example 2.4.3, we obtain that $\mathcal{K}, T^= \vdash \text{tower}(3, 2, 1) = \text{on}(3, 2) \wedge \text{on}(2, 1) \wedge \text{on}(1, 0)$ by applying twice the first axiom for *tower*. We call this the normal or *irreducible* form of *tower*(3, 2, 1) and we use the notation \downarrow to denote it.

With the above considerations, we can describe how to extend the matching problem for solving queries against belief bases which we detailed in Section 2.1.3 by reasoning *modulo* knowledge bases. We write $\mathcal{B}, \bigcup_i \mathcal{K}_i \vdash \phi$ to say that the query ϕ has a solution in \mathcal{B} modulo $\bigcup_i \mathcal{K}_i$. Formally, we have the following definition:

$$\mathcal{B} \bigcup_i \mathcal{K}_i \vdash \phi \quad \text{iff} \quad \bigcup_i \mathcal{K}_i, T^= \vdash \phi = \phi \downarrow \text{ and } \phi \downarrow \leq^? \mathcal{B} \quad (2)$$

which says that ϕ can be answered by a matcher in \mathcal{B} if and only if there exists a normal form $\phi \downarrow$ of ϕ which can be deduced in a combination of knowledge bases where we implicitly consider the theory of equality. We note that we do not need to explicitly add \bar{B} in line (2) because of the way we implemented $\phi \downarrow \leq^? \mathcal{B}$ in Section 2.1.3.

The immediate question is whether deducing $\phi = \phi \downarrow$ is a complete procedure in a union of theories. To this we can answer by referring to the results from theorem proving and automatic reasoning which state that completeness is guaranteed under requirements as convexity of the theories. This is what is referred as the Nelson-Oppen method [NO79]. We also mention [Nip89] as a reference for decidability results under regular theories. As a short observation, these works are the basis of SAT solvers modulo theories. Thus in practice, the validity of $\exists \phi$ can be answered by feeding it to a SMT solver. To name but a few we refer to Yices and Barcelogic [NORCR07].

With the new definition of matching modulo knowledge bases, the semantics of BUnity^K is given by straightforwardly modifying the transition rule (*act*) from Definition 2.2.1 as it can be seen in Figure 2.7. To simplify notation, we use $\theta \in \text{Sols}_{\mathcal{K}}(\mathcal{B}, (\phi \wedge \psi \wedge \text{Cond}))$ in place of “ $\bigcup_i \mathcal{K}_i, T^= \vdash \phi = \phi \downarrow$ and $\theta \in \text{Sols}(\mathcal{B}, (\phi \wedge \psi) \downarrow \wedge \text{Cond})$ ”.

$$\boxed{\frac{\begin{array}{l} \phi \triangleright \text{do}(a) \in \mathcal{A}^t \quad a = (\psi, \xi) \text{ if } \text{Cond} \in \mathcal{A}^b \\ \theta \in \text{Sols}_{\mathcal{K}}(\mathcal{B}, (\phi \wedge \psi \wedge \text{Cond})) \quad \perp \notin \xi \theta \end{array}}{\mathcal{B} \xrightarrow{a\theta} \mathcal{B} \uplus \xi \theta} \quad (\text{act})}$$

Figure 2.7: The Transition Rule for BUnity^K

The fact that we can easily generalise the semantics shows the flexibility of our approach. Furthermore, we note that the above described mechanism is the best we can do without breaking the validity of the updating operation. This is because we only add structure to the reasoning process, leaving the updating operation in itself unchanged. We recall that knowledge bases are immutable. The main argument against allowing the modification of the theories representing knowledge bases can be seen more easily by means of an example. We consider a knowledge base represented by Presburger theory. We have that deleting an axiom would result in losing completeness while adding some new "axiom" would result in

losing soundness. One might, however, come with a counterargument to our choice of thinking of knowledge bases as being immutable. The counterargument could possibly consist of the following example. We consider a belief base $\mathcal{B} = \{p(a), q(a)\}$, a knowledge base $\mathcal{K} = \{q(x) \rightarrow p(x)\}$ with only one axiom saying that if $q(x)$ holds then so does $p(x)$, and an action $act(x) = (q(x), \{\neg p(x)\})$. Executing $act(a)$ results in deleting $p(a)$. However, the query $q(x)$ can still provide a solution that $p(a)$ holds and this might be counterintuitive. To this we answer that the implementation of act may be seen as erroneous since it asks to delete an element which can be deduced from the knowledge base. Thus, the implementations of actions should take into account the information provided by knowledge bases.

We conclude the section by making the short observation that a more expressive agent implementation language can be obtained by simply merging BUnity^K and BUpL into BUpL^K and this is what illustrates Figure 2.8, where CR (resp. DR) stands for control (resp. data) refinement. However, for simplicity, in this thesis we mainly focus on the relation between BUnity and BUpL .

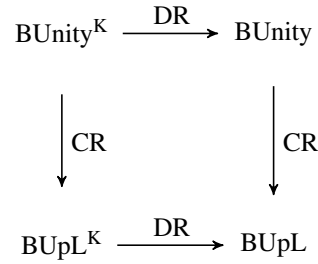


Figure 2.8: A Diagram of Agent Languages Refinement

2.5 An Implicit Modelling of Goals

We have deliberately cast aside *goals* in BUnity and BUpL . This is mainly for simplicity reasons. The usual way ([dBHvdHM07, DvRDM03]) to explicitly incorporate goals is to fix a particular representation, for example, as a conjunction of ground atoms (which we might understand as a special case of a belief base). The corresponding change in the semantics is to extend the queries of BUnity triggers and of BUpL repair rules such that they do not interrogate only belief bases but also goals. Additionally, plan calls should be extended such that goals can trigger plan executions.

Given that our focus is on verification, being able to represent goals implicitly is acceptable enough in our framework. Furthermore, the expressive power of the languages is not necessarily decreased. We can, without changing the syntax and the semantics of the languages, have a declarative modelling of goals as LTL formulae. In such a situation, we would be interested in any agent execution which satisfies a given goal. This problem can be equally

stated as a reachability problem and the answer can be provided by verification. More precisely, model-checking the negation of the goal returns, in fact, a counter-example denoting a successful trace (leading to the achievement of the goal) in the case that there exists one.

For example, we can define, with respect to the scenario introduced in the previous sections, the LTL predicates

$$\begin{aligned} goal_1 &= \Diamond fact(cleaned) \\ goal_2 &= \Diamond (fact(on(1,0)) \wedge fact(on(2,1)) \wedge fact(on(3,2))) \end{aligned}$$

where *fact* is a predicate defined on the mental states of either BUnity or BUpL agents in the following way:

$$(\mathcal{B}, p) \models fact(P) \text{ iff } \mathcal{B} \models P.$$

Model-checking that the property $\neg (goal_1 \wedge goal_2)$ holds in a state reachable from the initial one returns a counterexample representing the minimal trace *clean move(3,0) move(2,1) move(3,2)*. This execution leads to a state where both *goal*₁ and *goal*₂ are satisfied.

2.6 Prototyping BUpL as Rewrite Theories

So far we described a methodology of designing agent languages. In this section we describe rewriting logic and we show why it is useful for prototyping agent languages. To anticipate, the main advantages are that prototyping is easy and that we can both *execute* and *verify* agent programs.

Rewriting logic [Mes92] is a suitable *computational* framework for specifying, experimenting and developing with domain-specific languages that have operational semantics. This is mainly thanks to the fact that operational semantics can be faithfully captured as rewriting logic specifications [MOM00b, SRM09]. More precisely, there is a correspondence between operational steps in the language definition (i.e., with respect to the transitions giving the semantics of the language) and computational steps in the corresponding rewriting logic specification. This result presents some important advantages over the typical situation when one chooses to implement the designed language using a standard programming language like Java, for example. In this case, not only is it non-trivial to build a Java interpreter, often requiring the use of specific hacks, but also, this process is creating a “gap between the formal operational semantics of the language and its implementation”. Moreover, with respect to implementing nondeterministic languages in Java, one needs to basically dedicate/-commit oneself to a particular choice of scheduler, thus breaking the initial nondeterminism. On the contrary, using rewriting logic, all of the above difficulties are easily handled. Given that *rewritings are computations*, one implication of this agreement between operational and rewriting semantics is the fact that language definitions can be directly executed. From this one benefits of the possibility of *rapid* prototyping, in the sense that the language designer can experiment with the language definitions by only changing the definitions and thus avoiding spending time on implementation details. Thanks to this aspect of rewriting logic, the correctness between definitions and implementations is immediate. Furthermore, rewriting logic is intrinsically nondeterministic, thus ideal for modelling nondeterministic languages

like BUpL. In what comes next, we illustrate the above affirmations by effectively encoding BUpL as a rewrite theory. In doing so, we follow closely the methodology from [SRM09].

A rewrite theory is a triple (Σ, E, R) with Σ a sorted signature of functional symbols, E a set of (possibly conditional) Σ -equations and R a set of Σ -rewrite rules. (Σ, E) form the so-called *sorted equational theory* [Mes97]. Rewrite rules can be conditional, thus their general form is:

$$l : t \rightarrow t' \text{ if } (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

It basically says that l is the label of the rewrite rule $t \rightarrow t'$ which is used to “rewrite” the term t to t' when the conditions on t are satisfied. Such conditions can be either equations like $u_i = v_i$, memberships like $w_j : s_j$ (that is, w_j is of sort s_j) or other rewrites like $p_k \rightarrow q_k$. Alternatively, conditional rewrite rules can be put in the format of inference rules:

$$\frac{\bigwedge_i u_i = v_i \quad \bigwedge_j w_j : s_j \quad \bigwedge_k p_k \rightarrow q_k}{t \rightarrow t'} \quad (l)$$

and this makes more transparent the connection with transition rules in transition systems. To have the complete picture, rewrite theories axiomatise transition systems whose states are ground terms in the initial algebra $T_{\Sigma/E}$ and whose transitions are specified by the rules from R . The inference system of rewriting logic which we reproduce in Figure 2.9 allow us to derive as proofs all the possible computations in transition systems. Given a rewrite theory \mathcal{R}^7 , the sentences that \mathcal{R} proves are universally quantified rewrites of the form $(\forall X)(t \rightarrow t')$, with $t, t' \in T_{\Sigma}(X)$ obtained by a finite application of the deduction rules.

For $\mathcal{R} = (\Sigma, E, R)$, the notation $\mathcal{R} \vdash t \rightarrow t'$ states that the sequent $t \rightarrow t'$ is provable in \mathcal{R} using the inference rules from Figure 2.9. Intuitively, these could be understood as constructions of computations in the transition system specified by \mathcal{R} . The rule **Reflexivity** enables idling in a state with no change whatsoever. The rule **Equality** specifies that states are classes modulo the equations from E . The rule **Congruence** allows that the arguments of f to rewrite in parallel. In the rule **Replacement** θ is a substitution $\theta : X \rightarrow T_{\Sigma}(Y)$ and θ' is the substitution obtained from θ by some rewritings of each variable x in X . The rule **Transitivity** enables the derivation of longer computations by composing them sequentially.

We draw attention that $\mathcal{R} \vdash t \rightarrow t'$ does not necessarily represent an *atomic* step since it may contain complex computations. This means that rewriting logic does not have a built-in “one-step” rewrite relation. To refer precisely to one-step rewrites, we use the notation \rightarrow^1 which denotes a particular case of rewriting ground terms when **Transitivity** is disabled and when there is at least one application of **Replacement**.

To encode BUpL as a rewrite theory $\mathcal{R}_{BUpL} = (\Sigma_{BUpL}, E_{BUpL}, R_{BUpL})$, we need to consider the syntax and the semantics. To encode the BUpL syntax we make use of *order-sorted signature* [GM87]. An order-sorted signature is a sorted signature with additional subsort declarations in the form $s < s'$, with s, s' being sorts, to denote that s is a *subsort* of s' .

⁷The reader might notice that we use the same symbol as for the sets of repair rules. However, it will be clear from the context what \mathcal{R} refers to, thus there is no danger of confusion.

Reflexivity:	$\frac{}{(\forall X)t \rightarrow t}$
Equality:	$\frac{(\forall X)t \rightarrow t' \quad E \vdash (\forall X)t = u \quad E \vdash (\forall X)t' = u'}{(\forall X)u \rightarrow u'}$
Congruence:	$\frac{(\forall X)t_1 \rightarrow t'_1 \quad \dots \quad (\forall X)t_n \rightarrow t'_n}{(\forall X)f(t_1, \dots, t_n) \rightarrow f(t'_1, \dots, t'_n)} \quad \text{if } f \in F_n$
Replacement:	$\frac{\bigwedge_x (\forall Y)\theta(x) \rightarrow \theta'(x) \quad (\forall Y)C\theta}{(\forall Y)t\theta \rightarrow t'\theta'} \quad \text{if } r : (\forall Y)t \rightarrow t' \text{ if } C \in R$
Transitivity:	$\frac{(\forall X)t \rightarrow t' \quad (\forall X)t' \rightarrow t''}{(\forall X)t \rightarrow t''}$

Figure 2.9: The rules for rewriting logic deduction

The BUpL syntax, in its BNF form, can be seen as an order-sorted signature Σ_{BUpL} having one operation definition per production, terminals giving the name of the operation and non-terminals the arity. For example, the productions defining plans as sequential compositions (respectively sums) can be seen as an algebraic operation which is associative and has *nil* unit element (respectively also commutative):

$$\begin{aligned} _ ; _ & : \text{Action} \times \text{Plan} \rightarrow \text{Plan} & [\text{assoc id: nil}] \\ _ + _ & : \text{Plan} \times \text{Plan} \rightarrow \text{Plan} & [\text{assoc comm id: nil}] \end{aligned}$$

where *Action*, *Plan* are the sorts (corresponding to BUpL non-terminals) in K_{BUpL} and “+”, “;” are operators in Σ_{BUpL} (corresponding to BUpL terminals). Further, we have relations between sorts, i.e., subsorts, like *Action* < *Plan*. This is because any action is a plan.

The sorted signature Σ_{BUpL} contains also sorts for beliefs and belief bases, denoted *Belief* and respectively *BeliefBase*. There is a natural subsort relation between them, *Belief* < *BeliefBase*, saying that every belief is a belief base. Similarly, Σ_{BUpL} contains the sorts *Literal* and *Literals* with the subsort relation *Literal* < *Literals*. Since a belief is a positive literal we have the additional subsort relation *Belief* < *Literal*. Further, to construct sets of literals we use the operator “,” in mixfix notation:

$$_,_ : \text{Literal} \times \text{Literals} \rightarrow \text{Literals} \quad [\text{assoc comm id: empty}]$$

Thanks to the subsort relation *Belief* < *Literal* we can overload “,” to construct also belief bases.

Besides syntax, there are a couple of BUpL operations with a fixed definition. We make the distinction between two levels of definitions. Updating belief bases, matching queries and checking the consistency of effects are at the level of the language. We call them language

dependent or BUpL *native*. Matching terms and applying substitutions are, on the contrary, at a meta-language level. They are independent of the language, in the sense they belong to the underlying logical framework of BUpL. In this section we consider only the language dependent ones and we leave the more detailed discussion of the meta-level ones for the next section.

The BUpL native operations are organised in E_{BUpL} . We discuss each of them in turn. The update \uplus maps to $update : BeliefBase \times Literals \rightarrow BeliefBase$ which is equationally defined as follows:

$$\begin{aligned} update(B, empty) &= B \\ update(B, (L, Ls)) &= \text{if } neg(L) \text{ then } update(remove(B, L), Ls) \\ &\quad \text{else } update(add(B, L), Ls) \text{ fi} \end{aligned}$$

where B , L and Ls are variables of sorts *BeliefBase*, *Literal* and respectively *Literals*. The correspondence with the definition of \uplus from Section 2.1 is immediate.

The other operation on beliefs is $match : BeliefBase \times Query \rightarrow Substitution$. Describing this operation is a good moment to introduce the notion of *reflection*, an inherent characteristic of rewriting logic. A reflective logic is a logic in which important aspects of its meta-theory can be represented at the object level in a consistent way, so that object-level representation correctly simulates the relevant meta-theoretic aspects [Cla00a]. Using reflection, we have a clear separation between the theory representing the prototype of BUpL language and the one representing BUpL agents. We assume that at the meta-level we have a functionality $metaMatch$ which we describe in the next section. We further consider the relation $Belief < Term$ to say that any belief is a term. This enables us to simply define the basic case of the match operation as follows:

$$match((Bel, B), Bel'^8) = metaMatch(\overline{T}, \overline{Bel}, \overline{Bel'})$$

where $\overline{\mathcal{R}}$ denotes the meta-representation of the theory where the agent itself is defined and \overline{Bel} , $\overline{Bel'}$ are the representations of the variables Bel , Bel' at the meta-level. As we show in Section 2.6.2, the functionality $metaMatch$ is precisely an implementation of $\leq^?$. In particular, $match((Bel, B), Bel')$ defined as above represents the basic case $l \leq^? a_n$ from Figure 2.1 where l should be read as Bel and a_n as Bel' . Along the lines of the procedure from Figure 2.1, the other cases follow easily from a case analysis on the inductive structure of queries.

We note that $metaMatch$ is not compulsory however it eases the burden of implementing $match$. If it were to not use $metaMatch$ we would have had to provide this facility at the language level. This implies that the underlying logical framework of any agent language would have had to be piggybacked on each individual language, thus flattening the so-called meta-level at object-level. At a finer grain of detail, another consequence is that we would have had to precisely define what terms are, thus we would have had to fix a priori the set of all functional symbols that might appear in beliefs. Besides being elegant, the separation between dependent and independent language functionalities is also more practical as it allows re-usability and modularity.

Checking the consistency of effects, i.e., that $\perp \not\in \xi\theta$ is true, is done by means of a predicate, i.e., a function with boolean values:

⁸To simplify, we consider the subsort relation $Belief < Query$ to say that a belief can be seen as a basic query.

$$\text{consistent}((L, \text{neg}(L), Ls)) = \text{false}.$$

We remark that the above equation makes use of pattern-matching modulo ACI, since the constructor “;” for *Literals* is ACI. There is no need to define *consistent* totally since we are only interested in $\text{consistent}(Ls) \neq \text{false}$, thus in this case we benefit from the possibility of having partial function definitions in equational theories.

The operational semantics given by the BUP_L transition rules maps naturally to rewrite rules. For example, the transition (*act*) from Figure 2.5:

$$\frac{a = (\psi, \xi) \text{ if } \text{Cond} \in \mathcal{A} \quad \theta \in \text{Sols}(\mathcal{B}, \psi \wedge \text{Cond}) \quad \perp \not\in \xi \theta}{(\mathcal{B}, a; p') \xrightarrow{a\theta} (\mathcal{B} \uplus \xi \theta, p' \theta)} \quad (\text{act})$$

maps to the following rewrite rule:

$$\begin{aligned} \text{act} \quad : \quad \langle \mathcal{B}, a; p' \rangle &\rightarrow \langle \text{update}(\mathcal{B}, E), \text{metaSubstitute}(p', \theta) \rangle \\ &\text{if } \theta := \text{match}(\mathcal{B}, \text{pre}(a)) \wedge \theta \neq \text{noMatch} \\ &\wedge E := \text{metaSubstitute}(\text{post}(a), \theta) \wedge \text{consistent}(E) \end{aligned}$$

where $\langle \mathcal{B}, p \rangle$ is the term of sort *BpState* corresponding to the state (\mathcal{B}, p) . Though we could have used the same notation, we explicitly use $\langle \mathcal{B}, p \rangle$ instead to make it clearer when we work with transition systems and when with the corresponding rewriting theories. This is only in this chapter, elsewhere we will use the same notation for simplicity.

Though using the same format as the transition rule (*act*), the rewriting rule *act* differs in the following aspects. First, there is no explicit labelling. This can be achieved by means of an intermediary configuration but we do not give the details here since it is not important for the moment. We will, however, discuss the construction when we need it, in Section 3.1. Second, there is no special handling of the equality-based condition *Cond*. This is thanks to the equational deduction underlying rewriting logic. Third, the rule makes use of additional functions. Two of them, *pre*, *post*, are implemented at object-level with the purpose of returning the pre- (respectively, the post-) condition of *a*. The other one is the meta-level functionality *metaSubstitute* for applying substitutions. Roughly, we use a meta-level functionality in order to avoid an explicit implementation of the underlying logical language of the agent languages. We discuss this matter more thoroughly in Section 2.6.2.

To illustrate the usefulness of memberships, we describe an adaptation of *act* to a particular case of observable actions is:

$$\begin{aligned} o\text{-act} \quad : \quad \langle \mathcal{B}, a; p' \rangle &\rightarrow \langle \text{update}(\mathcal{B}, E), \text{metaSubstitute}(p', \theta) \rangle \\ &\text{if } \theta := \text{match}(\mathcal{B}, \text{pre}(a)) \wedge \theta \neq \text{noMatch} \\ &\wedge E := \text{metaSubstitute}(\text{post}(a), \theta) \wedge \text{consistent}(E) \\ &\wedge a : A^o \end{aligned}$$

where A^o denotes the sort of observable actions. As it will be clear in the next sections, we need the distinction between internal and observable actions for testing, in order to have a more expressive framework.

All the other transition rules are encoded as rewrite rules in a similar manner and we do not further explain them. In what follows, we only need to remember that each transition has a corresponding rewrite rule labelled with the same name.

It is not difficult to formally state the correspondence between the operational semantics for BUpL and the associated rewrite theory. This is precisely in the sense that any BUpL execution trace is in a one-to-one mapping with a computation in the associated rewrite theory which we denote by \mathcal{R}_{BUpL} . Equally stated, BUpL has executable semantics.

2.6.1. PROPOSITION. $SOS_{BUpL} \vdash \langle \mathcal{B}, p \rangle \rightarrow^* \langle \mathcal{B}', p' \rangle$ iff $\mathcal{R}_{BUpL} \vdash \langle \mathcal{B}, p \rangle \rightarrow \langle \mathcal{B}', p' \rangle$.

Proof. By induction on the derivation trees. ■

Proposition 2.6.1 states that rewritings using \mathcal{R}_{BUpL} (together with rewriting deduction) and BUpL computations coincide, thus we have *executable* BUpL semantics almost for free.

Having defined BUpL as a rewrite theory $\mathcal{R}_{BUpL} = (\Sigma_{BUpL}, E_{BUpL}, R_{BUpL})$, the only related matter which needs to be addressed is “how to prototype BUpL agents”. We explain this by referring to the BUpL example from Figure 2.6. There we see a set of assignments denoting the initial configuration of the agent. These assignments map in a straightforward way as equations in an equational theory $T_{agent} = (\Sigma_{agent}, E_{agent})$. We depict it in Figure 2.10. The only extra definition we need to add to T_{agent} is $T = 'T_{agent}$. This is because the matching described above relies on *metaMatch* which has \bar{T} as a parameter denoting the meta-representation of the theory where the matching should take place.

$$\begin{aligned}
 \Sigma_{agent} = & \\
 & on : \mathbb{N} \times \mathbb{N} \rightarrow Pred \\
 & B_0 : \rightarrow BeliefBase \\
 & move : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow A^o \\
 & p_0 : \rightarrow Plan \\
 & r : \rightarrow RepairRule \\
 E_{agent} = & \\
 & B_0 = (on(3,1), on(1,0), on(2,0), clear(3), clear(2), clear(0)), \\
 & move(x,y,0) = (on(x,y) \wedge clear(x), \\
 & \quad (on(x,z), neg(on(x,y)), clear(y))), \\
 & move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \\
 & \quad (on(x,z), \neg on(x,y), \neg clear(z), clear(y))) \text{ if } z \neq 0 \wedge y \neq z, \\
 & p_0 = move(2, 0, 1); move(3, 0, 2), \\
 & r = (on(x,y) \leftarrow move(x,y,0); p_0) \\
 & T = 'T_{agent}
 \end{aligned}$$

Figure 2.10: The Equational Theory of the BUpL Toy Example

2.6.1 Executing Agents by Rewriting

Proposition 2.6.1 shows that the BUpL semantics are executable. In this section we illustrate in turn what it means to execute an agent program via rewriting. We consider an initial configuration like $\langle B_0, p_0 \rangle$ with B_0, p_0 being the ones defined in Figure 2.10. We can rewrite

$\langle B_0, p_0 \rangle$ with the rules from R_{BUPL} . As an example of a execution we consider the following rewritings:

$$\langle B_0, p_0 \rangle \rightarrow^1 \langle B_0, \text{move}(3, 1, 0); p_0 \rangle \rightarrow^1 \langle B_1, p_0 \rangle$$

where B_1 is the placeholder of:

$$(\text{on}(3, 0), \text{on}(1, 0), \text{on}(2, 0), \text{clear}(2), \text{clear}(0), \text{clear}(1))).$$

We describe the rewriting logic deduction behind in more detail. First, the rule **Replacement** is applied with r being the label (*fail*), C being the condition of (*fail*), that is, the existence of an enabled repair rule in T_{agent} , and θ being $[B/B_0][p/p_0][x/3][y/1]$ such that $C\theta$ holds. Next, the rule **Replacement** is applied with r being the label (*act*), C being the condition of (*act*), and θ being $[B/B_0][a/\text{move}(3, 1, 0)][p'/p][E/(\text{on}(3, 0), \text{clear}(1), \text{neg}(\text{on}(3, 1)))]$. Finally, the rule **Equality** is applied 3 times by using the equalities:

$$\begin{aligned} & \text{update}(B_0, (\text{on}(3, 0), \text{clear}(1), \text{neg}(\text{on}(3, 1)))) = \\ & \text{update}(\text{add}(B_0, \text{on}(3, 0)), (\text{clear}(1), \text{neg}(\text{on}(3, 1)))) = \\ & \text{update}(\text{add}(\text{add}(B_0, \text{on}(3, 0)), \text{clear}(1)), (\text{neg}(\text{on}(3, 1)))) = \\ & \text{remove}(\text{add}(\text{add}(B_0, \text{on}(3, 0)), \text{clear}(1)), \text{on}(3, 1)). \end{aligned}$$

From the latter, by applying 3 more times the rule **Equality**, we obtain the normal form B_1 .

2.6.2 A Taste of Rewriting as a Metalanguage

Reflection is a powerful and useful feature which makes it possible to “meta-reason” about languages represented in a logical language in the *same* logical framework. As it is also the case with equational logics and Horn logic, rewriting logic is reflective. The corresponding reflection theorems for these logics are the subject of [CMP07]. Rewriting logic is reflective precisely in that there can be defined a universal rewrite theory U and a representation function $(_ \vdash _)$ such that for any statement ϕ true in a rewrite theory \mathcal{R} its meta-representation is true in U :

$$\mathcal{R} \vdash \phi \text{ iff } U \vdash \overline{(\mathcal{R} \vdash \phi)}.$$

In the same way there could be defined \bar{U} such that U can simulate its own meta-level at the object-level, and this process can be iterated ad infinitum in a so-called “reflective tower” [Cla00b]:

$$\mathcal{R} \vdash \phi \text{ iff } U \vdash \overline{(\mathcal{R} \vdash \phi)} \text{ iff } U \vdash U \vdash \overline{\overline{(\mathcal{R} \vdash \phi)}} \text{ iff } \dots$$

Having an explicit specification of universal theories is of great importance since they can serve as a foundation which provides built-in facilities for meta-reasoning. Among these facilities we are interested mainly in using meta constructions at object level. To illustrate this, we first need to describe a few basic concepts from the meta-theory of rewriting logic. More precisely, we focus on those particular aspects of an universal meta-theory which make it possible (1) to directly represent elements from the object level as meta-terms, (2) to use meta-level operations like matching at object-level, and (3) to control the execution at the object-level by instrumenting rewrite rules at meta-level. The meta-theory we refer to is the one

providing the main functionality of the universal theory U from [Cla00b, CM02, CMP07]. For the ease of reference we denote it by *META-THEORY* and we adapt it following the presentation from [CDE⁺07]. In *META-THEORY* terms are meta-represented by elements of sort *Term*. The base cases are given by the subsorts *Constant* and *Variable* of the sort *Qid* which stands for quoted identifiers⁹. We assume constants (respectively variables) are quoted identifiers that contain the name and the sort of the constant (respectively variable) separated by ‘.’ (respectively ‘:’). For example, ‘*Bel.Belief*’ (respectively ‘*VBel:Belief*’) denotes the meta-representation of a constant *Bel* (respectively a variable *VBel*) of sort *Belief*. The inductive case defining terms is as follows:

$$\begin{array}{llll} _[_] & : & Qid \times TermList & \rightarrow Term & [ctor] \\ _[-] & : & TermList \times TermList & \rightarrow TermList & [ctor \text{ assoc id} : empty] \end{array}$$

where *TermList* denotes list of terms and the concatenation operator is declared as being associative with unit element the empty list and a constructor.

In a similar way, theories are meta-represented as elements of sort *Theory*. Their definition is easy, however longer, as it includes the meta-representation of all composing (sub)-elements of equations and rules. In what follows, we only consider that there is a so-called *descent* function *upTheory* which takes as parameter a rewrite theory \mathcal{R} and returns its meta-representation. To make notation shorter, we denote *upTheory*(\mathcal{R}) by $\overline{\mathcal{R}}$. Descent functions enable us to move between reflection levels, that is, to obtain the meta-representation of an element at object-level and vice-versa. For example, given the meta-theory $\overline{\mathcal{R}}$, we can compute the meta-representation of a particular term using the descent function *upTerm*:

$$\begin{array}{lll} upTerm(\overline{\mathcal{R}}, c) & = & 'c.Sort \quad \text{if } c \text{ is a constant of sort } Sort \text{ in } \overline{\mathcal{R}} \\ upTerm(\overline{\mathcal{R}}, v) & = & 'v : Sort \quad \text{if } v \text{ is a variable of sort } Sort \text{ in } \overline{\mathcal{R}} \\ upTerm(\overline{\mathcal{R}}, f(tl)) & = & 'f[upTerm(\overline{\mathcal{R}}, tl)] \quad \text{if } f \text{ is an operator in } \overline{\mathcal{R}}. \end{array}$$

When $\overline{\mathcal{R}}$ is clear from the context we will use the alternative notation \bar{t} to stand for *upTerm*($\overline{\mathcal{R}}, t$). We note that we use the terms “constant” and “variable” only to make the description more intuitive. Actually, at the meta-level, that is, in *META-THEORY*, where all meta-representations $\overline{\mathcal{R}}$ are contained, there is no distinction between constants and variables since both are represented by identifiers, with the only difference between them consisting in using “.*<Sort>*” for denoting constants and “: *<Sort>*” for denoting variables. The inverse of *upTerm* the descent function *down* which returns from a meta-term \bar{t} the term t at object-level.

To manipulate meta-representation of terms we use the so-called *meta-functions*. We consider for example the meta-function *metaApply*. This function reifies the process of applying a rewrite rule. More precisely, given a meta-representation of a theory $\overline{\mathcal{R}}$, of a term \bar{t} , of a rule name \bar{l} and of a substitution $\bar{\theta}$, *metaApply*($\overline{\mathcal{R}}, \bar{t}, \bar{l}, \bar{\theta}$) is implemented as follows:

$$\begin{aligned} metaApply(\overline{\mathcal{R}}, \bar{t}, \bar{l}, \bar{\theta}) &= metaSubstitute(\bar{t}_2, \bar{\theta}_1) \\ &\text{if } (\bar{l} : \bar{t}_1 \rightarrow \bar{t}_2) \in R \wedge \\ &\quad \bar{\theta}_1 := metaMatch(\overline{\mathcal{R}}, \bar{t}, \bar{l}, \bar{\theta}) \end{aligned}$$

⁹We assume that *Qid* provides a construction $\langle Qids \rangle$ to denote sets of constants without being algebraically constructed

where R represents the rewrite rules from the meta-theory $\overline{\mathcal{R}}$. To explain it in more detail, *metaApply* consists of:

1. normalising \bar{t} using the equations from $\overline{\mathcal{R}}$;
2. matching the normalised \bar{t} against all rules $\bar{t}_1 \rightarrow \bar{t}_2$ identified by \bar{l} partially instantiated by $\bar{\theta}$; and
3. returning the term resulting from the application of the substitution $\overline{\theta}_1$ (from the matching from (2)) on \bar{t}_2 .

The functions *metaSubstitute*, *metaReduce*, and *metaMatch* are discussed below. The function *metaSubstitute* applies the meta-representation of a substitution on meta-terms in the usual inductive way:

$$\begin{aligned} \text{metaSubstitute}(\bar{t}, \text{none}) &= \bar{t} \\ \text{metaSubstitute}('c.\text{Sort}, \bar{\theta}) &= 'c.\text{Sort} \\ \text{metaSubstitute}('v : \text{Sort}, ('v : \text{Sort} \leftarrow \bar{t})) &= \bar{t} \\ \text{metaSubstitute}('f[\bar{t}], \bar{\theta}) &= 'f[\text{metaSubstitute}(\bar{t}, \bar{\theta})] \end{aligned}$$

where *none* denotes the empty substitution and $('v : \text{Sort} \leftarrow \bar{t})$ is the meta-representation of the substitution $[v/t]$.

The function *metaReduce* uses the equations from $\overline{\mathcal{R}}$ to obtain the form of a term which can no longer be simplified:

$$\text{metaReduce}(\overline{\mathcal{R}}, \bar{t}) = \begin{cases} \text{if } (\bar{t} = \bar{t}' \in E \wedge \bar{\theta} := \text{metaMatch}(\overline{\mathcal{R}}, \bar{t}, \bar{t}')) \\ \text{then } \text{metaReduce}(\overline{\mathcal{R}}, \text{metaSubstitute}(\bar{t}', \bar{\theta})) \text{ else } \bar{t} \text{ fi.} \end{cases}$$

where E represents the equations from the meta-theory $\overline{\mathcal{R}}$. If E is Church-Rosser (confluent) and terminating then the computation is guaranteed to terminate.

As it is the case with the usual term matching algorithm which we referred to by using the notation “ $\leq?$ ” in Section 2.1, the definition of *metaMatch* follows from a case analysis, however, for meta-terms:

$$\text{metaMatch}(\overline{\mathcal{R}}, \bar{t}_1, \bar{t}_2) = \begin{cases} c_1 ==_{\overline{\mathcal{R}}} c_2 & \text{if } \bigwedge_{i \in \{1,2\}} \bar{t}_i = 'c_i.\text{Sort} \\ ('v : \text{Sort} \leftarrow \bar{t}) & \text{if } \bar{t}_1 = 'v : \text{Sort} \\ \text{metaMatch}(\overline{\mathcal{R}}, \bar{t}_{l_1}, \bar{t}_{l_2}) & \text{if } \bigwedge_{i \in \{1,2\}} \bar{t}_i = 'f[t_{l_i}] \\ \text{noMatch,} & \text{otherwise} \end{cases}$$

The above definitions of *metaReduce* and *metaApply* rely on the existence of a substitution $\bar{\theta}$, thus they only provide a partial characterisation. To make it total, it suffices to define their values as being a constant term *failure* for the cases when *metaMatch* returns *noMatch*.

The reflection theorems which apply to the universal theory allows us to have the following equivalences ensuring the correctness of further using the meta-functions defined above:

2.6.2. PROPOSITION. *Let \mathcal{R} be a theory, t and t' be terms, and θ be a substitution. We have that the following equivalences hold:*

- $\text{META-THEORY} \vdash \text{metaReduce}(\overline{\mathcal{R}}, \bar{t}) = \bar{t}' \text{ iff } \mathcal{R} \vdash t = t'$
- $\text{META-THEORY} \vdash \text{metaMatch}(\overline{\mathcal{R}}, \bar{t}, \bar{t}') = \bar{\theta} \text{ iff } \mathcal{R} \vdash t = t'\theta.$

2.6.3 A Strategy Language

The meta-level functionalities are the main building blocks needed to define rewrite strategies languages in a declarative manner. However, the details of the meta-level functionalities which are used in the definition of strategy are not necessary to follow this section, it suffices to assume their existence. The section can be also read without the details of the implementation of the strategies themselves. What we want, in the end, to stress, is the *form* of the strategy expressions. This is what is important to remember to understand Section 3.4 where we make more heavy use of strategies.

The main advantage of designing a strategy language is that it facilitates the use of meta-level functionalities at object level. The strategy language we refer to has been introduced in [EMOMV07]. For brevity we denote the strategy language by S . We briefly describe its formalisation in what follows and refer to [MOMV09] as the underlying source of the description. We consider $\mathcal{R} = (\Sigma, E, R)$ to be a rewrite theory and a term t which can be rewritten by the rewrite rules from R . Given a strategy expression s in S , the application of s to t is denoted by $s@t$. The semantics of $s@t$ is the set of successors which result by rewriting t in the associated rewrite theory \mathcal{R}_S . More precisely, “@” is a function $_{@} : S \times T_{\Sigma}(X) \rightarrow 2^{T_{\Sigma}(X)}$ which is extended to $_{@@} : S \times 2^{T_{\Sigma}(X)} \rightarrow 2^{T_{\Sigma}(X)}$ with $T_{\Sigma}(X)$ being the set of terms in the rewrite theory \mathcal{R} , thus at the object-level. The following assumption is made. For any term w such that $\mathcal{R}_S \vdash s@t \rightarrow^* w$ one can define a function *sols* to denote the set of *solution terms* already computed by the strategy. Namely, t' is in *sols*(w) if t' is an intermediary term reached from t to w .

Naturally, S is defined such that there is a correspondence between rewrites in \mathcal{R} and rewrites in \mathcal{R}_S :

- **Soundness.** If $\mathcal{R}_S \vdash s@t \rightarrow^* w$ and $t' \in \text{sols}(w)$, then $\mathcal{R} \vdash t \rightarrow^* t'$.
- **Completeness.** If $\mathcal{R} \vdash t \rightarrow^* t'$ then there exists a strategy $s \in \mathcal{R}_S$ and a term w such that $\mathcal{R}_S \vdash s@t \rightarrow^* w$ and $t' \in \text{sols}(w)$.

The simplest strategies we can define in S are the constants *idle* and *fail*: *idle* @ $t = \{t\}$, *fail* @ $t = \emptyset$. Another basic strategy consists of applying to t a rule from \mathcal{R} identified by a label l , $l@t$, possibly with instantiating some variables appearing in the rule, $l[x/t']@t$, with x being a variable in the rule identified by l and t' a term. The semantics of $l[x/t']@t$ is the set of all terms to which t rewrites in one step using the rule labelled l where x is substituted by t' . To connect to the meta-level functionalities described above, we note that the underlying implementation of $l[x/t']@t$ is based on *metaApply*, namely:

$$\begin{aligned} l[x/t']@t_0 &= \{t'' \mid t'' := \text{down}(\text{metaApply}(\overline{\mathcal{R}}, \overline{t_0}, \overline{l}, \overline{[x/t']s})) \wedge t'' \neq \text{failure} \\ &\quad \wedge (\overline{l} : \overline{t} \rightarrow \overline{t'} \in R \\ &\quad \wedge \overline{s} := \text{metaXmatch}(\overline{\mathcal{R}}, \text{metaSubstitute}(\overline{t}, \overline{[x/t']}), \overline{t_0}) \} \end{aligned}$$

where *metaXmatch*($\overline{\mathcal{R}}, \overline{t_1}, \overline{t_2}$) is an extension of *metaMatch* which returns a substitution when $\overline{t_1}$ is successfully matched against of a subterm of $\overline{t_2}$. Since there might be more subterms with successful matchings, the application of the strategy l returns sets of terms. Since meta-functions return meta-representations, we use the descent function *down* to return the terms corresponding to the object level.

Another basic strategy is a test *match* t' s.t. C which returns true if the matching is successful. As expected, this strategy is based on *metaMatch*:

$$\begin{aligned} \text{match } t' \text{ s.t. } C @ t &= \text{if } \bar{s} := \text{metaMatch}(\bar{\mathcal{R}}, \bar{t}', \bar{t}) \wedge \bar{s} \neq \text{noMatch} \\ &\quad \wedge \text{metaSubstitute}(\bar{C}, \bar{s}) = \text{true} \text{ then true else false.} \end{aligned}$$

where for simplicity we use *true* to denote the constant boolean *true* either at the object- or the meta-level, i.e., $\overline{\text{true}} = \text{true}$.

The language S allows further strategy definitions by combining them under the usual regular expression constructions: concatenation (“;”), union (“|”), iteration (“*”, “+”). Thus, given the strategies E, E' , the strategy $(E; E')@t$ is defined as $E'@(E@t)$, that is, E' is applied to the result of applying E to t . The strategy $(E | E')@t$ defined as $(E@t) \cup (E'@t)$ means that both E and E' are applied to t . The strategy $E^+@t$ is defined as $\bigcup_{i \geq 1} (E^i@t)$ with $E^1 = E$ and $E^n = E^{n-1}; E$, $E^* = \text{idle} | E^+$, thus it recursively applies itself.

The *if-then-else* combinators are denoted by $E ? E' : E''$ and their definition is (if $(E@t) \neq \emptyset$ then $E'@(E@t)$ else $E''@t$ fi) with the meaning that if, when evaluated in a given state term, the strategy E is successful then the strategy E' is evaluated in the resulting states, otherwise E'' is evaluated in the *initial* state. This strategy is further used to define:

$$\begin{aligned} \text{not}(E) &= E ? \text{fail} : \text{idle} & \text{try}(E) &= E ? \text{idle} : \text{idle} \\ \text{test}(E) &= \text{not}(E) ? \text{fail} : \text{idle} & E ! &= E^* ; \text{not}(E) \end{aligned}$$

which have the following meaning. The strategy *not* reverses the result of applying E . The strategy *try* changes the state term if the evaluation of E is successful, and if not, returns the initial state. The strategy *test* checks the success/failure result of E but it does not change the initial state. The strategy $E !$ “repeats until the end”.

We conclude with the description of a basic strategy combinator for rewriting subterms. The strategy *matchrew* t' s.t. C by t_1 using E_1, \dots, t_n using E_n , when applied to a term t , it first matches the pattern t' against the subject t and if successful and furthermore condition C holds, then it changes according to the subterms t_i rewritten under the corresponding strategies E_i .

$$\begin{aligned} \text{matchrew } t' \text{ s.t. } C \text{ by } tE @ t &= \text{if } \bar{s} := \text{metaMatch}(\bar{\mathcal{R}}, \bar{t}', \bar{t}) \wedge \bar{s} \neq \text{noMatch} \wedge \\ &\quad \text{metaSubstitute}(\bar{C}, \bar{s}) = \text{true} \wedge \\ &\quad \bar{\text{sol}} := \Pi \text{gen-sol}(\text{metaSubstitute}(\bar{t}, \bar{s}), tE) \\ &\quad \text{then } \{ \text{down}(\text{metaSubstitute}(\bar{t}, \bar{\text{sol}}_i)) \mid 1 \leq i \leq |\bar{\text{sol}}| \} \\ &\quad \text{else } \emptyset. \end{aligned}$$

where tE is a strategy list like t_1 using E_1, \dots, t_n using E_n and Π denotes the cartesian product of the sets returned by *gen-sol*. These sets contain substitutions for each t_i in the case the application of E_i is successful. These substitutions are used to compute the solutions of the whole strategy *matchRew*.

The function *gen-sol* is a so-called *continuation*. The way it works is as follows. First it checks for each t_i if it is a subterm of t with respect to substitution s . This is done at the meta-level, as the first parameter of *gen-sol* is a meta-term. If the matching is successful, the strategy E_i is applied to the substituted subterm.

$$\begin{aligned}
gen\text{-}sol(\bar{t}, (t_i \text{ using } E_i)) &= \text{if } \bar{s} := metaXmatch(\overline{\mathcal{R}}, \bar{t}_i, \bar{t}) \wedge \bar{s} \neq noMatch \wedge \\
&\quad metaSubstitute(\overline{\mathcal{C}}, \bar{s}) = true \wedge \\
&\quad sol_i := E_i @ down(metaSubstitute(\bar{t}_i, \bar{s})) \\
&\quad \text{then } map(\bar{t}_i, \overline{sol_i}) \text{ else } \emptyset.
\end{aligned}$$

where *map* builds a set of substitutions by associating to each \bar{t}_i each of the solutions sol_i resulting from applying E_i to the term (at the object-level, see the use of *down*) corresponding to the substituted meta-term \bar{t}_i with respect to the meta-substitution \bar{s} . In this way, *gen-sol* returns precisely the set of sets of substitutions $\{[t_i/sol_{i_1}], \dots, [t_i/sol_{i_k}]\}$ in the general case, that is, for lists with more than one element, by recursively calling itself:

$$gen\text{-}sol(\bar{t}, tE_1tE) = gen\text{-}sol(\bar{t}, tE_1) \cup gen\text{-}sol(\bar{t}, tE)$$

2.6.4 Controlling Executions with Strategies

In Section 2.6.1 we showed what it means to execute agents by rewriting. In this section we show how we can *strategically* rewrite agents by means of rewrite strategies. This has the advantage that we can control the executions of the agents without changing their code.

We have mentioned that one reason to control the executions is fair behaviours and that one way to achieve this is by means of fairness constraints expressed by LTL properties. We now focus on a different situation when control is needed, and that is, for example, when we want to enforce a certain execution. Thus we are interested not in the properties of the agent's behaviour in its totality but in particular executions. This sort of control we can enforce by means of rewrite strategies. For example, applying the strategy *o-act* to the initial configuration $\langle B_0, p_0 \rangle$ of the BUpL specification from Figure 2.10 has as result \emptyset because initially the only possible observable action *move*(2,0,1) fails. However, applying the strategy *fail-act* has as result the set:

$$\{\langle B_0, (move(3, 1, 0); p_0) \rangle, \langle B_0, (move(1, 0, 0); p_0) \rangle, \langle B_0, (move(2, 0, 0); p_0) \rangle\},$$

thus the set of all possible states reflecting a solution to the matching problem $on(x, y) \leq^? B_0$. Some of these resulting states are meaningless as it is the case with $\langle B_0, (move(1, 0, 0); p_0) \rangle$ because there is no point in moving a block from the floor to the floor.

A much more adequate strategy is *fail-act* $[\theta \leftarrow [x/3][y/1]]$, that is, to explicitly give the value we are interested in to the variable θ which appears in the rewrite rule *fail-act*. This application has a more refined and precise result, in our particular case, a set which contains only the state $\langle B_0, (move(3, 1, 0); p_0) \rangle$.

Besides forcing an agent to execute a precise action, we can also force an agent reach a precise mental state without explicitly saying which action can lead to such a state. For example, applying the strategy:

$$match \langle B, p \rangle \text{ s.t. } on(2, 1) \in B$$

to $\langle B_0, p_0 \rangle$ has as result \emptyset because *on*(2,1) is not in B_0 . To synthesise the above in one line and to also illustrate the usefulness of the strategy combinators we consider the following strategy definition:

$$o\text{-act} ? (\text{match } \langle B, p \rangle \text{ s.t. } \text{on}(2, 1)) : (\text{fail-act} ; o\text{-act}[o\text{-a} \leftarrow \text{move}(3, 1, 0)])$$

which has a successful application in two cases: (1) if the agent can perform an observable action leading to a new configuration where block 2 is on top of 1; (2) if the agent can repair itself such that it becomes possible to move block 3 on the floor.

Strategies are useful also at a language (and not agent) level. In this respect, the interest is in experimenting with the nondeterminism in the semantics. For example, BUpL semantics says nothing about the order of application of *sum*, *fail* or *o-act*. Since more than one may be applicable at a given point, it is of interest to analyse different scheduling policies. These can be seen as a first step in defining the so-called *agent deliberation cycles*. We do not discuss this issue in more detail here. However, the interested reader can refer to Section 4.4.4 where we treat a similar topic.

2.7 A Theory of Agent Refinement

In this section we introduce a theory of agent refinement. We do this by taking as working material the languages from the previous sections where we introduced a top-down design methodology, from abstract to concrete agent languages. In this setup, a natural question is with respect to the relation between the abstract and the concrete level, where the abstractness concerns either control or data. In what follows, we address the problem of when a concrete BUpL agent program is a *refinement* of an abstract BUnity agent. More precisely, we focus on two issues: (1) a formal definition of control refinement as trace inclusion and (2) a proof technique for refinement by means of the usual notion of *simulation*. With these being fixed, we show that two agent programs are in a refinement relation if there is no *deadlock* in the automaton corresponding to their “product” with respect to simulation. The product should be understood as a game the agents play, in the sense that the game advances one step when both agents can do the same action. A deadlock state is a state from which the game cannot advance, and is reached when the concrete agent can do an action which the abstract one cannot mimic. We express deadlock as an LTL property which we can model-check and this will constitute, in fact, one approach to verification in Chapter 3.

2.7.1 Control Refinement

Refinement is usually defined as trace inclusion, all the traces of the implementation are contained among the traces of the specification.

2.7.1. DEFINITION. [Control Refinement] Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent $((\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0)$ iff every trace of the BUpL agent is also a trace of the BUnity agent, that is $Tr((\mathcal{B}_0, p_0)) \subseteq Tr(\mathcal{B}_0)$. \blacklozenge

Being that we are interested only in fair agent executions, we consider also refinement in terms of fair trace inclusion where we take into account the definitions of *just₁*, *just₂* and *compassionate* as introduced in the previous sections.

2.7.2. DEFINITION. [Fair Refinement] Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . We say that the fair executions of the BUpL agent refine the fair executions of the BUnity agent $((\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0)$ iff every fair trace of the BUpL agent is also a fair trace of the BUnity agent, that is $(\forall tr \in Tr((\mathcal{B}_0, p_0))) (\sigma_{tr} \models \text{compassionate} \wedge \text{just}_2) \Rightarrow (tr \in Tr(\mathcal{B}_0) \wedge \sigma'_{tr} \models \text{just}_1)$, where σ_{tr} (resp. σ'_{tr}) is any corresponding computation path in the transition system associated to the BUpL (resp. BUnity) agent. \blacklozenge

We note that in the above definitions we have used the same symbols for both initial belief bases (\mathcal{B}_0) and sets of action definitions (\mathcal{A}) . This is not a restriction, it only simplifies the notation.

Proving refinement by definition is not practically feasible because the set of traces may be considerably large. We would need to check that for any solution to matching problems the corresponding trace belongs to both implementation and specification. Instead, refinement is usually proved by means of simulation, which has the advantage of locality of search: one looks for checks at the immediate (successor) transitions that can take place. We recall that the possible transitions for BUpL and BUnity agents are either τ steps (corresponding to choices between plans and repair rules) or steps labelled with action terms. Since we are interested in simulating only visible actions, we refer to weak simulation, which is oblivious with respect to τ steps.

2.7.3. DEFINITION. [Weak Simulation] Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . Let Σ, Σ' be the sets of mental states for each agent ($\mathcal{B}_0 \in \Sigma, (\mathcal{B}_0, p_0) \in \Sigma'$) and let R be a relation, $R \subseteq \Sigma \times \Sigma'$. R is called a weak simulation if, whenever $\mathcal{B}_0 R (\mathcal{B}_0, p_0)$, if $(\mathcal{B}_0, p_0) \xrightarrow{a} (\mathcal{B}, p)$, then it is also the case that $\mathcal{B}_0 \xrightarrow{a} \mathcal{B}$ and $\mathcal{B} R (\mathcal{B}, p)$. \blacklozenge

2.7.4. DEFINITION. Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . We say that the BUnity agent weakly simulates the BUpL agent $((\mathcal{B}_0, p_0) \lesssim \mathcal{B}_0)$ if there exists a weak simulation R such that $\mathcal{B}_0 R (\mathcal{B}_0, p_0)$. \blacklozenge

We recall that in general simulation is a sound but not necessarily complete proof technique for refinement. We take the classical situation from Figure 2.11 as a counter-example. However, simulation is complete when the simulating system is deterministic (see, for example, [vG90]). We make the remark that in the case of finite transition systems it is always possible to transform a nondeterministic system into a deterministic one by means of a power set construction ([ATW06] for the case of finite traces, and [Saf89, MS95] for the case of infinite traces). However, “determinization” is computationally hard ($2^{O(n \log n)}$ in the number of states [Saf89]) and thus usually infeasible when the focus is on verification.

In our case, the simulating agent is a BUnity one. BUnity agents, though highly nondeterministic with respect to control issues, have the property that they are modelled by *deterministic* transition systems. This is because though a BUnity agent makes arbitrary decisions regarding which action to execute, the mental state reflecting the effect of the chosen action is uniquely determined, thus *actions themselves are deterministic*. It follows that, in our framework, simulation is not only a sound but also a complete proof technique for refinement.

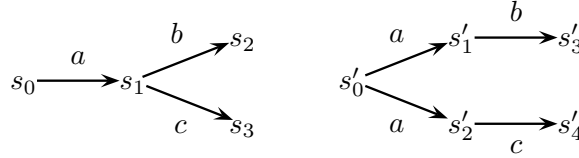


Figure 2.11: Refinement but not simulation

2.7.5. PROPOSITION. Let \mathcal{B}_0 be the initial mental state of a BUnity agent and let (\mathcal{B}_0, p_0) be the initial mental state of a BUpL agent. We have that $((\mathcal{B}_0, p_0) \lesssim \mathcal{B}_0)$ if and only if $((\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0)$.

Proof. The proof follows simply from the fact that BUnity agents are deterministic transition systems. ■

To decide simulation we take the following approach. We give a modal characterisation to simulation by constructing the synchronised product of a BUpL and BUnity agent and by defining an LTL property on the states of the product which we can effectively model-check as we shortly explain in Section 3.1.

2.7.6. DEFINITION. [BUpL-BUnity Synchronised Product] Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . Let also $I = (\Sigma, (\mathcal{B}_0, p_0), Act \cup \{\tau\}, \rightarrow)$ and $S = (\Sigma', \mathcal{B}_0, Act, \rightarrow)$ be the corresponding transition systems to the BUpL, resp. BUnity agent. Their left synchronised product is denoted by $I \otimes S$ and is defined as a transition system $(\Sigma \times \Sigma', \langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle, Act, \rightarrow)$. The semantics is given by the transition rule in Figure 2.12 where (\mathcal{B}_1, p) and \mathcal{B}_2 denote arbitrary BUpL, respectively BUnity states. ♦

$$\boxed{\frac{(\mathcal{B}_1, p) \xrightarrow{a} (\mathcal{B}'_1, p') \quad \mathcal{B}_2 \xrightarrow{a} \mathcal{B}'_2}{\langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle \xrightarrow{a} \langle (\mathcal{B}'_1, p'), \mathcal{B}'_2 \rangle}}$$

Figure 2.12: The Transition Rule for $I \otimes S$

Mathematically, the choice between either first the BUpL agent performs the step and then the BUnity performs the same step or the other way around is insignificant. However, from an implementation point of view, it is better to make the transition rule conditional. Only if the BUpL agent can fire an action the product changes state depending on whether the BUnity agent can mimic the BUpL agent. We say that the BUpL agent drives the simulation. We further say that if the BUnity agent can execute the same action, the product reaches a “good” state. Otherwise, the product is in a deadlocked state.

2.7.7. DEFINITION. [Deadlock] Let ∇ be the property $((\mathcal{B}_1, p) \xrightarrow{a} (\mathcal{B}'_1, p') \wedge \mathcal{B}_2 \not\xrightarrow{a})$. The state $\langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle$ has a *deadlock* when ∇ holds. That is:

$$\langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle \models \nabla \text{ iff } (\mathcal{B}_1, p) \xrightarrow{a} (\mathcal{B}'_1, p') \wedge \mathcal{B}_2 \not\xrightarrow{a}.$$

We say that the product is *deadlock-free* if it has no deadlocks. \blacklozenge

We note that we make the difference between a deadlocked and a terminal product state, where the only possible transition for the BUpL agent is the idling transition. We further make the remark that, since it basically depends on the BUnity agent being able to perform a certain action, the definition of deadlock introduces *asymmetry* in the executions of the BUpL and BUnity product.

2.7.8. PROPOSITION. Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . We have that the BUpL agent refines the BUnity agent $((\mathcal{B}_0, p_0) \subseteq \mathcal{B}_0)$ iff $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \models \Box \neg \nabla$, where $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle$ is the initial state of the BUpL-BUnity left synchronised product.

Proof. Since the proof is basically a simplification of the one we present for Theorem 2.7.9, we leave it to the reader. \blacksquare

In what follows we focus on the “fair” version of Proposition 2.7.8.

2.7.9. THEOREM. Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity agent with its initial mental state \mathcal{B}_0 and let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUpL agent with its initial mental state (\mathcal{B}_0, p_0) . We then have that the BUpL agent fairly refines the BUnity agent $((\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0)$ iff $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \models \text{compassionate} \wedge \text{just}_2 \rightarrow \text{just}_1 \wedge \Box \neg \nabla$, where $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle$ is the initial state of the BUpL-BUnity left synchronised product.

Proof. We recall that an LTL property holds in a state s if and only if it holds for any computation path σ beginning with s .

“ \Rightarrow ”:

Assume that $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \not\models \text{compassionate} \wedge \text{just}_2 \rightarrow \text{just}_1 \wedge \Box \neg \nabla$. This means that there exists a computation path (σ, σ') in the BUpL-BUnity product such that $\sigma \models \text{compassionate} \wedge \text{just}_2$ (*) and either (1) $\Diamond \nabla$ or (2) $\neg \text{just}_1$ holds. From (*) we have that $\text{tr}(\sigma)$ is a fair BUpL trace. From the hypothesis $(\mathcal{B}_0, p_0) \subseteq_f \mathcal{B}_0$ we have that there exists a BUnity computation path (which must be σ' since BUnity is deterministic) such that $\text{tr}(\sigma) = \text{tr}(\sigma')$ and $\sigma' \models \text{just}_1$ thus (2) cannot be true. Let us now consider (1). In order to have that $\Diamond \nabla$ holds for (σ, σ') there must be a deadlocked state $\langle (\mathcal{B}, p), \mathcal{B} \rangle$ on this path. But this implies that there is a fair BUpL trace $\text{tr}(\sigma)a$ which does not belong to the set of fair BUnity traces, thus contradicting the hypothesis.

“ \Leftarrow ”:

Assume that $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \models \text{compassionate} \wedge \text{just}_2$, meaning that any product path is fair with respect to the BUpL path. We make the remark that we do not need to worry about the “vacuity” problem ($\text{compassionate} \wedge \text{just}_2$ being always false) since there always exists a fair computation path (any scheduling algorithm will provide one). We need to prove that $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \models \text{just}_1$ (*) and $\langle (\mathcal{B}_0, p_0), \mathcal{B}_0 \rangle \models \Box \neg \nabla$ (**) implies fair refinement. We do this

by proving that $(**)$ implies simulation (thus also refinement). Since we have $(*)$, the product paths are also fair with respect to BUnity.

We construct a relation R containing all pairs of BUpL and BUnity reachable states and we prove R is a simulation relation. Let $R = \{(\langle \mathcal{B}, p \rangle, \mathcal{B}) \mid \langle \langle \mathcal{B}_0, p_0 \rangle, \mathcal{B}_0 \rangle \rightarrow^* \langle \langle \mathcal{B}, p \rangle, \mathcal{B} \rangle\}$. Let $(\langle \mathcal{B}, p \rangle, \mathcal{B}) \in R$ s.t. $\langle \mathcal{B}, p \rangle \xrightarrow{a} \langle \mathcal{B}', p' \rangle$. It is then the case that also $\mathcal{B} \xrightarrow{a} \mathcal{B}'$ otherwise $\langle \langle \mathcal{B}, p \rangle, \mathcal{B} \rangle \models \nabla$. We further need to prove that $\langle \langle \mathcal{B}', p \rangle, \mathcal{B}' \rangle$ is in R . This is, indeed, true since $\langle \langle \mathcal{B}_0, p_0 \rangle, \mathcal{B}_0 \rangle \rightarrow^* \langle \langle \mathcal{B}, p \rangle, \mathcal{B} \rangle \xrightarrow{a} \langle \langle \mathcal{B}', p \rangle, \mathcal{B}' \rangle$ thus $\langle \langle \mathcal{B}', p \rangle, \mathcal{B}' \rangle$ is a reachable state of the product. ■

2.7.10. REMARK. Refinement does not necessarily imply fair refinement. We consider a BUpL agent which can continuously perform only action a while the BUnity specification can additionally perform b . Refinement trivially holds ($\{a^\omega\} \subset \{(a^*b^*)^\omega\}$) however a^ω is unfair with respect to BUnity. ♣

We consider, for example, the BUpL and BUnity agents building the *ABC* tower. Any visible action that BUpL executes can be mimicked by the BUnity agent, thus in this case BUnity simulates BUpL and refinement is guaranteed. If we now pose the question whether fair executions of the BUpL agent refine fair executions of the BUnity agent, we have that, conforming to Theorem 2.7.9, the answer is positive if the formula $(compassionate \wedge just_2) \rightarrow (just_1 \wedge \Box \neg \nabla)$ is satisfied in the left product. This is because the traces of the product are of the form $(clean^* (move\theta)^*)^\omega$ and thus satisfying the fairness constraints for both BUpL and BUnity.

In this chapter we introduce several approaches to the verification of agent programs. We mainly focus on control refinement. In this respect, the simulation technique introduced in Section 2.7.1 can be seen as *one* way to verify agent correctness. Being that the technique can be reduced to model-checking, as we discuss in Section 3.1, this approach works fine, but for finite (small) state agents. As for infinite (large) state agents, it is a semi-decidable procedure: if the agents are *not* in a refinement relation then the procedure stops with a counter-example showing an example of a trace ending in a deadlock state. If, however, the agents *are* in a refinement relation then the procedure never stops. Infinite state agents can be understood as agents with *maintenance* goals, thus, their verification might be an issue of concern. To address it, we first consider in Section 3.2 whether there is a decision procedure for simulation. The answer to this issue is negative since we can reduce it to the halting problem for Minsky machines which is well-known to be undecidable ([Min61, SS63]). In fact, we can show more, namely that the languages are Turing powerful, thus also problems like reachability and (uniform) termination are undecidable since we can reduce them to undecidable properties for Turing machines.

Due to the above negative results, we need to tackle the verification of infinite state agents differently. Following classical approaches like in [CM88], we define a *weakest precondition calculus* for BUnity. This permits a more syntactical analysis. More precisely, we compute assertions about agent programs by simply looking at the specifications of actions, independent of the initial configurations. This is a crucial difference with respect to model-checking where at each step a next configuration is computed. Since we do not need to run the agent program, the method is naturally applicable to infinite state agents. The main use of a weakest precondition calculus for BUnity is to prove safety and progress properties. By definition, the refinement preserves properties, in the sense that any property of a BUpL agent is a property of the BUnity agent as well. This allows us to study the properties of BUpL by deriving properties of BUnity. The advantage here lies in the fact that there is no longer needed to explicitly verify the concrete BUpL agent, which might be, in principle, harder since BUpL structures are more sophisticated compared to BUnity.

Another orthogonal direction with respect to model-checking and the weakest precondition calculus is *testing* and this is the subject of Section 3.4. With respect to the calculus, testing is a semantical method. It focuses more on the “state-explosion” problem. We recall

that agent programs can be highly non deterministic and this can give rise to a large state space. Sometimes, exploring all of them like when model-checking, is unfeasible. This is why we propose a testing methodology where we define *test cases* as *deterministic* protocols specifying a plausible/possible agent execution. We implement them as *test drivers* by means of rewrite strategies, thus having a clear distinction between agent execution and control. This separation is important since it provides us with a “clean” way to “force” a different execution by simply defining a new test driver instead of changing the code of the agent itself.

3.1 Model-Checking Control Refinement

The decision procedure for simulation from Section 2.7.1 reduces to model-checking the BUpL-BUnity product for the absence of deadlock. In this section we explain how we can effectively model-check refinement via rewriting. To do this, we follow closely [Mes92, EMS02]. An important observation is that the models of rewrite specifications are, roughly, transition systems which can naturally be adapted to Kripke structures. We explain this in more detail by first focusing on the the affirmation that the models of rewrite specifications are roughly transition systems. We already mentioned in Section 2.6 that rewrite specifications axiomatise transitions systems. In what follows we give a more precise explanation. We recall that rewrite rules are meant to model concurrency and nondeterminism, thus, in general, algebras are not a suitable model choice for rewrite theories. This means that rewriting logic does not have initial *algebraic* semantics. Adequate models are instead transition systems which capture the dynamics and the operational semantics of rewrite rules. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ a transition system $L(\mathcal{R})$ is constructed by mapping states to equivalence classes $[t] \in T_{\Sigma/E}(X)$ and transition relation is given by the set of deduction rules together with the rewrite rules in R . $L(\mathcal{R})$ satisfies the sequent $t \rightarrow t'$ if and only if there is a computation $[t] \Rightarrow [t']$.

To describe the association of a Kripke structure to a rewrite specification we first recall that Kripke structures are labelled transition systems specialised such that their main use is in verification. Basically, the specialisation consists in associating to each state a set of properties in terms of atomic propositions. Given a rewrite specification \mathcal{R} with its model $L(\mathcal{R})$, the associated Kripke structure $K(\mathcal{R})$ extends the transition system $L(\mathcal{R})$ by incorporating (1) a predefined sort *State* to Σ (we denote the new signature Σ') and (2) a labelling function mapping each state $[t] \in T_{\Sigma'/E}$ to the set of state predicates that hold in $[t]$. The detailed construction can be found in [EMS02]. In this way, model-checking can be used for verifying LTL properties explicitly for rewrite specifications. In particular, for a formula $\phi \in \text{LTL}$, $K(\mathcal{R}), [t_0] \models \phi$ states that the rewrite specification \mathcal{R} and the initial state $[t_0]$ satisfy ϕ .

With the above, to model-check refinement in our framework, it suffices (1) to prototype the BUpL-BUnity product as a rewrite theory and (2) to express the deadlock as a state property. The latter comes easily from Definition 2.7.7. In what follows we discuss (1). Mapping the transition rule giving the semantics of the BUnity-BUpL program to a conditional rewrite rule is almost straightforward using Definition 2.7.6. The only tricky part concerns the prototyping of the conditions. This is because rewriting logic is, unlike structural operational semantics (SOS), context-sensitive due to the **Congruence** rule. To see this, we first recall

the transition rule:

$$\frac{(\mathcal{B}_1, p) \xRightarrow{a} (\mathcal{B}'_1, p') \quad \mathcal{B}_2 \xrightarrow{a} \mathcal{B}'_2}{\langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle \xrightarrow{a} \langle (\mathcal{B}'_1, p'), \mathcal{B}'_2 \rangle} \text{ (sync)}$$

which describes one unique configuration reached under certain conditions. There is no other possible transition, simply by definition. If we naively encode *(sync)* into the following rewrite rule:

$$\text{sync} : \langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle \xrightarrow{a} \langle (\mathcal{B}'_1, p'), \mathcal{B}'_2 \rangle \text{ if } (\mathcal{B}_1, p) \xRightarrow{a} (\mathcal{B}'_1, p') \wedge \mathcal{B}_2 \xrightarrow{a} \mathcal{B}'_2,$$

we may, however, end with unwanted executions corresponding to the situations where any of the agent configurations can rewrite on its own. We cannot prevent, by default, rewritings of, for instance, \mathcal{B}_2 , if there are applicable rewrites for this configuration. To see this, it suffices to take a closer look at the **Congruence** rule. This observation has the implication that we should be cautious at the way we handle contexts. In our particular framework, we must seek that the BUpL and the BUnity agent are executed with respect to the conditions of *(sync)*, and to forbid their random execution in the product configurations. We can achieve by means of *frozen* arguments. This concept is formally introduced in [BM03]. There, the authors present an extension of rewriting logic called *generalized rewriting logic* (GRL). GRL theories contain an additional function for denoting for each operator in the underlying equational logic signature which arguments are frozen, i.e., the positions under which rewriting is forbidden. This addition reflects in the GRL deduction rules **Congruence** and **Replacement** in the precise way that rewritings (resp., substitutions) never occur (resp., apply) below frozen argument positions in any given operator.

In this setup we take advantage of rewrite theories with flexible context-sensitive rewriting capabilities by expressing more control of the rewriting under contexts. In our case, we explicitly declare as frozen [BM03] the operator $\langle _, _ \rangle$ which we use to represent BUpL-Bunity configurations in the transition rule *(sync)*¹. In this way we forbid individual rewritings of the BUpL or the BUnity agent on their own.

What we further need to consider are the following two aspects. We first note that the above rewrite rule *sync* is actually incorrect: it contains information about the labels, while rewriting is oblivious with respect to labelling (*). We address this issue by the solution we present for the second aspect. This second aspect concerns modelling SOS *one-step* semantics (*(small)-step semantics*). Because of **Transitivity**, in general $R \vdash t \rightarrow t'$ does not represent an *atomic* step but may involve many complex computations. To explicitly allow *only* one step of execution of precisely the same action provided by the label in the transition *(sync)*, we use *auxiliary* frozen configurations for each agent:

$$\text{sync} : \langle (\mathcal{B}_1, p), \mathcal{B}_2 \rangle \rightarrow^1 \langle (\mathcal{B}'_1, p'), \mathcal{B}'_2 \rangle \text{ if } (\mathcal{B}_1, p) \rightarrow^* [a] (\mathcal{B}'_1, p') \wedge (\mathcal{B}_2 \rightarrow^1 [a] \mathcal{B}'_2),$$

where $\langle _, _ \rangle$ is declared as frozen and $[_]$ are the frozen auxiliary configurations for BUpL and BUnity incorporating also the labelling information and thus solving (*). The construction $[_]$ contains strings encoding action names modulo τ -steps. By this we mean that the

¹For simplicity, we use the same symbol in the rewrite rule corresponding to *(sync)*

special label τ play the role of identity in the concatenation operation, i.e., $[\tau^*a] = [a]$. This allows us to express by $x \rightarrow^* [a] x$ any computation like:

$$x \rightarrow^1 [\tau] x \dots x \rightarrow^1 [\tau] x \rightarrow^1 [a] x$$

with an arbitrary number of τ -steps before an a step which ensures that we correctly implement \xrightarrow{a} from the transition rule (*sync*).

The additional constructions $[_] _$ need to reflect in the rewrite rules corresponding to the transition rules for action execution. We only include the one for BUPL, as this has already been described in Section 2.6:

$$\begin{aligned} o\text{-act} : \langle B, a; p' \rangle &\rightarrow \langle \text{update}(B, E), \text{metaSubstitute}(p', \theta) \rangle \\ &\text{if } \theta := \text{match}(B, \text{pre}(a)) \wedge \theta \neq \text{noMatch} \\ &\wedge E := \text{metaSubstitute}(\text{post}(a), \theta) \wedge \text{consistent}(E) \\ &\wedge a : A^o \end{aligned}$$

The change in the rewrite rule *o-act* is illustrated as follows:

$$o\text{-act} : (\mathcal{B}, p) \rightarrow [(o\text{-}a)\theta] (\text{update}(\mathcal{B}, \xi\theta), \dots) \text{ if } \dots$$

where the “...” represent the missing information from the original rule. We note that there is no rewrite rule associated with the configuration $[_] _$. This ensures that in the rule *sync* we force the BUPL configuration to execute precisely one step (up to invisible τ steps). We remark that the resulting BUPL-BUnity configuration in the right-hand side of the rule *sync* contains the usual BUPL, BUnity configurations and *not* the auxiliary ones, i.e., it makes it possible to proceed with a new one step of execution for both BUPL and BUnity agents.

3.2 Undecidability results

In this section we address the question whether the proof technique for refinement by means of simulation works for *infinite* state agents. We answer this negatively, by first showing that BUnity is Turing powerful. We do this by adapting the translation of Turing machines to rewrite systems from [EGZ09, K1092, BKdV03] which is a simplification of the classical approach from [HL78].

We recall that a (deterministic) Turing machine is a quadruple (Q, Γ, q_0, δ) where Q is a finite set of states, q_0 is the initial state, Γ is a finite alphabet containing a designated constant \square and $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \{L, R\}$ is a partial transition function with L (resp. R) denoting a move to the left (resp. to the right). Informally, a transition $\delta(q, f) = (q', f', L)$ is read as an instruction “reading symbol f in state q replace f by f' , go left (or right) and update the current state as being q' ”. A configuration of a Turing machine is a pair (q, tape) where q is a state in Q and tape is the content of the tape, $\text{tape} : \mathbb{Z} \rightarrow \Gamma$ such that the carrier $\{n \in \mathbb{Z} \mid \text{tape}(n) \neq \square\}$ is finite. The set of all configurations is denoted by Conf_M . The relation \rightarrow_M on the set of configurations Conf_M is defined as $(q, \text{tape}) \rightarrow_M (q', \text{tape}')$ whenever:

1. $\delta(q, \text{tape}(0)) = (q', f, L)$, $\text{tape}'(1) = f$, and $\forall n \neq 0 \text{ tape}'(n+1) = \text{tape}(n)$, or

2. $\delta(q, \text{tape}(0)) = (q', f, R)$, $\text{tape}'(-1) = f$, and $\forall n \neq 0 \text{ tape}'(n-1) = \text{tape}(n)$.

The cases correspond to the Turing machine moving its head to the left (resp. right) and they're graphically illustrated in Figure 3.1, where t_i (resp. t'_i) stands for $\text{tape}(i)$ (resp. $\text{tape}'(i)$).

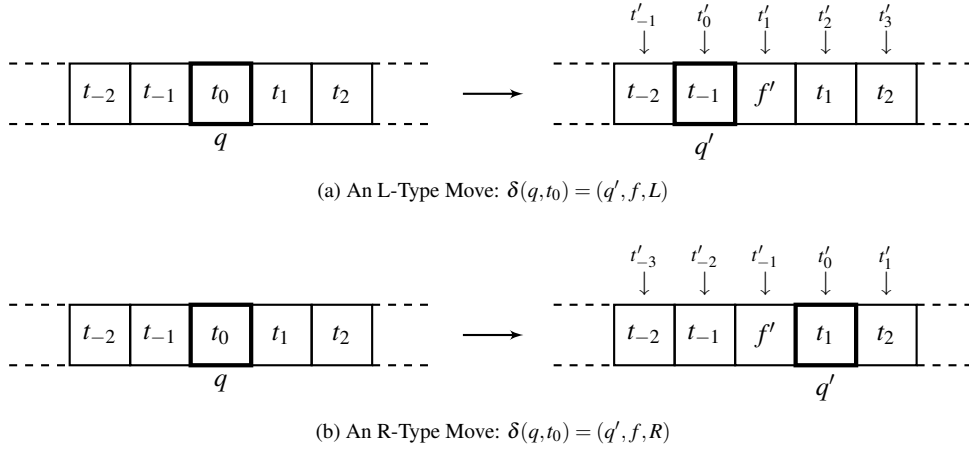


Figure 3.1: Transitions of (Deterministic) Turing Machines

If one assumes that the set of states and the alphabet are disjoint, one can see configurations (q, tape) as strings $w_1^{-1}qw_2$ with w_1 (resp. w_2) being a string over Γ^* representing the content of the tape at the left of the state q (resp. to the right) and $\text{tape}(0)$ being the first element to the right of q , i.e., $w_2(0)$. More precisely, given a string $w_1^{-1}qw_2$ we deduce that the state of the Turing machine is q and that the content of the tape is as follows:

$$\begin{cases} \text{tape}(-i) = w_1(i), & 1 \leq i \leq |w_1| \\ \text{tape}(i) = w_2(i-1), & 0 \leq i < |w_2| \\ \text{tape}(i) = \square, & i < -|w_1| \vee i \geq |w_2| \end{cases}$$

Given a Turing machine $M = (Q, \Gamma, q_0, \delta)$ we define a BUnity agent B_M as follows. The signature Σ_{B_M} is $\mathcal{F} = Q \cup \Gamma \cup \{\Delta\}$ where the symbols q from Q are interpreted as predicates of arity 2, the symbols f from Γ are interpreted as functions of arity 1, and Δ is a constant symbol representing an infinite number of blank symbols. We consider that the set of all possible beliefs, i.e., the Herbrand base of B_M , is $\mathfrak{B}_B = \{q(l, r) \mid q \in Q, l, r \in \mathfrak{U}_B\}$ with \mathfrak{U}_B being $T_{(\Gamma \cup \{\Delta\})}$, i.e., the ground terms constructed from the signature represented by $\Gamma \cup \{\Delta\}$. The belief bases of B_M consist of one belief and they change by means of the following basic actions:

$$\text{moveR}(x, y) = (\{q(f(x), y)\}, \{\neg q(f(x), y), q'(x, f'(y))\})$$

for every move to the right of the Turing machine ($\delta(q, f) = (q', f', R)$) and respectively,

$$\text{move}L(x, y) = (\{q(g(x), f(y))\}, \{\neg q(g(x), f(y)), q'(x, g(f'(y)))\}) \quad (3.1)$$

for every move to the left ($\delta(q, f) = (q', f', L)$) while reading symbol f . Both $\text{move}R$ and $\text{move}L$ correspond to the cases when the tape is non-empty and the symbol being read is non-blank. We now consider the following particular situations:

$$\text{move}L_{lb}(x, y) = (\{q(\Delta, f(y))\}, \{\neg q(\Delta, f(y)), q'(\Delta, \square(f'(y)))\})$$

for every move to the left ($\delta(q, f) = (q', f', L)$) on a tape with an empty left hand side and

$$\text{move}R_{rb}(x, y) = (\{q(x, \Delta)\}, \{\neg q(x, \Delta), q'(f'(x), \Delta)\})$$

for every move to the right on a tape with an empty right hand side while reading a blank symbol ($\delta(q, \square) = (q', f', R)$) and

$$\text{move}L_{rb}(x, y) = (\{q(g(x), \Delta)\}, \{\neg q(g(x), \Delta), q'(x, g(f'(\Delta)))\})$$

for every move to the left with empty right hand-side while reading a blank ($\delta(q, \square) = (q', f', L)$) and

$$\text{move}L_{lr}(x, y) = (\{q(\Delta, \Delta)\}, \{\neg q(\Delta, \Delta), q'(\Delta, \square(f'(\Delta)))\})$$

for every move to the left on an empty tape ($\delta(q, \square) = (q', f', L)$).

The terms from \mathcal{U}_B are meant to represent the content of the tape on the left (resp. on the right). To effectively translate from terms to tape content, we consider the mapping $\varphi : \mathcal{U}_B \rightarrow \Gamma^*$ defined recursively as $\varphi(\Delta) = \varepsilon$ and $\varphi(f(x)) = f\varphi(x)$. To make the connection between BUnity mental states (i.e., belief bases) and configurations of the Turing machine we consider the mapping $\phi : \mathcal{B}_B \rightarrow \text{Conf}_M$ defined as $\phi(q(l, r)) = \varphi^{-1}(l)q\varphi(r)$ with $l, r \in \mathcal{U}_B$.

We can now state the following proposition which says that for any Turing machine $M = (Q, \Gamma, q_0, \delta)$ we can construct a BUnity agent B'_M which simulates M . Basically, B'_M is B_M where we fix the initial belief base and we define the set of triggers as $\{\{\top\} \triangleright \text{do}(\text{move}*)\}$, with \top denoting the boolean *true*, and $\text{move}*$ being the placeholder for any of the move actions described above. Given the initial state q_0 of the Turing machine M , B'_M has an initial belief base $\mathcal{B}_0 = \{q_0(\Delta, \Delta)\}$.

To show that B'_M simulates M we have to consider three issues: (1) that for any configuration c of M there exists a mental state ms of B'_M such that $\phi(c) = ms$; (2) that for any mental state update $ms \rightarrow ms'$ there is a transition $\phi(ms) \rightarrow_M \phi(ms')$ of M ; and (3) that any configuration c of M that can be reached from $\phi(ms)$ there is a mental state ms' such that $ms \rightarrow ms'$ and $\phi(ms) = c$.

3.2.1. PROPOSITION. *Let M be a Turing machine. We have that B'_M simulates M :*

- (1) *for all $c \in \text{Conf}_M$ there exists $ms \in \mathcal{U}_B$ such that $\phi(ms) = c$*
- (2) *for all $ms \in \mathcal{U}_B$ such that $ms \rightarrow ms'$ implies that $\phi(ms) \rightarrow_M \phi(ms')$*

- (3) for all $ms \in \mathcal{U}_B$ such that $\phi(ms) \rightarrow c$ implies that there exists ms' with $ms \rightarrow ms'$ and $c = \phi(ms')$.

Proof.

- (1). follows immediately from the bijectivity of ϕ .
 (2). follows by case analysis. We only consider $ms \rightarrow ms'$ by means of a *moveL* basic action. All other cases are similar. We further assume that ms is of the form $\{q(l, r)\}$ with l, r different from \triangle , i.e., $l = g(c_1)$, $r = f(c_2)$ and c_1, c_2 are ground terms in \mathcal{U}_B . We then have that $\phi(ms) = \varphi^{-1}(c_1)gqf\varphi(c_2)$ (*). From $ms \rightarrow ms'$ we have that the precondition of *moveL* matches ms with a substitution θ . From Equation 3.1 we deduce that θ is $[x/c_1, y/c_2]$ and that there is a transition $\delta(q, f) = (q', f', L)$ (**) corresponding to a move to the left in the Turing machine. The effect of *moveL* on ms is to change it to $ms' = \{q'(x, g(f'(y)))\theta\}$, that is to $ms' = \{q'(c_1, g(f'(c_2)))\}$. Thus $\phi(ms')$ is $\varphi^{-1}(c_1)q'gf'\varphi(c_2)$ and together with (*) and (**) we have that $\varphi^{-1}(c_1)gqf\varphi(c_2) \rightarrow_M \varphi^{-1}(c_1)q'gf'\varphi(c_2)$.
 (3). is similar to (2). We assume $ms = \{q(g(c_1), f(c_2))\}$, thus we have that $\phi(ms) = \varphi^{-1}(c_1)gqf\varphi(c_2)$. We further assume that from this configuration the only possible move is to the left, that is, we consider that the transition is given by $\delta(q, f) = (q', f', L)$. It follows that $\phi(ms) \rightarrow_M c$ where $c = \varphi^{-1}(c_1)q'gf'\varphi(c_2)$ which is, in fact, $\phi(q'(c_1, g(f'(c_2))))$. From this we conclude that there is a mental state ms' defined as $\{q'(c_1, g(f'(c_2)))\}$ which can be reached from ms by doing a basic action *moveL* corresponding to $\delta(q, f) = (q', f', L)$. ■

From Proposition 3.2.1, the second item, we can show that problems like reachability and (uniform) termination are undecidable by a reduction to the (uniform) halting problem. By reachability and (uniform) termination we mean the following:

3.2.2. DEFINITION. [State Reachability and (Uniform) Termination] Given the BUnity mental states ms, ms' we say that ms' is *reachable* from ms if there exist a sequence ms_0, \dots, ms_n such that $ms = ms_0 \rightarrow ms_1 \dots \rightarrow ms_n = ms'$. We denote this property by $reach(ms, ms')$.

We say that a BUnity agent with an initial mental state ms is *terminating* if every execution from ms is finite. We denote this property by $T(ms)$. *Uniform termination* is the property that the agent terminates for any mental state, i.e., $UT = (\forall ms)T(ms)$. ♦

We note that reachability and termination relate to the notion of weak and strong normalisation from term rewriting systems. The difference between them is that for reachability we need to find one possible path leading to a given state while for termination we need to prove that all paths are finite. We further note that we explicitly distinguish between single and uniform termination. This is important since the problems lie in different complexity classes, the universal quantifier in UT classifies the property as Π_2^0 -complete², making it harder than single termination which is Σ_1^0 -complete.

3.2.3. COROLLARY. For any BUnity mental state ms , we have that $T(ms)$, and respectively $(\exists ms')(reach(ms, ms'))$, if and only if M halts on $\phi(ms)$, and respectively it halts with final configuration $\phi(ms')$.

²The interested reader can refer to [EGZ09] for an overview of the arithmetic hierarchy of the complexity of the undecidable properties.

Since any BUnity agent is trivially a BUpL agent (with one plan as a choice of all BUnity basic actions and one repair rule for each BUnity conditional action) the same BUnity undecidability results hold also for BUpL. The fact that BUnity and BUpL are Turing powerful has also the meaning that the set of BUnity and BUpL executions is recursively enumerable. From this we have the following proof for the undecidability of refinement.

3.2.4. PROPOSITION. *The refinement problem for BUpL and BUnity is undecidable.*

Proof. We have that refinement is simulation. Deciding simulation in the context of BUnity and BUpL reduces to finding an algorithm for the emptiness of recursively enumerable languages which is undecidable ([HMRU00]). ■

As a remark, we note that, in the context of the construction of A_m , if we think of Q denoting instructions and fixing $\Gamma = \{0, s\}$ for denoting zero and the successor function, we can imagine the construction of a BUnity agent which simulates a Minsky machine. We then obtain an encoding like the one used in classical works like [Bör87] for showing the undecidability of query answering in logic programming. We recall that a Minsky machine is a finite state program consisting of a finite set of instructions which manipulate two counters by incrementing (type 1.) and decrementing them (type 2.). The type 1. instructions have the form $q_i : \text{inc}_j ; \text{goto}(q_k)$ where inc_j increments the counter j . The type 2. instructions have the form $q_i : \text{if } (c_1 = 0) \text{ then } \text{goto}(q_k) \text{ else } \text{dec}_j ; \text{goto}(q_n)$ where dec_j decrements the counter j if its value is nonzero. The BUnity agent is constructed by associating to each instruction of the Minsky machine of type 1. one basic action:

$$a_i(x, y, k) = (\{q_i(x, y)\}, \{\neg q_i(x, y), q_k(s(x), y)\})$$

and to each type 2. instruction two basic actions handling the test on zero:

$$\begin{aligned} a_i(x, y, k) &= (\{q_i(s(x), y)\}, \{\neg q_i(s(x), y), q_k(x, y)\}) \\ a_i(0, y, n) &= (\{q_i(0, y)\}, \{\neg q_i(0, y), q_n(0, y)\}) \end{aligned}$$

The same is done for the instructions modifying c_2 .

Or, taking advantage of conditional actions, we have the following more structured BUnity program, however at the cost of adding new symbols and basic actions:

$$\begin{aligned} \mathcal{B}_0 &= \{inst(l_0), c_1(0), c_2(0)\} \\ \mathcal{A} &= \{ \text{inc}_j(x) = (\{\neg done(l) \wedge c_j(x)\}, \{\neg c_j(x), c_j(s(x)), done(l_i)\}), \\ &\quad \text{dec}_j(x) = (\{\neg done(l) \wedge c_j(s(x))\}, \{\neg c_j(s(x)), c_j(x), done(l_i)\}), \\ &\quad \text{goto}(l') = (\{done(l)\}, \{\neg done(l), \neg inst(l), inst(l')\}) \} \\ \mathcal{C} &= \{ \{inst(l_i)\} \triangleright do(\text{inc}(x)), \{inst(l_i)\} \triangleright do(\text{dec}(x)) \\ &\quad \{inst(l_i) \wedge c_j(0)\} \triangleright do(\text{goto}(l_n)), \{done(l_i)\} \triangleright do(\text{goto}(l_k)) \} \end{aligned}$$

3.3 A Weakest Precondition Calculus for BUnity

We recall that proving correctness of BUnity specifications and their refinement using model-checking techniques requires a reduction to *finite* models. Therefore, in this section, we introduce a deductive method based on assertions which allow to prove correctness of *infinite* state systems. Such proofs in general cannot be fully automated and require human interaction.

The earliest reference to axiomatic semantics is the work on flowcharts by Floyd [Flo67]. The current reference is the subsequent work of Hoare [Hoa69, Hoa72] which proposed a logical system for proving properties of programs. There, the main concept is of *pre-post formulae* $\{Pre\}S\{Post\}$ which expresses the *correctness* of the program fragment S with respect to Pre and $Post$. A Hoare theory provides axioms and inference rules to derive pre-post formulae. Another direction is the one taken by Dijkstra in [Dij76]. This approach aimed at developing a *calculus*, and not a logical theory, for reasoning about assertions by *transforming* them. Being based on predicate transformers, the approach is somehow closer to denotational than to axiomatic semantics, however the principles are similar to the ones in a Hoare theory. The calculus is usually referred to as a *weakest precondition calculus*. The main difference with respect to Hoare theories is that the calculus is meant to compute not just any arbitrary assertion, like one does in a Hoare theory, but the weakest one. This is why we choose this approach since we see weakest preconditions as being more "interesting" because they are the most general preconditions and thus we can derive all other preconditions from them (simply by the consequence rule).

In our framework, the weakest precondition calculus we envisage is meant to compute *assertions* about agents without running the agents themselves as it is the case when model-checking. This means that the computation is independent of any initial configuration. The main application of the weakest precondition calculus is in reasoning about safety properties like *invariants* or about progress properties like *leads-to* of (maybe infinite) state agents. For example, we imagine an agent which has a maintenance goal of constructing higher and higher towers of increasing lengths. Such an agent can be implemented in BUnity by adding to the set \mathcal{A}^t from Figure 2.4 a trigger like:

$$\{max(x)\} \triangleright do(move(x+1, z, x)),$$

where we assume that the *move* action is modified such that whenever a block x is added on another of smaller value it records x in the belief base in order to keep track of the maximum tower constructed so far. For such an agent model-checking a property like the one saying that towers are sorted, which it is not difficult to check by hand that it holds in all states, never terminates. It does terminate if there is a state which violates the property we are interested in. This means that model-checking can still be used in a first phase in order to falsify the property by finding counter-examples. However, assuming that model-checking does not terminate in a reasonable amount of time, we need to prove that the property holds by other means. One possible approach is to use axiomatic semantics in order to reason symbolically about BUnity agent programs. In an agent setup, an axiomatic semantics defines the meaning of agents by describing the effects of their actions on assertions about mental states.

3.3.1 Assertions

Assertions express relevant properties of belief bases. We consider them as first-order formulae extended with equalities. The free variables are to be interpreted as universal, i.e., an assertion ϕ represents the formula $\forall \bar{x}\phi$, with \bar{x} being $Var(\phi)$. This is a crucial difference between assertions and queries as described in Section 2.1.3. In contrast to the semantics of queries which was defined in terms of Herbrand satisfaction, the semantics of assertions is defined in terms of Herbrand validity. We say an assertion ϕ is valid, in symbols $\models \phi$, if and only if any Herbrand model \mathcal{B} in $\mathfrak{B}_{\mathcal{A}}$ satisfies ϕ , in symbols $\models_{\mathcal{B}} \phi$. As it was the case for queries, the ground terms substituting \bar{x} are from $\mathcal{U}_{\mathcal{A}}$. We use the same convention as for queries, that is, we denote “ $\models_{\mathcal{B}} \phi(\bar{t})$ for any $\bar{t} \in \mathcal{U}_{\mathcal{A}}$ ” as “ $\models_{\mathcal{B}} \phi \theta$ for any $\theta : Var(\phi) \rightarrow \mathcal{U}_{\mathcal{A}}$ ” and “ $\models_{\mathcal{B}} \forall \phi$ ” as “ $\models_{\mathcal{B}} \phi$ ” whenever it is clear from the context that ϕ is an assertion. The equalities in $\phi \theta$ are interpreted syntactically.

3.3.2 Action Correctness

To deduce properties of agent programs, the assertions are used in two main classes: *preconditions* and *postconditions*. Intuitively, an agent is correct with respect to an action a , a precondition Pre and a postcondition $Post$ if and only if when a is executed in a state in which Pre is satisfied then the updated state satisfies $Post$. Syntactically, they form the so called Hoare triples $\{Pre\}a\{Post\}$.

3.3.1. DEFINITION. [Action correctness] We say that an action $a = (\psi, \xi)$ if $Cond$ is correct with respect to precondition Pre and postcondition $Post$, written $\models \{Pre\}a\{Post\}$, if for belief base \mathcal{B} and for any θ in $Sols(\mathcal{B}, \psi)$ such that $\xi \theta$ is consistent we have that $Cond \theta$ is true and $\models_{\mathcal{B}} Pre \theta$ implies $\models_{\mathcal{B}'} Post \theta$, where \mathcal{B}' is $\mathcal{B} \uplus \xi \theta$. \blacklozenge

We draw attention that for $a = (\psi, \xi)$ if $Cond$ to be executed there must be an answer θ to the query ψ . We recall that θ guarantees to ground $Cond$. However, θ does not necessarily ground a precondition Pre or postcondition $Post$ as there is no constraint on the variables of Pre and $Post$ and the ones from ξ . Any such variable will be by default considered as universally quantified. Further, since our Herbrand interpretations do not contain equalities between ground terms, equality in assertions denotes *syntactical* identity.

3.3.2. EXAMPLE. Recalling the action *move* in the case where z is non-zero, let $\psi(\xi)$ be the pre (post)condition, and θ a substitution such that $\psi \theta$ is true. We show that the Hoare triple $\{clear(0)\}move(x, y, z)\{clear(0)\}$ is valid. We assume that $\models_{\mathcal{B}} clear(0) \theta$ (1). This means that $clear(0)$ is in \mathcal{B} . Because $\xi \theta$ does not delete $clear(0)$ we have that (1) implies $\models_{\mathcal{B}'} clear(0) \theta$ with $\mathcal{B}' = \mathcal{B} \uplus \xi \theta$. Thus $move(x, y, z)$ is correct with respect to the precondition $clear(0)$ and the same postcondition. The Hoare triple $\{\top\}move(x, y, z)\{clear(y)\}$ is also valid because if the move action is executed then by definition it updates the belief base with the information that y is clear ($clear(y)$ is in ξ). \spadesuit

3.3.3. EXAMPLE. Let $\{\exists w \text{ on}(x, w)\} \text{move}(x, y, z) \{\exists w \text{ clear}(w)\}$ be another example. By definition, the assertion $\exists w \text{ on}(x, w)\theta$ is satisfied in a belief base \mathcal{B} if there is some ground term t such that $\text{on}(x, t)\theta$ is in \mathcal{B} . This is indeed the case because whatever block represented by x it must be either on 0 or on another block. In fact, the same t substitutes the variable y in move (because x cannot be on two different blocks in the same time), thus y and w denote the same block. This means that, since y is clear after move , the assertion $\exists w \text{ clear}(w)$ holds. We note that if the variable w were a free variable, then the assertion $\text{on}(x, w)$ would have been implicitly universally quantified and thus $\text{on}(x, w)$ would have been no longer satisfied, as x cannot be on *any* block.

As a last example, let $\{\top\} \text{move}(x, y, z) \{\text{clear}(w)\}$ be another Hoare triple. This triple is invalid because w is implicitly universally quantified and there is always a ground term t such that $\text{clear}(w)[w/t]$ is false, for instance t is the block denoted by z , the block where x is placed after the move. ♠

3.3.3 The Predicate Transformer wp

A predicate transformer for actions is a function which takes as parameters an action a and a postcondition $Post$ and it computes a precondition Pre such that $\{Pre\}a\{Post\}$ is valid. In this section, we introduce wp as a predicate transformer which computes not just any precondition, but *the weakest* one.

Before defining wp , we consider some auxiliary constructions intended to simplify the next definitions and proofs. Given $\xi = \{l_1, \dots, l_n\}$ and a predicate symbol P we denote by I the set of indices such that $\{l_i \in \xi \mid i \in I\}$ represents all the positive literals with predicate P . Similarly, we denote by J the set of indices such that $\{l_j \in \xi \mid j \in J\}$ represents all the negative literals with predicate P . We further make use of a special construction which we denote by $Post\xi$ and which we define as follows.

3.3.4. DEFINITION. Given ξ the effect of an action and an assertion $Post$, we define $Post\xi$ inductively on the structure of $Post$:

- $P(\bar{t})\xi = (\bigwedge_{i \in I} \bar{t} \neq \bar{t}_i) \rightarrow (P(\bar{t}) \wedge \bigwedge_{j \in J} \bar{t} \neq \bar{t}_j)$
- $(\neg Post)\xi = \neg(Post\xi)$
- $(Post_1 \text{ op } Post_2)\xi = (Post_1\xi) \text{ op } (Post_2\xi)$ with $op \in \{\wedge, \vee, \rightarrow, =\}$
- $(Qx Post)\xi = Qx (Post\xi)$ with $Q \in \{\forall, \exists\}$ and $x \in Var(Post)$.



To give some intuition for the construction $Post\xi$ we take a closer look at its definition in the base case, which is the most interesting. Suppose we want to compute the weakest precondition such that $P(\bar{t})$ holds in the state resulting after the execution of an action $a = (\psi, \xi)$ if $Cond$, i.e., that $P(\bar{t})$ holds after the update $\xi\theta$ where θ is any ground substitution generated by ψ . Basically, we can distinguish between the following cases: either (1) $P(\bar{t})\theta$ holds before and it is not removed, i.e., $(\bar{t} \neq \bar{t}')\theta$ holds (or equally, \bar{t} and \bar{t}' are not unifiable)

for any $\neg P(\bar{t}')$ in ξ , or (2) $P(\bar{t}')\theta$ is added for some $P(\bar{t}')$ in ξ (and here it is important that $\xi\theta$ is consistent) in which case the equality $(t = t')\theta$ is valid, or equally, \bar{t} and \bar{t}' are syntactically the same under any θ . This reasoning justifies the need to allow equalities in assertions.

3.3.5. EXAMPLE. Let ξ be the effect of the $move(x, y, z)$ action when z is non-zero. We have the following calculations:

$$\begin{aligned}
 clear(0)\xi &= (0 \neq y) \rightarrow (clear(0) \wedge 0 \neq z) \\
 &\equiv 0 = y \vee clear(0) \wedge 0 \neq z & (*) \\
 clear(y)\xi &= (y \neq y) \rightarrow (clear(y) \wedge y \neq z) \\
 &\quad (using(y = y) \equiv \top) \\
 &\equiv \top \\
 \exists w \, clear(w)\xi &= \exists w ((w \neq y) \rightarrow (clear(w) \wedge w \neq z)) \\
 &\equiv \exists w ((w = y) \vee (clear(w) \wedge w \neq z)) \\
 clear(z)\xi &= (z \neq y) \rightarrow (clear(z) \wedge z \neq z) \\
 &\quad (using(z \neq z) \equiv \perp) \\
 &\equiv (z = y)
 \end{aligned}$$

With respect to the above calculations we may notice that the first three relate Example 3.3.2. To link assertion $(*)$ to the precondition from $\{clear(0)\}move(x, y, z)\{clear(0)\}$ we actually need the complete definition of wp . As explained in Example 3.3.13, the wp computation yields an assertion weaker than $clear(0)$. The assertion computed for $clear(y)\xi$ coincides with the precondition from $\{\top\}move(x, y, z)\{clear(y)\}$. The third computation is slightly more tricky to relate to the Hoare triple $\{\exists w \, on(x, w)\}move(x, y, z)\{\exists w \, clear(w)\}$. The relevant observation is that the satisfiability of $\exists w (w = y)$ is implied by the satisfiability of the precondition $\exists w \, on(x, w)$. ♠

3.3.6. LEMMA (SUBSTITUTION). *For any belief base \mathcal{B} , any assertion $Post$, any effect ξ , and any substitution θ which grounds ξ , under the assumption that $\xi\theta$ is consistent, we have that $\models_{\mathcal{B}} (Post\xi)\theta$ iff $\models_{\mathcal{B}'} Post\theta$ where \mathcal{B}' is $\mathcal{B} \uplus \xi\theta$.*

Proof. By induction on $Post$. We treat the **base case**: $P(\bar{t})$ and either I or J is non-empty: We begin with $\models_{\mathcal{B}} P(\bar{t})\xi\theta$ (1). By the definition of $P(\bar{t})\xi$ we have that (1) holds iff

$$\models_{\mathcal{B}} (\bigwedge_{i \in I} \bar{t}\theta \neq \bar{t}_i\theta) \rightarrow (P(\bar{t}\theta) \wedge \bigwedge_{j \in J} \bar{t}\theta \neq \bar{t}_j\theta) \quad (2)$$

For technical convenience only, we assume that $P(\bar{t})\xi\theta$ does not contain (implicitly universally quantified) free variables. If $\bigvee_{i \in I} \bar{t}\theta = \bar{t}_i\theta$ then there exists an index k in I such that $\bar{t}\theta = \bar{t}_k\theta$, that is, $l_k\theta = P(\bar{t}\theta)$, by the definition of ξ . Taking into account that $\xi\theta$ is consistent, by the definition of \uplus it follows that $\models_{\mathcal{B}'} P(\bar{t}\theta)$. Otherwise, it follows that $\models_{\mathcal{B}} P(\bar{t}\theta)$ and $l_j\theta \neq P(\bar{t}\theta)$, for all $j \in J$. By the definition of \uplus we then have that $\models_{\mathcal{B}'} P(\bar{t}\theta)$. ■

Thanks to the definition of $Post\xi$ we can now define wp in a concise way as follows:

3.3.7. DEFINITION. Let $a = (\psi, \xi)$ if $Cond$ be a bc-action and $Post$ be an assertion. The predicate transformer $wp : \mathcal{A} \times Pred \rightarrow Pred$ is defined as:

$$\begin{aligned} wp(a, Post) &= \psi \wedge Cond \rightarrow Post \xi \\ wp(\{\phi\} \triangleleft do(a), Post) &= \phi \rightarrow wp(a, Post). \end{aligned}$$

◆

Intuitively, $wp(a, Post)$ describes all agent states such that $Post$ holds in the next states resulting from executing a . Since wp defines a transformation of $Post$ into another assertion it is called a predicate transformer.

The Substitution Lemma facilitates the proof that wp is a *weakest* precondition, where the relation “weaker than” is defined as follows. We say that an assertion ϕ_1 is *weaker* than an assertion ϕ_2 if ϕ_2 logically implies ϕ_1 .

3.3.8. THEOREM. *The predicate wp is a weakest precondition, i.e., the following two statements hold:*

1. $\models \{wp(a, Post)\}a\{Post\}$
2. *if $\models \{Pre\}a\{Post\}$ then $\models Pre \rightarrow wp(a, Post)$.*

Proof. Let a be (ψ, ξ) if $Cond$, \mathcal{B} a belief base, θ a substitution such that $(\psi \wedge Cond)\theta$ holds in \mathcal{B}^* and $\xi\theta$ is consistent. Let also \mathcal{B}' be $\mathcal{B} \uplus \xi\theta$.

By Definition 3.3.1, proving 1. is equivalent to proving that $\models_{\mathcal{B}} wp(a, Post)\theta$ (**) implies $\models_{\mathcal{B}'} Post\theta$ for θ . By Definition 3.3.7, using (*), (**) is equivalent to $\models_{\mathcal{B}} (Post\xi)\theta$. From this, the implication follows easily by using “ \Rightarrow ” of Lemma 3.3.6 and this proves the first statement.

By Definition 3.3.1, we have that $\models_{\mathcal{B}} (Pre \wedge \psi)\theta$ (1) implies $\models_{\mathcal{B}'} Post\theta$ (2). From (“ \Leftarrow ” of Lemma 3.3.6) and using that $\models_{\mathcal{B}} \psi\theta$ follows from (1), (2) is equivalent to $\models_{\mathcal{B}} (\psi \rightarrow Post\xi)\theta$. Thus we have that $\models_{\mathcal{B}} Pre\theta$ (from (1)) implies $\models_{\mathcal{B}} wp(a, Post)\theta$. By ground satisfaction and by the definition of the semantics for assertions, we obtain that $\models_{\mathcal{B}} Pre \rightarrow wp(a, Post)$ and from this follows the second statement. ■

Further, we show that wp satisfies the usual properties like strictness and monotonicity which we use in Section 3.3.5.

3.3.9. PROPOSITION. *For any action a , and any assertions p and q , wp has the following properties:*

- (wp-1) $wp(a, \perp) = \perp$
- (wp-2) $\models p \rightarrow q$ implies $\models wp(a, p) \rightarrow wp(a, q)$

Proof.

- (wp-1) Assume that $wp(a, \perp) \neq \perp$. Then, using Lemma 3.3.6, we have that $\models_{\mathcal{B}} wp(a, \perp)\theta$ iff $\models_{\mathcal{B}'} \perp$, with $\mathcal{B}' = \mathcal{B} \uplus \xi\theta$, absurd.

(wp-2) It suffices to note that $p \rightarrow q$ implies that $p\xi \rightarrow q\xi$.

■

3.3.4 Invariants

An important use of the above weakest precondition calculus for actions is in proving invariants for BUnity programs. Invariants are usually defined as properties which hold in every reachable state. In LTL terminology, they correspond to “always” formulae $\Box\phi$.

3.3.10. DEFINITION. [Invariant] A property I is an invariant for a BUnity agent $\langle \mathcal{B}_0, \mathcal{A}^b, \mathcal{A}^t \rangle$ if and only if for any BUnity execution $\mathcal{B}_0, \dots, \mathcal{B}_i, \dots$ we have that $\models_{\mathcal{B}_i} I$ for any i . ♦

Definition 3.3.10 is a semantic definition which requires the exploration of the whole (possibly infinite) state space. Therefore we introduce *inductive* invariants which can be proved by the weakest precondition calculus.

3.3.11. DEFINITION. [Inductive Invariant] A property I is an inductive invariant for a BUnity agent $\langle \mathcal{B}_0, \mathcal{A}^b, \mathcal{A}^t \rangle$ iff $\models_{\mathcal{B}_0} I$ and for belief bases $\mathcal{B}, \mathcal{B}'$ and any trigger $a \in \mathcal{A}^t$ s.t. $\models_{\mathcal{B}} I$ and $\xrightarrow{a} \mathcal{B}'$ then we have that $\models_{\mathcal{B}'} I$. ♦

Intuitively, definition 3.3.11 says that an assertion I is an inductive invariant if it holds in the initial belief base and if, whenever it holds in a belief base \mathcal{B} and \mathcal{B} changes by action a to \mathcal{B}' , then I holds also in \mathcal{B}' . As an example, we consider $clear(0)$ which is an inductive invariant for the BUnity agent from Figure 2.4. First, the predicate $clear(0)$ holds in the initial belief base. Second, if $clear(0)$ holds in a belief base \mathcal{B} , then by Herbrand satisfaction $clear(0)$ is in \mathcal{B} ; since no ground effect of *move* deletes $clear(0)$ it follows that $clear(0)$ holds also in the updated belief base.

In general, invariants are not necessarily inductive. One classical example, referred to as “alternating” in the literature [BM08], is as follows. We consider the following action $a(x) = (P(x), \{-P(x), P(-x)\})$ and an initial belief base $\mathcal{B} = \{P(1)\}$. We have that, $I(x) = (P(x) \rightarrow x \geq -1)$ is an invariant (because there is one possible execution $\{P(1)\} \xrightarrow{a} \{P(-1)\} \xrightarrow{a} \{P(1)\} \dots$ and it is clear that in any belief base I holds) but it is not inductive because $I(x)$ is not guaranteed to hold in $\mathcal{B} \uplus \xi$ ($x \geq -1$ does not imply $-x \geq -1$). However, $I'(x) = P(x) \rightarrow (x \geq -1 \wedge x \leq 1)$ is inductive.

We note that the inductive definition relates to the LTL operator “next”: “if I holds in the current state, it will hold in the *next* state”. This affirmation can be expressed by means of correctness assertions, i.e., the Hoare triples $\{I\}a\{I\}$ must be valid for any a . In UNITY [CM88] terms, whenever $\models \{I\}a\{I\}$, I is called *stable*. With this in mind, thanks to Theorem 3.3.8 we show there is a direct connection between inductive invariants and weakest preconditions.

3.3.12. PROPOSITION. Any inductive invariant I of a BUnity agent $\langle \mathcal{B}_0, \mathcal{A}^b, \mathcal{A}^t \rangle$ satisfies the following properties:

(I-1) I holds in \mathcal{B}_0

(I-2) $\models I \rightarrow wp(a, I)$ for all triggers a in \mathcal{A}^t .

Proof. I is an inductive invariant iff I holds in the initial belief base and $\models \{I\}a\{I\}$. Since $wp(a, I)$ is the weakest we must have $\models I \rightarrow wp(a, I)$ for all triggers a . ■

3.3.13. EXAMPLE. To see an application of Proposition 3.3.12 we show how to compute $wp(\text{move}(x, y, z), \text{clear}(0))$. If we let ψ be the query of $\text{move}(x, y, z)$, from Definition 3.3.7 and from the calculations from Example 3.3.5 we have that $wp(\text{move}(x, y, z), \text{clear}(0))$ reduces to $\psi \wedge z \neq 0 \wedge y \neq z \rightarrow (y = 0 \vee \text{clear}(0) \wedge 0 \neq z)$ which is equivalent to $\psi \wedge y \neq z \rightarrow (y = 0 \vee \text{clear}(0))$. Thus, $\text{clear}(0) \rightarrow wp(\text{move}(x, y, z), \text{clear}(0))$ is valid and $wp(a, \text{clear}(0))$ is weaker than $\text{clear}(0)$. ♠

3.3.5 Leads-to Properties

In this section we show how we can use the wp predicate to compute the weakest precondition which *leads to* a postcondition. Leads-to properties are typical progress properties, usually denoted as $Pre \mapsto Post$. To see an example relating the action *move*, we consider the property that from any initial configuration of 3 blocks, after an arbitrary number of execution steps an agent reaches a configuration illustrating that block 3 is on 2 and 2 is on 1. Let $Post$ be this property, i.e., $Post = on(3, 2) \wedge on(2, 1)$. With no interesting triggers, besides a default one like $\{\top\} \triangleright do(\text{move}(x, y, z))$ which allows any action, under fairness assumptions, a leads-to property is $\top \mapsto Post$. The fairness assumptions forbid executions like moving to and from a single block by requiring that each enabled action will be at a point executed.

However, \top provides little information. Depending on the initial setup, the configuration reflecting $Post$ is attained after a certain number of steps. For each trace there is an assertion which leads to $Post$. The *weakest* one is the disjunction of them all.

Originally, *leads-to* was formulated as a relation defined as the disjunctive transitive closure of a relation *ensures*:

$$\frac{p \text{ ensures } q}{p \mapsto q} \quad \frac{p \mapsto q \quad q \mapsto r}{p \mapsto r} \quad \frac{\forall p(p \mapsto q)}{\exists p(p \mapsto q)}$$

where *ensures* satisfies the requirements stated as follows:

(E1) $p \text{ ensures } p$

(E2) $p \text{ ensures } \perp$ implies $\neg p$

(E3) $p \text{ ensures } q$ and $q \rightarrow r$ implies $p \text{ ensures } r$.

This relation *ensures* was formalised in UNITY by means of another relation *unless*. We recall their definitions, adapted to our framework:

$$Pre \text{ unless } Post \text{ iff } \models (Pre \wedge \neg Post) \rightarrow wp(a, Pre \vee Post),$$

for all enabled instantiated action a

$$Pre \text{ ensures } Post \text{ iff } \models Pre \text{ unless } Post \wedge (Pre \wedge \neg Post \rightarrow wp(a', Post)),$$

for some enabled instantiated action a'

The definition of *ensures* is such that the Requirements (E1)-(E3) are satisfied.

Intuitively, two assertions *Pre* and *Post* belong to the relation *unless* if and only if for any enabled instantiated action *a*, whenever *Pre* holds but not also *Post*, the weakest precondition of *Pre* \vee *Post* with respect to *a* holds. The latter ensures, by definition, that in each next state either *Pre* or *Post* holds. Similarly, *Pre* and *Post* belong to the relation *ensures* if and only if there exists an enabled *a'* such that in the next state resulting after *a* we have that *Post* holds. To draw another parallel to LTL notation, we observe the correspondence between *unless* and “until”, resp., between *ensures* and “eventually”.

We make the observation that the above definitions can be used to check if a given precondition *Pre* leads to a given postcondition *Post*. However, they do not say *how* to compute *Pre* and this is what we are interested in. We want to use *wp* in order to define a predicate transformer which, given a postcondition *Post*, computes the weakest precondition *Pre* such that *Pre* \mapsto *Post*. These issues have already been addressed in the literature and we refer in particular to [JKR89] as the work which we follow closely in the rest of the section. There the authors defined a predicate transformer *wlt* for computing the weakest precondition which leads to a postcondition. The construction of *wlt* uses a generic *wp* with an abstract definition in terms of the property which *wp* must fulfil. This property is monotonicity and this implies that we can make use of their results, thanks to Proposition 3.3.9 which says that our concrete definition of *wp* enjoys monotonicity. In what follows, we motivate and to explain the construction of *wlt* by reasoning at a “meta-level” about the *wp* predicate. To begin, we take a closer look at the definition of *Pre ensures Post*. By the definition of *unless*, *Pre ensures Post* is then equivalent to:

$$\bigwedge_a (Pre \wedge \neg Post \rightarrow wp(a, Pre \vee Post)) \wedge (Pre \wedge \neg Post \rightarrow wp(a', Post)). \quad (3.2)$$

Using $p \rightarrow q \wedge p \rightarrow r \equiv p \rightarrow q \wedge r$, (3.2) is equivalent to:

$$\bigwedge_a (Pre \wedge \neg Post \rightarrow wp(a, Pre \vee Post) \wedge wp(a', Post)). \quad (3.3)$$

Using $p \wedge \neg q \rightarrow r \equiv p \vee q \rightarrow r \vee q$, (3.3) is equivalent to:

$$Pre \vee Post \rightarrow \bigwedge_a (wp(a, Pre \vee Post) \wedge wp(a', Post) \vee Post). \quad (3.4)$$

Substituting *Pre* \vee *Post* by *Y* on the right-hand side of (3.4) we obtain the implication:

$$Pre \vee Post \rightarrow \bigwedge_a (wp(a, Y) \wedge wp(a', Post) \vee Post). \quad (3.5)$$

Since the right-hand side of the implication is monotonic, we can apply Knaster-Tarski Theorem to derive that the equation $Y = \bigwedge_a (wp(a, Y) \wedge wp(a', Post) \vee Post)$ in the unknown *Y* has a weakest solution which in [JKR89] is denoted by *stp*(*a'*, *Post*). This means that Implication (3.5) is equivalent to:

$$Pre \vee Post \rightarrow (stp(a', Post)). \quad (3.6)$$

The next step is to think of $(stp(a', Post))$ as a predicate transformer which entails $Post$ via a single *fixed* action. In [JKR89] this predicate transformer is denoted by $we(Post)$ (since a' is fixed we can abstract away from it). We note that the difference between we and wp is that we is oblivious with respect to “which” action entails $Post$. If we replace $Post$ by $wlt(Post)$ Implication (3.6) is equivalent to:

$$Pre \vee wlt(Post) \rightarrow we(wlt(Post)). \quad (3.7)$$

Using $p \rightarrow p \vee q$ on both sides in Implication (3.7) we obtain the following derivation:

$$Pre \rightarrow Pre \vee wlt(Post) \rightarrow we(wlt(Post)) \rightarrow we(wlt(Post)) \vee Post. \quad (3.8)$$

Taking into account that we are interested in the *weakest* precondition which leads to $Post$, (3.8) brings us to the fixpoint characterisation of wlt .

3.3.14. DEFINITION. [[JKR89]] The weakest predicate that leads to a postcondition $Post$ is recursively defined as:

$$wlt(Post) = Post \vee we(wlt(Post))$$

◆

The advantage of Definition 3.3.14 (over *ensures*) is that wlt is computable. Furthermore, it can be shown that wlt and \mapsto as defined in UNITY have the same expressive power. These connections are stated in Proposition 3.3.15.

3.3.15. PROPOSITION ([JKR89]). *The following statements hold:*

1. *Pre ensures Post implies that $Pre \rightarrow wlt(Post)$.*
2. *$Pre \rightarrow wlt(Post)$ iff $Pre \mapsto Post$.*

We stress that the above fixpoint characterisation of wlt would not have been possible if it weren't for Proposition 3.3.9.

3.3.16. EXAMPLE. Let $Post$ be $(on(2,1) \wedge on(3,2))$. We show how to compute $wlt(Post)$. We use the abbreviations: $P_1 = stp(move(2,0,1), Post)$, $P_2 = stp(move(3,0,2), P_1)$, ψ_1 as the query of $move(2,0,1)$ and ψ_2 as the query of $move(3,0,2)$. The next calculations are derived from Definition 3.3.4:

$$wp(move(2,0,1), Post) = \psi_1 \rightarrow on(3,2) \quad (3.9a)$$

$$wp(move(3,0,2), Post) = \psi_2 \rightarrow on(2,1) \quad (3.9b)$$

$$wp(move(3,0,2), \psi_1 \rightarrow on(3,2)) = \psi_2 \rightarrow \psi_1 \quad (3.9c)$$

The definition of stp gives a fixpoint characterisation such that either in the current state or in the next state (after executing a *fixed*) the postcondition holds. Since stp is the weakest solution it is a disjunction of all possible preconditions resulting when fixing the actions. For the sake of the example, we only detail a “local” view of this disjunction which is relevant to

us, i.e., the locality is with respect to the assumption that for any enabled action the weakest precondition for P_i is true which we consider as implicit in the following computations.

$$\begin{aligned}
P_1 &= wp(move(2, 0, 1), Post) \vee Post && \text{(using (3.9a))} \\
&\equiv (\psi_1 \rightarrow on(3, 2)) \vee Post \\
P_2 &= wp(move(3, 0, 2), P_1) \vee P_1 && \text{(using (3.9b), (3.9c))} \\
&\equiv (\psi_2 \rightarrow \psi_1) \vee (\psi_2 \rightarrow on(2, 1)) \vee Post \\
wlt(Post) &= Post \vee we(Post \vee we(Post \vee wlt(Post))) \\
&\equiv Post \vee stp(move(3, 0, 2), Post \vee \\
&\quad \underbrace{stp(move(2, 0, 1), Post \vee wlt(Post))}_{P_3}) \\
&= Post \vee \underbrace{stp(move(3, 0, 2), Post \vee P_3)}_{P_4} \\
P_3 &= P_1 \vee Post \vee wlt(Post) \\
P_4 &= P_2 \vee P_1 \vee Post \vee wlt(Post),
\end{aligned}$$


where we used $p \vee p \equiv p$. Thus, from the calculations for P_i :

$$\begin{aligned}
wlt(Post) &= Post \vee (\psi_1 \rightarrow on(3, 2)) \vee (\psi_2 \rightarrow on(2, 1)) \\
&\quad \vee (\psi_2 \rightarrow \psi_1) \vee wlt(Post)
\end{aligned}$$

which illustrates that the weakest precondition that leads to $Post$ from a state \mathcal{B} is either one of the following assertions:

- $Post$ itself, if $\models_{\mathcal{B}} Post$ (0 steps needed)
- $\psi_1 \rightarrow on(3, 2)$, if $\models_{\mathcal{B}} on(2, 1)$ (1 step needed)
- $\psi_2 \rightarrow on(2, 1)$, if $\models_{\mathcal{B}} on(3, 2)$ (1 step needed)
- $\psi_2 \rightarrow \psi_1$, if none of the above (2 steps needed)
- $wlt(Post)$, i.e., further unfoldings are needed ($Post$ cannot be reached in at most 2 steps).



3.3.17. REMARK. The definition of “ensures” uses the keyword “enabled”. This is to avoid typical situations where actions are not enabled. For instance, given $a = (P(c), \{Q(c)\})$ we have that $wp(a, Q(c))$ is true. If $P(c)$ is in the current belief base, i.e., a is enabled, then $\top \mapsto Q(c)$, however this is not the case if $P(c)$ were not in the belief base. A cleaner solution, i.e., which does not use semantic notions like the fact that actions are enabled, is to use the so-called “weak-until” relation [CM88] instead of “unless”. This allows the computation of Pre such that $Pre \mapsto Post$ regardless of initial configurations. 

Extending BUnity with knowledge bases requires little changes in the wp-calculus thanks to the fact that our formalisation allows in a straightforward manner to handle assertions expressed by means of knowledge bases. The main observation is that, together with a set of knowledge bases, a belief base can no longer be seen as a Herbrand model since clearly the logical theories representing the knowledge bases may contain variables. This means that we only need to define the Herbrand satisfaction relation by the standard model theoretic semantics.

3.3.18. EXAMPLE. This example illustrates the use of immutable knowledge bases in a wp calculation. Given the implication $P(c) \rightarrow Q(c)$ and the action $a = (\top, \{\neg Q(c)\})$, we want to prove using the wp calculus that $\models \{P(c)\}a\{Q(c)\}$ which shows that in the context of this implication removing $Q(c)$ in a belief base which contains $P(c)$ does not have any observable effect. First we observe that, by Definition 3.3.4, $Q(c)\{\neg Q(c)\}$ equals the assertion $\top \rightarrow (Q(c) \wedge c \neq c)$ which is logically equivalent inconsistent! However, computing $P(c)\{\neg Q(c)\}$ gives (again, by Definition 3.3.4) the assertion $\top \rightarrow (P(c) \wedge \top)$ which is clearly logically equivalent to $P(c)$. So we derive by the wp calculus that $\models \{P(c)\}a\{P(c)\}$. Since $P(c) \rightarrow Q(c)$, $P(c) \models Q(c)$, we obtain the desired result by the consequence rule.

This example shows that we can still use the basic wp-calculus (without knowledge bases) to reason about correctness of BUnity agents in the presence of knowledge bases. ♠

To conclude, we discuss the relation between our approach (as inspired by the classical works on weakest preconditions) and the closest reference in the field of artificial intelligence. This is the *situation calculus* [Rei01], used in solving the *frame problem* which consists in characterising the changes due to action execution. As explained in [KMS09], Reiter's solution is the so-called *successor axiom* which describes the precondition that ensures successful performance of actions. Informally, the successor axiom says that under some conditions pertaining to an action (to be read “the existence of answers for queries” in our framework), the *fluent* f (to be read “the postcondition”) becomes true if and only if either the condition making f true holds (in our case ξ^+ unifies with $Post$) or f is already true and the condition making f false does not hold (in our case $Post$ and ξ^- does not unify with $Post$). Though the idea is similar, there are a couple of notable differences. First, the successor axiom gives a generic definition which means that in order to use it one has to think of *what* exactly are the conditions which make the fluent f true or false. In contrast, what we have is a precise characterisation of such conditions thanks to the fact that we work with a fixed language, BUnity, and this allows us to be specific, for instance we can say that the condition of the effect which makes $Post$ boils down to syntactic equalities between terms. Second, all possible actions need to be considered in the successor axiom due to the universal quantification while the wp approach allows modularity by computing weakest assertions for each action separately. Third, the purposes are, in our view, different: for instance, there is no explicit application of the successor axiom in verification.

As a final remark, as pointed out in the recent survey from [KMS09], the techniques inspired either by the results in computer science (the wp calculus approach) or artificial intelligence (the situation calculus approach) can be seen as belonging to a larger class which concerns the problem of *reasoning about actions*. In fact, [KMS09] goes beyond these techniques and details the history of the *logic of action* not only in computer science and artificial

intelligence, but also in philosophy and in linguistics. As the authors note, the intention is to overview techniques developed in different communities while emphasising their similarities in the hope of future cross-fertilisation between communities.

3.4 Testing BUpL Using Strategies

In this section we attack the verification problem from a different perspective. More precisely, we investigate the problem whether a BUpL agent is conformant with respect to a given specification which is not necessarily a BUnity agent. We understand conformance as control refinement, that is, it holds when the set of traces of a BUpL agent is included in the set of traces of the specification. In a straightforward approach, one solution is to look at each execution trace of the agent and to check whether it is also a trace of the specification. However, this is often practically unfeasible due to large (possibly infinite) sets of agent executions. A more clever way is to consider the trace inclusion problem in the opposite direction, that is, to look first at the traces of the specification and to check whether these are also traces of the agent. Usually, “check” is achieved by model-checking or inductive verification. However, both approaches have their disadvantages: with model-checking one might run into the state explosion problem, while inductive verification is not automatic. An orthogonal technique is to use *testing*, and thus we describe a methodology for *testing* agents. Besides the expected use of finding “bugs” in an agent program, with respect to the previously introduced techniques, testing can be of further use either as an alternative to model-checking refinement in the state explosion problem or as a specialised check for invariants.

As a motivating example, throughout this section we use an extension of the BUpL builder described in Figure 2.6. We consider the agent from Figure 3.2³. It is meant to implement the specification “the agent should always construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number”⁴ (for example, 21, 4321 are “well-formed” towers).

The agent is designed such that it always builds a higher tower. The example can be understood as a typical agent with *maintenance* goals. Since the number of its mental states continuously increases, instead of model-checking, we test it. For illustration purposes, the implementation of the agent is on purpose faulty: assuming a correct initialisation, the agent program does not check the parity of X before adding the fact $done(X)$ to signal that it constructed a tower X .

3.4.1 Methodology

Our testing methodology consists of the following steps. We see the traces of the specification as the basic constructions for *test cases*. Since specifications are meant to be “small”, generating test cases is a much simpler task than exhaustively exploring possible agent executions. Either represented by regular expressions or by finite transition systems, specifications

³The code presents only the constructions which are additional to the ones from Figure 2.6.

⁴Since it is just meant to be an illustration, the notion of specification is merely informal.

$$\begin{aligned}
\mathcal{A} &= \{ \text{incLength}(x) = (\text{length}(x), \{ \neg \text{length}(x), \text{length}(x+1) \}), \\
&\quad \text{addBlock}(x) = (\neg \text{on}(x,0), \{ \text{on}(x,0), \text{clear}(x) \}), \\
&\quad \text{setMax}(x,y) = (\text{max}(y), \{ \neg \text{max}(y), \text{max}(x) \}), \\
&\quad \text{finish}(x,y) = (\neg \text{done}(x) \wedge \text{done}(y), \{ \neg(\text{done}(y)), \text{done}(x) \}) \} \\
\\
\mathcal{P} &= \{ \text{build}(n,c) = \text{move}(c-n,0,c-n-1); \text{incLength}(c-n-1); \text{build}(n-1,c) \\
&\quad \text{generate}(x,y) = \text{addBlock}(x); \text{generate}(x-1,y), \\
&\quad p_0(x,y) = \text{setMax}(x,y); \text{generate}(x,y) \} \\
\\
\mathcal{R} &= \{ \text{length}(x) \wedge \text{max}(y) \wedge (x \leq y) \leftarrow \text{build}(y,y+x-1), \\
&\quad \text{length}(x) \wedge \text{max}(x) \wedge \text{done}(y) \wedge (x \geq y) \leftarrow \text{finish}(x,y); \perp, \\
&\quad \text{max}(x) \wedge \text{done}(x) \leftarrow \text{setMax}(x+2,x), \text{generate}(x+2,x) \}
\end{aligned}$$

Figure 3.2: A BUpL Builder with Infinite State Space

can be used to generate test cases by model-checking, for example. Traces are *deterministic*, and since we build test cases on top of traces, also test cases are deterministic, in contrast to specifications. This is an important feature which makes testing an efficient approach. We define test cases as pairs of tests on actions and tests on facts. The tests on actions are finite sequences of pairs (a, E) where a is the action to be executed and E is the set of actions which are allowed to be executed at a given state. Whenever the agent *cannot execute* the action specified by the test on actions, or whenever the agent *can execute a forbidden* action, the corresponding trace represents a non-conformant execution. Tests on facts are temporal formulae that are checked on the traces generated with respect to tests on actions. They can be further used to detect “bad” executions.

Given that we define a formal language for expressing *what* a test case is, we then describe *how* to implement test cases. Namely, we use rewriting strategies to define *test drivers*. We recall that in a rewrite-based framework, strategies are meant to control nondeterministic executions by instrumenting the rewrite rules at a meta-level. Usually, in concrete implementations the nondeterminism is reduced by means of scheduling policies. While testing a concrete implementation, e.g., a multi-threaded Java application, there is no obvious distinction between testing the program itself and testing the default scheduling mechanism of the threads. We emphasise that the language we consider, BUpL, is a modelling language, where the *nondeterminism* in choices among plans, exception handling mechanisms and internal actions is a main aspect we deal with. Strategies give a great degree of flexibility which becomes important when the interest is in verification. For example, in our case, in order to analyse or experiment with a new testing formalism one only needs to change the *strategy* instead of changing *the semantics of the agent language* or *the agent program* itself.

Though test cases are deterministic, test drivers need to search all intermediary states that can be reached by nondeterministically executing internal BUpL computations. Defining test drivers by means of strategies is an elegant solution to the implicit nondeterminism in BUpL. However, it does not directly solve the problem of possibly divergent executions of internal steps. To avoid some divergent computations, we need to impose restrictions on the applica-

tion of the strategies. This makes it less intuitive that test drivers are faithfully implementing test cases, and thus the last issue we focus upon is the correctness of our mapping between test cases and test drivers.

3.4.2 Formalising Test Cases

Our test case format is based on two main concepts: observable actions and facts as appearing in belief bases. Our test case format is a kind of black box testing, aimed at testing the observable behaviour of agents. For this reason, we have made a distinction between internal and observable actions. The idea is that the execution of observable actions is visible from outside the agent. Observable actions can be actions the agent executes in the environment in which it operates. In the sequel, we will sometimes omit the adjective “observable” if it is clear from the context.

We introduce a general test case format that allows to express that certain sequences of observable actions are executed, and that the belief bases of the corresponding trace satisfy certain properties. That is, we consider that a test case \mathcal{T} is a pair consisting of a test on actions \mathcal{T}_a and a test on facts \mathcal{T}_f . Tests on actions are finite sequences of pairs $(a_0, E_0); \dots; (a_n, E_n)$. Each pair (a_i, E_i) consists of a ground observable action a_i to be executed and a set of actions E_i which are allowed to be executed from the current state. The idea is that a test on actions controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. Furthermore, the sets E can be used to identify “bad” traces. If, at a certain state of execution, the agent can perform a *forbidden* action, i.e., which is not allowed by the test case, then the corresponding trace is seen as a counter-example. If no restriction is imposed on the enabled actions we simply use the notation a instead of the pair (a, E) . It is then the case that a counter-example can be generated when the agent cannot execute the action indicated by the test. Tests on actions can be derived from a given specification by means of model-checking, for example. We stress that though the specification may be nondeterministic, tests on actions should be deterministic. This is crucial for reducing the state space and makes this approach essentially different from search techniques since it is more efficient. Tests on facts are specified like LTL formulae. For ease of presentation, we work only with a subset of basic formulae:

$$\mathcal{T}_f ::= \text{true} \mid \text{fact} \mid \neg \text{fact} \mid \Box(\neg \bigcirc \text{true} \rightarrow \text{fact}) \mid \text{fact} \wedge \text{fact} \mid \Box \text{fact} \mid \Diamond \text{fact}$$

with *fact* being a ground atomic formula. Observe that the syntax allows also test cases consisting of tests on actions only, $(\mathcal{T}_a, \text{true})$ which we write shortly as \mathcal{T}_a . The LTL formula $\Box(\neg \bigcirc \text{true} \rightarrow \text{fact})$ can be used to check if *fact* holds in the last states, that is, in the states reachable after executing the test on actions. Tests on facts are meant to provide additional counter-examples besides those reflecting forbidden actions. While tests on actions can be automatically derived from the specification (where the tester needs only to choose adequate test cases), using tests on facts requires more effort and intuition from the tester. For illustration purposes, we provide an example of an adequate test on facts by the end of the paper.

To define formally when a BUpL agent satisfies a test we use induction on the structure of test cases. We denote the application of a test \mathcal{T} on an initial configuration (an initial BUpL

mental state) ms_0 as $\mathcal{T}@ms_0$. The (set) semantics is defined such that it yields the set of final states reachable through executing the agent restricted by the test, i.e., only those actions are executed that comply with the test. This means that an agent with initial mental state ms_0 satisfies a test \mathcal{T} if $\mathcal{T}@ms_0 \neq \emptyset$, in which case we say that a test \mathcal{T} is *successful*.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms \mid ms_0 \xRightarrow{a} ms\}, & \mathcal{T} = (a, E) \wedge E(ms_0) \subseteq E \\ \emptyset, & \mathcal{T} = (a, E) \wedge E(ms_0) \not\subseteq E \\ \mathcal{T}_a^2 @ (\mathcal{T}_a^1 @ ms_0), & \mathcal{T} = \mathcal{T}_a^1; \mathcal{T}_a^2 \\ \{ms \mid ms \in \mathcal{T}_a @ ms_0 \wedge \Pi_{ms_0}^{\mathcal{T}_a}(ms) \models \mathcal{T}_f\}, & \mathcal{T} = (\mathcal{T}_a, \mathcal{T}_f) \end{cases}$$

The arrow \xRightarrow{a} stands for $\Rightarrow \xrightarrow{a} \Rightarrow$, where \Rightarrow denotes the reflexive and transitive closure of $\xrightarrow{\tau}$, and $E(ms)$ denotes the set of enabled actions, i.e., the actions that can be readily executed from ms , $E(ms) = \{a \mid \exists ms' \text{ s.t. } ms \xrightarrow{a} ms'\}$. The idea behind the definition of the semantics of $(a, E)@ms_0$ is that the test should be successful for ms_0 if action a can be executed in ms_0 , while the enabled actions from the states reached by doing a should be a subset of E (defined by $E(ms) \subseteq E$). The result is then the set of mental states resulting from the execution of a , as defined by $\{ms \mid ms_0 \xRightarrow{a} ms\}$. We need to keep those mental states to allow a compositional definition of the semantics. In particular, when defining the semantics of $\mathcal{T}_a^1; \mathcal{T}_a^2$ we need the mental states resulting from applying the test \mathcal{T}_a^1 , since those are the mental states in which we then apply the test \mathcal{T}_a^2 , as defined by $\mathcal{T}_a^2 @ (\mathcal{T}_a^1 @ ms_0)$. In the definition of the semantics of $(\mathcal{T}_a, \mathcal{T}_f)$, by abuse of notation, we use $\Pi_{ms_0}^{\mathcal{T}_a}(ms)$ to denote the paths from ms_0 to ms which are taken while executing \mathcal{T}_a . These paths are with respect to observable actions, that is, we abstract from intermediary states reached by doing τ steps. More specifically, each state in a path is reached from the previous by executing an observable action and then executing a number of τ steps until an observable action is again about to be executed (or no transitions are possible). In the initial state, first τ steps can be executed before the first observable action is executed. Tests on facts are thus checked in states resulting from the execution of an observable action and as many τ steps as possible. We call these states *stable*. The definition says that the result of applying the test $(\mathcal{T}_a, \mathcal{T}_f)$ is a subset of $\mathcal{T}_a @ ms_0$, namely, those states ms which are reachable after executing \mathcal{T}_a and the corresponding path LTL satisfies \mathcal{T}_f .

Our language is such that tests on facts can be omitted. By design, they are meant to provide more expressiveness and to give more freedom to the tester. One might raise the issue that inspecting facts classifies our method as white-box testing. However, since facts can be deduced from the effects of actions, our method lies at the boundary between black-box and grey-box testing. In order to define test cases, there is no need to understand the way BUpL agents work (i.e., the internal mechanism for updating states or the structure of repair rules and plans), but only to look at basic actions, which we see as the interface of BUpL agents.

3.4.3 Using Rewrite Strategies to Define Test Drivers

In this section we describe how to define *test drivers* for test cases by means of the strategy language S described in Section 2.6.2. To give some intuition and motivation, we consider the way one would implement the basic test case a . By definition, the application of this test

case to a BUpL mental state ms is the set of all mental states which can be reached from ms by executing the observable action a after eventually executing τ steps corresponding to internal actions, applying repair rules or making choices, i.e., after computing closure sets of particular types of rewrite rules. It thus represents a *strategic* rewriting of ms . We are only interested in those rewritings which finally make it possible to execute a . To achieve this at the object-level means to have a procedure implementing the computation of the closure sets. However, the semantics of the application of the test a is independent of the computation of closure sets. Following [EMOMV07], we promote the design principle that automated deduction methods (e.g., closure sets of τ steps) should be specified *declaratively* as non-deterministic sets of inference rules and not *procedurally*. Depending on the application, specific algorithms for implementing the specifications should be given as *strategies* to apply the inference rules. This has the implication that there is a clear separation between *execution* (by rewriting) at the object-level and *control* (of rewriting) at the meta-level.

In what follows, for ease of reference, we denote by \mathbb{S} (resp. \mathbb{T}) the set of strategies (tests) and by s the mapping from tests to test drivers, i.e., $s : \mathbb{T} \rightarrow \mathbb{S}$. Since the definition of tests is inductive, so is the definition of s . We first consider the test drivers for tests on actions:

$$s(\mathcal{T}) = \begin{cases} allow(E) ; do(a), & \mathcal{T} = (a, E) \\ s(\mathcal{T}_1) ; s(\mathcal{T}_2), & \mathcal{T} = \mathcal{T}_1 ; \mathcal{T}_2 \end{cases}$$

thus sequences of tests map to sequences of strategies. We describe the basic test driver $do(a)$ in more detail. Observe that though tests on actions are deterministic, there are still possibly many executions due to internal actions, choices in plans and repair rules. Thus the test driver must search “all” possible intermediary states which can be reached by doing τ steps. By means of strategies, this is an easy process. By definition, the transitive closure of τ steps, \Rightarrow , is $\xrightarrow{\tau^*}$, with τ being one of the label *sum*, *i-act*, or *fail-act* and the corresponding being maximal, in the sense that no τ steps are possible from the last state. Thus, in a naive approach, we could simply consider the following test driver:

$$tauClosure = (sum \mid i-act \mid fail-act)!$$

which is clearly implementing \Rightarrow . However, though the order of application of the τ steps does not matter when the computation paths are finite, this is no longer the case when considering infinite paths. Consider an extraneous agent program with a plan $p = i-a + i-b$ where $i-a$ is always enabled and $i-b$, on the contrary, is never enabled and a repair rule ($true \leftarrow i-b$) which says that whenever there is a failure repair it by executing $i-b$. Applying $tauClosure$ as defined above we obtain two solutions corresponding to a finite path reflecting the choice for executing $i-a$ and a divergent path reflecting the choice for executing $i-b$ then failing all the time. As long as we are only interested in the “first” solution, then $tauClosure$ is fine, however, if we want to generate also the “next” solution then the computation will not terminate. From this we conclude that we may lose termination if any application order is allowed while we may be able to achieve it if we impose a certain order. Since one source of non-termination is mainly in a sort of “unfairness” with regard to enabled internal actions, a much more adequate test driver is implemented if we enforce the execution of internal actions after eventually applying the sequence ($sum; fail-act$). That is, $tauClosure$ becomes:

$$\text{tauClosure} = (\text{try}(\text{sum}); \text{try}(\text{fail-act}); i\text{-act})!; \text{try}(\text{sum}); \text{try}(\text{fail-act})$$

We make a few observations with respect to the new definition of *tauClosure*. First, since one might expect multiple *sum* and *fail* applications before an internal action is executed, it is no longer immediately clear that *tauClosure* faithfully implements \Rightarrow . We present a correctness proof by the end of the section. Second, because we use the sequential strategy, we need to surround both *sum* and *fail-act* by *try* blocks. Otherwise, if either one of them were not applicable, i.e., the current plan is not a *sum* and the “head” action is enabled, then the strategy $(\text{sum} ; \text{fail-act} ; i\text{-act})$ fails which is not what we want. By means of the parametrised strategy *try* the initial state is preserved in the case that *sum* or *fail-act* fails. Third, we order *fail-act* after *sum* because if we were to use the strategy $(\text{try}(\text{sum} \mid \text{fail-act}) ; i\text{-act})$ and the current plan is a *sum* of two failing plans, then the whole strategy fails though there might have been possible to replace the failing plans with a “good” plan by applying *fail-act*. Fourth, we require that repair rules are of a particular format, that is $\phi \leftarrow p$ with *p* not containing the *sum* operator. This is in order to avoid situations where the application of *fail-act* entails the application of *sum* which entails the application of *fail-act* and so forth (that is, non-terminating strategies $(\text{sum} ; \text{fail-act})!$). Such format does not result in the loss of expressiveness since having one repair rule $\phi \leftarrow p_1 + p_2$ is equivalent to having two repair rules $\phi \leftarrow p_i$, with $i \in \{1, 2\}$. Fifth, the use of strategies can be tricky. Though one might be tempted to use the strategy $\text{try}(\text{sum} ; \text{fail-act})$ instead of $\text{try}(\text{sum}) ; \text{try}(\text{fail-act})$, the first one is “wrong”, meaning that if *fail-act* is not applicable after *sum* then the original state is returned instead of the one reached by applying *sum*. The last observation is with respect to the normalisation strategy. Since “!” returns the state previous to the one that failed, we need to apply again $\text{try}(\text{sum}); \text{try}(\text{fail-act})$ to make sure that from the resulting state no τ steps can be taken.

By means of *tauClosure*, the definition of *do(a)* is straightforward:

$$\text{do}(a) = \text{tauClosure}; o\text{-act}[o\text{-}a \leftarrow a]; \text{tauClosure}$$

which corresponds to the definition of \xRightarrow{a} . We note that *tauClosure* is no longer applicable when *i-act* fails after *sum* and *fail-act* have been applied. This means that the only possible scenario is that the head of the current plan is an observable action. If this action is in fact *a*, then $o\text{-act}[o\text{-}a \leftarrow a]$ is successful, otherwise it fails.

The definition of the strategy *allow(E)* makes use of the *match* construction:

$$\text{allow}(E) = \text{match } ms \text{ s.t. } \text{ready}(ms) \subseteq E$$

which means that *allow(E)* succeeds if the current mental state satisfies the condition $\text{ready}(ms) \subseteq E$, where *ready* is a function defined on BUpL mental states. This function is implemented such that it returns the set of actions ready to be executed. For simplicity, we do not detail its implementation but briefly describe it. Recall that BUpL mental states are pairs of belief bases and plans. The function *ready* reasons on possible cases. If the current plan is a *sum* of plans then *ready* is called recursively. Otherwise, depending on the action *a* in the head of the plan, either *a* is enabled and so the function *ready* returns *a*, or *a* fails and the function *ready* recursively considers all the plans that can substitute the current one, that is, it recursively analyses the active repair rules.

So far, we have focused on tests on actions \mathcal{T}_a . We focus now on the general test cases $(\mathcal{T}_a, \mathcal{T}_f)$. We begin by first considering the test driver implementing the test case for checking whether $fact$ is in the last states reachable by executing \mathcal{T}_a , i.e., $s((\mathcal{T}_a, \Box(\neg \bigcirc true \rightarrow fact)))$. For this, we consider an auxiliary strategy $check(fact)$:

$$check(fact) = match(\mathcal{B}, p) \text{ s.t. } fact \in \mathcal{B}$$

which is successful if $fact$ is in the belief base from the current state. With this strategy we can define $s((\mathcal{T}_a, \Box(\neg \bigcirc true \rightarrow fact)))$ simply as $s(\mathcal{T}_a; check(fact))$. We can further use $check(fact)$ for defining test drivers working with $\neg fact$ as $not(check(fact))$ and with $fact_1 \wedge fact_2$ as $check(fact_1); check(fact_2)$. The cases with respect to the temporal formulae are defined by case analysis. We present only the implementation of the non-trivial ones:

$$\begin{aligned} s(((a, E); \mathcal{T}_a, \Diamond fact)) &= check(fact) ? s((a, E); \mathcal{T}_a) : s((a, E)) ; s((\mathcal{T}_a, \Diamond fact)) \\ s(((a, E); \mathcal{T}_a, \Box fact)) &= check(fact) ; s((a, E)) ; s((\mathcal{T}_a, \Box fact)) \end{aligned}$$

which illustrates that the main difference between them is that for $\Diamond fact$ we stop checking $fact$ as soon as we reached a state where $fact$ is in the belief base; from this state we continue with only executing the test on actions. However, for $\Box fact$ we check until the end.

Observe that the semantics of the testing language was defined such that we have a separation between implementing test drivers and *reporting* the results. This is important since running a test driver should be orthogonal to the interpretation and the analysis of the possible output. One plausible and intuitive interpretation is the following one. When the test driver is successful the tester has the confirmation that the test case corresponds to a “good” trace in the agent program. When the test driver fails, the tester can further define new strategies to obtain more information. Consider, as an example, a strategy returning the states previous to the failure. More sophisticated implementations like gathering information about traces instead of states are left to the imagination of the reader. These traces correspond to the shortest counter-examples. This follows from the semantics of the testing language. At each action execution a check is performed whether forbidden actions are possible. If this is the case, then the test fails.

Assuming that we fix an interpretation of the results as above, we proceed by showing that test drivers are partially correct and complete with respect to the definition of test cases.

3.4.1. DEFINITION. Given a test case \mathcal{T} and the corresponding test driver $s(\mathcal{T})$, we say that the application of $s(\mathcal{T})$ is correct, if, on the one hand, successful executions of the test driver are successful applications of the test case, and if, on the other hand, the test driver fails then test case also fails. Similarly, s is complete if (un)successful applications of the test case \mathcal{T} are (un)successful executions of the test driver $s(\mathcal{T})$. \blacklozenge

Before stating the main result, we show two helpful lemmas. Recall that, at each repetition step, the strategy *tauClosure* tries to apply *sum* and *fail-act* only once. Intuitively, this is sufficient for the following reason. Let us first consider *fail-act*: if, on the one hand, after the application of *fail-act* no action can take place then applying *fail-act* again can do no good, since nothing changed; if, on the other hand, after applying once *fail-act* the first action of the new plan can be executed then we are done, the faulty plan has been repaired. From this, we have the following lemma:

3.4.2. LEMMA. *The strategy $\text{try}(\text{fail-act})$ is idempotent, i.e., for any ms $\text{try}(\text{fail-act})^2 @ms = \text{try}(\text{fail-act}) @ms$.*

Proof. Let $Res = \text{try}(\text{fail-act}) @ms$. Any $ms' \in Res$ different from ms is the result of applying the rewrite rule fail-act so it has the form $(\mathcal{B}, p\theta)$, where $\phi \leftarrow p \in \mathcal{R}$ (the set of repair rules) and $\theta \in \text{Sols}(\mathcal{B}, \phi)$. If fail-act were again applicable for such ms' , the resulting term ms'' is also of the same form since \mathcal{R} is fixed and \mathcal{B} does not change. Thus, any ms'' is already an element of Res and so $\text{try}(\text{fail-act}) @Res = Res$. ■

An analogous reasoning works also for sum . Taking into account that the “+” operator is commutative and associative and that the “;” operator is associative, a *normal form* (i.e., sum of plans with only sequence operators) always exists. Since sum is applied to states where the plans are reduced to their normal form we have that states with *basic* plans will always be in the result of trying to apply sum more than once.

3.4.3. LEMMA. *Given a mental state ms we have that $\text{sum}! @ms \subseteq \text{try}(\text{sum}) @ms$.*

Proof. We only consider the interesting case where sum is applicable, that is, when $\text{try}(\text{sum}) @ms = \text{sum} @ms$. Let $ms = (\mathcal{B}, p)$ where p has been reduced to the form $\sum_{i=1}^n p_i$ and p_i are basic plans (composed by only the “;” operator). Since sum is commutative, we have that $\text{sum} @ms = \{(B, \sum_{j=1}^k p_{i_j}) \mid \forall k, i_j \in \{1, \dots, n\}\}$, i.e., any possible combination of p_i . On the other hand, $\text{sum}! @ms = \{(B, p_i) \mid i \in \{1, \dots, n\}\}$ which is clearly included in $\text{sum} @ms$. ■

3.4.4. THEOREM (PARTIAL CORRECTNESS & COMPLETENESS). *Given ms a mental state, T a test case we have that $s(T) @ms = T @ms$.*

Proof. We consider only the strategy do . The proof for the compositions follows from the definitions of the strategies. We proceed, by showing, as usually, a double inclusion.

“ \subseteq ”: By the definition of $\text{do}(a)$ we have that the result of applying it on ms is:

$$Res = \underbrace{\text{tauClosure} @ (o\text{-act}[o \leftarrow a] @ \underbrace{\text{tauClosure} @ ms}_{Res'})}_{Res''}$$

If the normalisation strategy “!” from the definition of tauClosure terminates, then by definition, there exists an $i \geq 0$ s.t.:

$$Res_i = i\text{-act} @ (\text{try}(\text{fail-act}) @ (\text{try}(\text{sum}) @ Res_{i-1}))$$

and for any $ms_i \in Res_i$ we have that $i\text{-act} @ (\text{try}(\text{fail-act}) @ (\text{try}(\text{sum}) @ ms_i))$ is empty (1). Thus, we can construct the computation:

$$ms_0 \xrightarrow{\tau^*} ms_1 \xrightarrow{\tau^*} \dots \xrightarrow{\tau^*} ms_{i-1} \xrightarrow{\tau^*} ms_i$$

where we take $ms_j \in Res_j$ with $j \leq i$, ms_0 as ms and $*$ denotes at most 3 τ steps, corresponding to the 3 possible rule labels for τ steps. By the definition of τ closure, Res' is the union of $try(fail-act) @ (try(sum) @ Res_i)$. This implies that any $ms' \in Res'$ is obtained from a ms_i after eventually applying sum and $fail-act$. From (1) we have that from ms' it is not possible to apply $i-act$. Furthermore, by the lemmas, whatever state can be reached from ms' by sum and $fail-act$ is already in Res' . Thus, $ms \Rightarrow ms'$.

By definition, Res'' is empty iff $o-act[o-a \leftarrow a] @ ms'$ fails for any element $ms' \in Res'$. That is, if Res'' is empty then $ms \not\stackrel{a}{\Rightarrow} ms'$ and thus $a@ms$ returns the empty set.

If Res'' were not empty, then for any element ms'' contained in it we have that $ms' \stackrel{a}{\Rightarrow} ms''$, thus $ms \Rightarrow \stackrel{a}{\Rightarrow} ms''$. Similarly, for any element $ms_f \in Res$ we have $ms'' \Rightarrow ms_f$ and from this we can conclude that $ms \stackrel{a}{\Rightarrow} ms_f$, thus ms_f is also an element of $a@ms$.

“ \supseteq ”: By the definition of \Rightarrow we have that, if no τ divergence, then there exists a $k \geq 0$ s.t. $ms \xrightarrow{\tau^k} ms_1$ and $ms_1 \not\rightarrow$. The trace τ^k can be divided in m packages of the form:

$$\sigma_m = (sum^{i_m}; fail-act^{j_m}; i-act^{l_m})^m,$$

with $\sum_m (i_m + j_m + l_m) * m = k$. By the lemmas we have that $sum^{i_m}; fail-act^{j_m}; i-act$ is obtained by applying the strategy $try(sum); try(fail-act); i-act$ (2). As for $i-act^{l_m-1}$, it is obtained by $(try(sum); try(fail-act); i-act)^{l_m-1}$ (3). If successive applications of $i-act$ are possible then neither $fail-act$ nor sum is applicable (at most one of $i-act$, $fail-act$, sum is enabled at a time) thus trying to applying them is harmless, i.e., does not change the state. Repeating m times the same argument from (2) + (3) and taking into account that we have that sequences σ_m where l_m is 0 are mapped to $try(sum); try(fail-act)$ we can derive that $ms_1 \in \tau$ closure $@ ms$ (4).

If $ms_1 \stackrel{a}{\Rightarrow} ms'$, then $ms' \in o-act[o-a \leftarrow a] @ ms_1$. Applying a similar reasoning for ms' we obtain (4'): $ms_2 \in \tau$ closure $@ ms'$. In consequence, we have that if $ms \stackrel{a}{\Rightarrow} ms_2$ then also $ms_2 \in do(a) @ ms$.

If $ms_1 \not\stackrel{a}{\Rightarrow} ms'$, then $o-act[o-a \leftarrow a] @ ms_1$ fails, thus this is also the case for $do(a)$. ■

Observe that in our proof we consider only finite computations. Thus, infinite computations do not violate the result. Since τ divergence is undecidable for BUpL agents, we cannot provide conditions such that test drivers terminate for all test cases. The most we can do, with respect to divergent computations, is to state the following proposition as a consequence of the above result:

3.4.5. COROLLARY (DIVERGENCE). *If the application of $s(T)$ diverges then so does T .*

We conclude with an illustration of an application of a test case to the BUpL agent described in the beginning of the section in Figure 3.2. We consider the test whether $done(2)$ appears in the belief base after executing $move(2, 0, 1)$. The corresponding test driver is the strategy $do(move(2, 0, 1)); check(done(2))$. The application of the strategy fails, meaning that the agent is not conformant with the test case.

Part II

**Refinement of Multi-Agent
Systems**

Chapter 4

From Agents to Multi-Agent Systems

If in the previous chapters it was enough to refer to an agent by its current mental state, this is no longer the case when considering multi-agent systems. This is why we associate with each agent an identifier and we consider a multi-agent system as a finite set of such identifiers. We further denote a state of a multi-agent system by $\mathcal{M} = \{(i, ms_i) \mid i \in \mathcal{I}\}$, where \mathcal{I} is the set of agent identifiers and ms_i is a mental state for the agent i . We abstract from *what is* the mental state of an agent. The choice of representation is not relevant, we only need to consider that the way to change (update) the mental state of an agent is by performing actions. However, for illustration purposes, we will instantiate such generic ms_i by either a BUnity or a BUpL mental state whenever the distinction is necessary.

We further note that considering the behaviour of a multi-agent system as simply the sum of the behaviours of individual agents is a too unrealistic idea since interaction is ignored. In this respect, we understand interaction as *coordination* and we look at it from two orthogonal perspectives with respect to the dichotomy action/state.

4.1 Classifying Coordination

We see *action-based* coordination as a mechanism to force certain groups of agents to execute certain actions synchronously while imposing to other groups certain restrictions or particular orders, i.e., scheduling policies on action execution. This is what inspired us to call them choreographies. One can picture them as global protocols which dictate the way agents behave by imposing ordering and synchrony constraints on their action executions. They represent *exogenous* coordination patterns and they can be seen as an alternative to message passing communication, with the potential advantage of not needing to establish a “common communication language”. Choreographies are useful in scenarios where *action synchrony* is more important than *data*. Choreographies are the subject of Section 4.2. Somewhat related to action-based coordination is the so called *channel-based* coordination. An already standard example in the coordination community is the language called Reo[Arb04]. The language is implemented as a Java software which makes it possible to design connectors by combining basic channels. These connectors can be seen as implementing sophisticated coordination patterns for agents. To have a complete experiment we take the Java 2APL platform

for designing agents and we propose a small tool integration exercise. Since the experiment is at a more practical level, not to interrupt the line of reasoning, we postpone the discussion to the last part of the thesis, in Chapter 6.

At a different¹ level, in Section 4.4.4, we look at what we will call, for simplicity, though by abuse of terminology, normative mechanisms as a way to enforce states, and not actions. More precisely, we describe a programming language that is designed to facilitate the implementation of norm-based organisation artifacts. Such artifacts refer to norms as a way to signal when violations take place and sanctions as a way to respond (by means of punishments) in the case of violations. In this context, a norm-based artifact observes the actions performed by the individual agents, determines their effects in the environment (which is shared by all individual agents), determines the violations caused by performing the actions, and possibly, imposes sanctions. Thus a normative artifact can be used to enforce the system to be in a specific, i.e., non-violating, state. Though the concepts we work with are simple, a couple of design decisions need to be considered. We can, for example, describe different scheduling strategies for the application of norms. From these strategies, if implemented directly into the language, different semantics arise, each characterising a different type of normative system. For instance, in the extreme case of an “autocratic agent society” each action an agent performs is followed by an inspection of the normative rules which might be applicable. At the other extreme, in a “most liberal society” the monitoring mechanism runs as a separate thread, independent of the executions of the agents. Such technicalities we discuss in more detail in Section 4.4.4. We further address normative properties like enforcement, regimentation, and their connection to different agent societies. For example, while in autocratic societies certain correctness (in terms of safety) properties are modelled by definition, this is no longer the case in liberal societies with infinite executions. This implies that we need to consider additional fairness constraints in order to ensure the well-behaviour of the systems.

A more expressive framework can be obtained when we extend both action-based coordination and normative mechanisms by explicitly modelling time. Time is an issue of concern since it allows one to model deadlines, timeouts, action scheduling or dynamic behaviour. Our approach in modelling time consists of adapting the theory of timed automata[Alu99]. There, time is modelled as clocks denoted by real-valued variables. Initially, all the clock variables are initialised with zero. They increase synchronously at the same uniform rate, counting time with respect to a fixed global time frame. Clocks are understood as fictitious, invented to express the timing properties of the system. We equip both agents and choreographies with clocks. In this way it is possible to model clock constraints which can (1) time restrict action execution, to force action execution to happen before certain time invariants are violated, (2) enforce delays between actions and (3) enable the sanctioning of delays, for example, postponing to pay a fine, or we can cancel the application of sanctions when certain deadlines have passed. Both extensions are discussed in Section 4.3 and 4.5.

Action-based (resp. channel-based) coordination artifacts and normative ones are orthogonal since neither choreographies nor coordination patterns imposed by Reo connectors are suitable for expressing organisational concepts like norms. Each concerns with distinct is-

¹It is tempting to see it as a “higher”, however some might argue that this is not distinctive feature differentiating between actions and states.

sues: coordination artifacts enforce specific *actions* to be executed while normative artifacts enforce the system to be in a specific, e.g., non-violating, *state*. Thus they have different, non-comparable expressive power. This is why we use the classification low-level vs high-level coordination. Since the expressive power is not the same we discuss the combination of their timed versions in Chapter 4.6.

4.2 Action-Based Coordination

Introducing coordination while respecting the autonomy of the agents is still a challenge in the design of multi-agent systems. The mechanisms we introduce, i.e., choreographies, are suitable in scenarios where synchronisation is important, take, for example, two agents being assigned the task of lifting a table together. To justify the choice of the naming, we imagine the setting of a ballet play where the main actors are the agents. In this context, agents may be able to perform many actions. However, they are supposed to perform precisely those actions which are indicated by the choreography. Once the agents adopt their roles, the play goes on without further assistance from any central coordinator. Thus, the advantage of the infrastructures we propose lies in their *exogenous* feature. This implies that the maintenance, i.e., the update of the agent's internals, i.e., mental states is separated from the coordination pattern. Consequently, nobody changes the agent's beliefs but itself. Besides that choreographies are oblivious to mental aspects, they control without having to know the internal structure of the agent. For example, whenever a choice between plans needs to be taken, a BUPL agent is free to make its own decision. The degree of freedom can be seen also in the mechanism for handling action failures. The agent chooses one among possibly many available repair rules without being constraint by the choreography. In these regards, the autonomy of agents executed with respect to choreographies is preserved.

4.2.1 Choreographies

For the ease of presentation, we introduce choreographies as regular expressions. The basic choreographic elements are pairs (i, a) which denote that the agent with identifier i should perform the action a . These pairs can be combined by sequence, parallel, choice or Kleene operators, with the usual meaning: $(i_1, a_1); (i_2, a_2)$ models orderings, agent i_1 executes a_1 which is followed by agent i_2 executing a_2 ; $(i_1, a_1) \parallel (i_2, a_2)$ models synchronisations between actions, agent i_1 executes a_1 while i_2 executes a_2 ; $(i_1, a_1) + (i_2, a_2)$ models non-deterministic choices, either i_1 executes a_1 or i_2 executes a_2 ; $(i, a)^*$ models iterated execution of a by i . The operators respect the usual precedence relation². Further, since “ \parallel ” and “ $+$ ” are associative and commutative, for simplicity, we use the notation $op_{\mathcal{I}}(i, a_i)$ to denote $(i_1, a_{i_1}) op \dots op (i_j, a_{i_j})$ where $op \in \{\parallel, +\}$, $\mathcal{I} = \{i_1, \dots, i_j\}$ and $j \geq 2$. The BNF grammar defining a choreography is as follows:

$$\begin{aligned} l_a &::= (i, a) \mid (i, x_a) \mid l_a \parallel l_a \\ ch &::= l_a \mid ch + ch \mid ch; ch \mid ch^* \end{aligned}$$

²If we denote \leq_p the precedence relation, then we have ‘ $+$ ’ \leq_p ‘ \parallel ’ \leq_p ‘ $;$ ’ \leq_p ‘ * ’

where x_a denotes an *action variable*. We use the naming convention that action variables are denoted by small letters with a as subscript (x_a, y_a, z_a, \dots). They are meant to be placeholders for action names. Action variables are seen as global static variables, thus, once bound, their value cannot be changed. The binding is according to the actions that an agent is enabled to execute at a given time. Variable bindings are recorded as action substitutions. The application of action substitutions to action variables is the same as for general expressions: given x_a an action variable and θ an action substitution containing $[x_a/a]$, the effect of the application of θ to x_a , denoted by $x_a\theta$, is a . If θ does not contain any $[x_a/a]$ then $x_a\theta$ equals x_a .

We note that the BNF syntax of choreographies allows parallel compositions only at the level of basic choreographic elements l_a . This is because our intention is to synchronise agents performing actions and not choreographies. As an example, we consider the following choreographic definition:

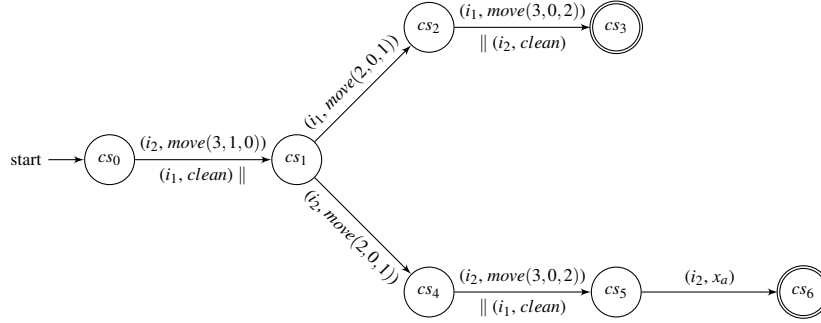
$$\begin{aligned} ch = & (i_1, \text{clean}) \parallel (i_2, \text{move}(3, 1, 0)); \\ & (i_1, \text{move}(2, 0, 1)); ((i_1, \text{move}(3, 0, 2)) \parallel (i_2, \text{clean})) + \\ & (i_2, \text{move}(2, 0, 1)); ((i_2, \text{move}(3, 0, 2)) \parallel (i_1, \text{clean})); (i_2, x_a). \end{aligned}$$

The choreography specifies that two agents i_1, i_2 work together in order to build the tower 123 and furthermore, that while one is building the tower the other one is cleaning the floor, all ending with i_2 executing no matter what action. More precisely, the definition of the choreography says that first i_2 deconstructs the initial tower (by moving the block 3 on floor) while i_1 is synchronously cleaning; next, either i_1 constructs the final tower while i_2 cleans or the other way around; in this latter case, after i_2 finishes the tower and i_1 the cleaning, i_2 executes any action and this leads the system to a final state. Further variations (like for example, in the case of a higher tower, one agent builds an intermediate shorter tower leaving the other to finish the construction) are left to the imagination of the reader.

The semantics of choreographies is given in terms of transition systems. These transition systems are in fact automata (equipped with final states) accepting choreographies. Such automata always exist as their construction is the one used for regular expressions. The standard approach from [MY60, HMRU00] is based on induction³ on the structure of the regular expression. We adapt this construction to show how it works for choreographies. Given a choreography ch , we denote by A^{ch} the associated automaton. The automaton associated with a basic choreography l_a has two states and one transition labelled with l_a . Given A^{ch_1} and A^{ch_2} the automata associated with the choreographies ch_1 and ch_2 , the automaton $A^{ch_1;ch_2}$ is the one obtained by concatenating A^{ch_1} and A^{ch_2} . As an example, Figure 4.1 illustrates the transition system corresponding to the choreography ch defined above.

We denote by $A^{ch} \otimes \mathcal{I}$ the synchronised product of a choreography ch and a multi-agent system \mathcal{I} . The states of $A^{ch} \otimes \mathcal{I}$ are pairs $\langle (cs, \theta), \mathcal{M} \rangle$ where (cs, θ_{ch}) is a *choreography state with a substitution* recording a choreography state cs of A^{ch} and an action substitution θ_{ch} , and \mathcal{M} is a state of the multi-agent system \mathcal{I} . The transition rule for $A^{ch} \otimes \mathcal{I}$ is given in Figure 4.2. There, cs, cs' are states of A^{ch} , $\theta_{ch}, \theta'_{ch}$ are action substitutions, a_j denote instantiated actions $x_{a_j}\theta'_{ch}$, \mathcal{J} is a subset of \mathcal{I} , ms_j, ms'_j are mental states of agent j and $\mathcal{M}, \mathcal{M}'$ are states of the multi-agent system with \mathcal{M}' being $\mathcal{M} \setminus \{(j, ms_j) \mid j \in \mathcal{J}\} \cup \{(j, ms'_j) \mid j \in \mathcal{J}\}$. The

³The reader can refer to [Brz64] for a direct deterministic construction using the derivatives of a given regular expression.

Figure 4.1: The Automaton A^{ch} Associated to the Choreography ch

notation $ms_j \xRightarrow{a_j} ms'_j$ represents that agent j performs action a_j (eventually with τ steps) in ms_j resulting in ms'_j . “Eventually τ steps” is needed for agents performing internal actions, like making choices among plans or handling failures in the case of BUP agents. In the case of agents “in the style of BUnity”, \xRightarrow{a} is simply \xrightarrow{a} since Bunity agents do not have τ steps. By abuse of notation, we use the construction $Sols(\bigwedge_j (E(ms_j), x_{a_j} \theta_{ch}))$ to denote the

set of solutions for matching $x_{a_j} \theta_{ch}$ against the set of names of actions $E(ms_j)$. We recall that in Section 3.4 we defined $E(ms)$ as the set of actions that are enabled to be executed from ms , $E(ms) = \{a \mid \text{there exists } ms' \text{ s.t. } ms \xRightarrow{a} ms'\}$. We also recall that in general, $x_a \theta$ can be either (1) a variable x_a or (2) an action name a if θ contains $[x_a/a]$. This means that matching $x_a \theta$ against a set of action names E can be an action substitution (if (1)), the identity if $x_a \theta = a$ and a is in E or a clash if a is not in E (if (2)). With this, we define $Sols(E, x_a \theta)$ as $\{\theta' \mid x_a \theta \theta' \in E\}$.

$$\boxed{
 \begin{array}{c}
 cs \xrightarrow{\parallel_{\mathcal{J}}(j, x_{a_j})} cs' \quad \theta'_{ch} \in Sols(\bigwedge_j (E(ms_j), x_{a_j} \theta_{ch})) \quad \bigwedge_{j \in \mathcal{J}} ms_j \xRightarrow{a_j} ms'_j \\
 \hline
 \langle (cs, \theta_{ch}), \mathcal{M} \rangle \xrightarrow{l} \langle (cs', \theta_{ch} \theta'_{ch}), \mathcal{M}' \rangle \quad (\text{sync-act})
 \end{array}
 }$$

Figure 4.2: The Transition Rule for $A^{ch} \otimes \mathcal{I}$

The transition rule (*sync-act*) shows the changes in the multi-agent system and in the choreography state with substitution. With respect to the state of the multi-agent system, the transition says that only the agents from the subset $\{ms_j \mid j \in \mathcal{J}\}$ are allowed to execute actions while the other ones remain unchanged. The new state of the multi-agent system reflects precisely the updates of action substitutions and the updates of mental states of the individual agents. With respect to the choreography state with substitution, the transition

says that the choreography state changes accordingly to the transition from A^{ch} and that all action variables x_{aj} are bound to a ground action name corresponding to one of the actions that agent j is enabled to execute. This binding is recorded in θ'_{ch} such that whenever x_{aj} appears in a choreography label from A^{ch} it will be substituted by the binding.

4.3 Timed Choreographies

In order to model timed choreographies we adapt the theory of timed automata [Alu99]. A *timed automaton* is a finite transition system extended with real-valued clock variables. Time advances only in states since transitions are instantaneous. Clocks can be reset at zero simultaneously with any transition. States and transitions have *clock constraints*, defined by the following grammar:

$$\phi_c ::= x_c \leq t \mid t \leq x_c \mid x_c < t \mid t < x_c \mid \phi_c \wedge \phi_c,$$

where $t \in \mathbb{Q}$ is a constant and x_c is a clock variable⁴. When a clock constraint is associated with a state, it is called *invariant*, and it expresses that time can elapse in the state as long as the invariant stays true. When a clock constraint is associated with a transition, it is called *guard*, and it expresses that the action may be taken only if the current values of the clocks satisfy the guard.

To record clock values one uses clock interpretations. A *clock interpretation* v for a set of clocks λ assigns a real value to each clock. A clock interpretation v is said to satisfy a clock constraint ϕ_c , $v \models \phi_c$, if and only if ϕ_c evaluates to true according to the values given by v . For $\delta \in \mathbb{R}$, $v + \delta$ denotes the clock interpretation which maps every clock $x_c \in \lambda$ to the value $v(x_c) + \delta$. For any $\lambda_1 \subseteq \lambda$, $v[\lambda_1 := 0]$ denotes the clock interpretation which assigns 0 to every $x_c \in \lambda_1$ and agrees with v over the other clocks.

In our multi-agent setting, *timed choreographies* are meant to impose time constraints on the actions executed by the agents. Syntactically, the BNF notation of timed choreographies extends the notation for the untimed ones:

$$\begin{aligned} l_\delta &::= x_c \leq t \\ l_a &::= (i, a) \mid (i, x_a) \mid (l_a \parallel l_a) \\ ch &::= l_\delta \mid l_a \mid (\phi_c, l_a, \lambda) \mid ch; ch \mid ch + ch \mid ch^* \end{aligned}$$

where l_δ denotes an additional elementary choreography for passing the time by delaying clocks. Timing the synchronisation is modelled by means of choreography states (ϕ_c, l_a, λ) , that is, by surrounding l_a with a clock constraint ϕ_c and a set of clocks λ to be reset. As we will see in the semantics, the purpose of the constructions (ϕ_c, l_a, λ) is (1) to time constrain the execution of the action denoted by l_a and (2) to update the clock valuation by resetting the clocks from λ .

We model timed choreographies as timed automata. The construction is similar to the one from choreographies to automata. The only additional construction corresponds to l_δ basic choreographies which instead of being labels on new transitions are associated to states to denote invariants. We consider as an example the following choreography *t-ch*:

⁴We use the naming convention of denoting clock variables by small letters with c as a subscript (x_c, y_c, z_c, \dots) in order to distinguish them from action variables

$$t\text{-}ch = ((i_1, \text{clean}), x_c := 0) \parallel (i_2, \text{move}(3, 1, 0)); (x_c \leq 5); \\ (i_2, \text{move}(2, 0, 1)); ((i_1, \text{move}(3, 0, 2)) \parallel (x_c > 6, (i_2, \text{clean}))).$$

The corresponding timed automaton is depicted in Figure 4.3. It shows that there is a single clock x_c . The initial state cs_0 has no invariant constraint and this means that an arbitrary amount of time can elapse in cs_0 . The clock x is always reset with the transition from cs_0 to cs_1 . Corresponding to a l_δ label, $x_c \leq 5$ is associated to the state cs_1 as an invariant. It ensures that the synchronous actions *clean* and *move*(3, 1, 0) must be executed within 5 units of time. The guard $x_c > 6$ associated with the transition from cs_2 to cs_3 ensures that the agents cannot spend an indefinite time in cs_2 because they must finish their tasks after 6 units of time.

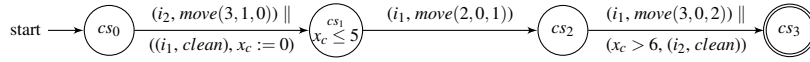


Figure 4.3: The Timed Automaton A^{ch} Associated to the Choreography $t\text{-}ch$

For convenience, we use A^{ch} instead of ch or $t\text{-}ch$. We define the semantics of timed choreographies A^{ch} by means of transition systems where the states are denoted as $\langle cs, v \rangle$ with cs being a state of A^{ch} and v the current clock interpretation. The transition rules are with respect to the transition labels of A^{ch} , that is, corresponding to delay (l_δ) and to agents' actions (l_a). These are illustrated in Figure 4.4. The first rule says that the choreography can pass time as long as the new valuation does not violate the invariant $I(cs)$ associated with the state cs . The second rule says that for any label $(\phi_c, \parallel_{\mathcal{J}}(ms_j, x_{aj}), \lambda)$ in A^{ch} we construct a transition labelled $\parallel_{\mathcal{J}}(ms_j, x_{aj})$ from $\langle cs, v \rangle$ only if ϕ_c is satisfied by the current clock interpretation v and if after resetting in v the clocks from λ the new interpretation v' does not violate the invariant associated with cs' .

$$\begin{aligned} & \langle cs, v \rangle \xrightarrow{\delta} \langle cs, v + \delta \rangle \text{ if } v + \delta \models I(cs) \text{ for any } \delta \in \mathbb{R}^+ \\ & \langle cs, v \rangle \xrightarrow{\parallel_{\mathcal{J}}(j, x_{aj})} \langle cs', v' \rangle \text{ if } cs \xrightarrow{\phi_c, \parallel_{\mathcal{J}}(j, x_{aj}), \lambda} cs', v \models \phi_c, v' := (v[\lambda := 0]) \text{ and } v' \models I(cs'). \end{aligned}$$

Figure 4.4: Transition Rules for Timed Choreographies

Having *timed choreographies*, however, does not make much sense without having time in the agents' programs. In what follows we illustrate how to extend BUnity and BUPL with time. We begin with some general remarks. We see basic actions as a *common ontology* shared by all agents. Since the nature of basic actions does not specify *when* to be executed, our extension is thought such that the ontology remains timeless and “when” becomes a specific part of the syntax of the language. In this regard, we consider that agents have a set of local clocks and that clock valuations can be performed by an observer. We further pose the problem of how agents make use of clocks. We recall the design principle: “the specification

of basic actions does not come with time”, thus actions are instantaneous. This implies that, in order to make the time pass, we need to extend the syntax of the agent languages with new application specific constructions such that the ontology of basic actions remains timeless (basic actions being specified only in terms of pre/post conditions). This is why we introduce *delay actions*, $\phi \rightarrow I$, where ϕ is a query on the belief base and I is an invariant like $x_c \leq 1$. Basically, their purpose is to make time elapse in a mental state where certain beliefs hold. As long as the invariant is true, the agent can stay in the same state while time passes. We refer to \mathcal{D} as the set of delays of either a BUnity or a BUpL agent. This is because, as it is the case for basic actions, delays are syntactical constructions belonging to both BUpL and BUnity languages. In what follows, we discuss the time extension for each language separately.

4.3.1 Timed BUnity

Extending BUnity with time reduces to considering time extensions for BUnity triggers. First, the queries of triggers are defined both on belief bases and clock valuations. Second, triggers specify the set of clocks to be reset after the execution of basic actions. Their syntax becomes $\{\phi \wedge \phi_c\} \triangleright do(a), \lambda$. Timed triggers are meant to say that if certain beliefs ϕ hold in the current mental state of a BUnity agent (as before) and additionally, certain clock constraints ϕ_c are satisfied, then the basic action a is executed and the clocks from the set λ are reset to 0. Taking into account the previous discussion of the mechanism of delay actions, the corresponding changes in the semantics are reflected in Figure 4.5.

$$\boxed{
 \begin{array}{c}
 \frac{\phi \rightarrow I \quad \mathcal{B} \models \phi}{(\delta \in \mathbb{R}_+)(\mathbf{v} + \delta \in I)} \quad (delay) \qquad \frac{\{\phi \wedge \phi_c\} \triangleright do(a), \lambda \quad \mathbf{v} \in \phi_c \quad \mathcal{B} \xrightarrow{a\theta} \mathcal{B}'}{(\mathcal{B}, \mathbf{v}) \xrightarrow{a\theta} (\mathcal{B}', \mathbf{v}[\lambda := 0])} \quad (t-act)
 \end{array}
 }$$

Figure 4.5: Transition Rules for Timed BUnity

In the transition rule $(t-act)$, λ is the set of clocks reset by performing action a and \mathbf{v} represents the current clock valuations. We use the notation $\mathbf{v} \in I$ (resp. $\mathbf{v} \in \phi_c$) to say that the clock valuations from \mathbf{v} satisfy the invariant I (resp. the constraint ϕ_c). When ϕ_c is absent we consider that trivially $\mathbf{v} \in \phi_c$ holds. We make a short note that our design decision is to separate the implementation of delays from the one of triggers. This is because a construction like $\{\phi\} \triangleright I, do(a), \lambda$ is ambiguous. If ϕ holds, it can either be the case that time elapses with respect to the invariant I and a is suspended, or that a is immediately executed. However, it sometimes is important to *ensure* that “time passes in a state”, instead of leaving this only as a non deterministic choice.

To illustrate the above constructions we recall the BUnity agent i_a from Figure 2.4. We basically extend the BUnity agent such that the agent has one clock, be it y_c , which is reset by triggers, and such that the agent can delay in given states, thus letting the time pass.

Figure 4.6 shows a possible timed extension. The clock y_c is reset after either performing

$$\begin{aligned}
\mathcal{A}' &= \{ \quad \top \triangleright (do(clean), y_c := 0), \neg on(2, 1) \triangleright do(move(2, 0, 1)), \\
&\quad \neg on(3, 2) \wedge on(2, 1) \triangleright do(move(3, 0, 2)), \\
&\quad \neg(on(2, 1) \wedge on(3, 2)) \triangleright (do(move(x, y, 0)), y_c := 0) \quad \} \\
\mathcal{D} &= \{ \quad on(3, 0) \vee cleaned \rightarrow (y_c < 9), on(2, 1) \vee cleaned \rightarrow (y_c < 10) \quad \}
\end{aligned}$$

Figure 4.6: Extending i_a with clock constraints

clean or moving a block on the floor. The agent can delay until the clock evaluates to 9 (resp. 10) units of time after moving 3 on the floor (resp. 2 on 1).

4.3.2 Timed BUpL

The timed extension of BUpL concerns changing plans such that previous calls $a; p$ are replaced by $(\phi_c, a, \lambda); p$ and $(\phi \rightarrow I); p$, where ϕ_c is time constraining the execution of action a and λ is the set of clocks to be reset. To simplify notation, if clock constraints and clock resets are absent we use a instead of (a) .

We make the remark that if previously actions failed when certain beliefs did not hold in a given mental state, it is now the case that actions fail also when certain clock constraints are not satisfied. Consider, for example, the plan $((x_c < 1), a, [x_c := 0]); ((x_c > 2), b, \emptyset)$. There is no delay action between a and b , thus the time does not pass and x_c remains 0, meaning that b cannot be executed. Such situations are handled by means of the general semantics of the repair rules. There are two possibilities: either to execute an action with a time constraint that holds, or to make time elapse. The latter is achieved by triggering a repair rule like $true \leftarrow \delta$, where for example δ is a delay action $true \rightarrow true$ which allows an indefinite amount of time to pass. The corresponding changes⁵ in the semantics are reflected in Figure 4.7.

$ \frac{p = (\phi \rightarrow I); p' \quad \mathcal{B} \models \phi \quad (\delta \in \mathbb{R}_+)(v + \delta \in I)}{(\mathcal{B}, p, v) \xrightarrow{\delta} (\mathcal{B}, p', v + \delta)} \quad (delay) $	$ \frac{p = (\phi_c, a, \lambda); p' \quad v \in \phi_c \quad (\mathcal{B}, p) \xrightarrow{a\theta} (\mathcal{B}', p'\theta)}{(\mathcal{B}, p, v) \xrightarrow{a\theta} (\mathcal{B}', p'\theta, v[\lambda := 0])} \quad (t-act) $
--	---

Figure 4.7: Transition Rules for Timed BUpL

To see a concrete example, we recall the BUpL agent from Section 2.3.3. We consider two delay actions $true \rightarrow (y_c < 9)$ and $true \rightarrow (y_c \leq 10)$. We further make the delays and the clock resets transparent in the plans. The plan *cleanR* changes to $(true, clean, y_c := 0); cleanR$ such that the clock y_c is reset after a clean action. The plan *rearrange* (x_1, x_2, x_3) changes to $true \rightarrow (y_c < 9); move(x_1, 0, x_2); true \rightarrow (y_c \leq 10); move(x_3, 0, x_1)$ such that time passes between moves.

⁵With respect to the untimed BUpL rules, we only present the extension for the rule *(act)*. In the case of timed repair rules the extension is similar and thus we do not include it.

The observable behaviour of either timed BUnity or BUpL agents is defined in terms of timed traces. A *timed trace* is a (possibly infinite) sequence $(t_1, a_1) (t_2, a_2) \dots (t_i, a_i) \dots$ where $t_i \in \mathbb{R}_+$ with $t_i \leq t_{i+1}$ for all $i \geq 1$. We call t_i a *time-stamp* of action a_i since it denotes the absolute time that passed before a_i was executed. We then have that a timed BUnity or BUpL agent computation over a timed trace $(t_1, a_1)(t_2, a_2) \dots (t_i, a_i) \dots$ is a sequence of transitions:

$$ms_0, v_0 \xrightarrow{\delta_1} \xrightarrow{a_1} ms_1, v_1 \xrightarrow{\delta_2} \xrightarrow{a_2} ms_2, v_2 \dots$$

where ms_i is a BUnity (BUpL) mental state and t_i are satisfying the condition $t_i = t_{i-1} + \delta_i$ for all $i \geq 1$.

For example, a possible timed trace for either the timed BUpL or BUnity agent is $(0, \text{clean}), (7, \text{move}(3, 1, 0)), (8, \text{move}(2, 0, 1)), (9, \text{move}(3, 0, 2))$. It is, in fact, the case that any BUpL timed trace is also a BUnity timed trace, thus the two agents are again in a refinement relation. We elaborate more on timed refinement in Section 4.7.1.

4.3.3 Timed Multi-Agent Systems

In the new context of timed choreographies, we need to revise the semantics of multi-agent systems. First we consider the set $v^{\mathcal{M}}$ as representing the valuation of all of the clocks of all agents from \mathcal{M} , i.e., $v^{\mathcal{M}} = \{v_i \mid i \in \mathcal{I}\}$ ⁶. We say that $v^{\mathcal{M}} \models \phi_c$ is true whenever the valuations from $v^{\mathcal{M}}$ satisfy the clock constraint ϕ_c . For example, let us consider a multi-agent system with two agents, i_1 and i_2 with i_1 having two clocks x_c, y_c and i_2 having one clock z_c . Let us further assume that we “freeze” the system for an instant in a state where the clock interpretation v_1 of the first agent is $v_1(x_c) = 2, v_1(y_c) = 4$ and the clock interpretation v_2 of the second agent is $v_2(z_c) = 6$. We have that, in the clock interpretation $v^{\mathcal{M}}$, i.e., $v_1 \cup v_2$, the clock constraint $\phi_c = (x_c < 3) \wedge (z_c > 5)$ is satisfied however this is not the case for the clock constraint $\phi_c = (y_c < 3)$.

The transition rules for timed multi-agent systems running under the directions of timed choreographies are depicted in Figure 4.8. They follow from the transitions for timed choreographies (Figure 4.4) and extend the transition (*act*) for the untimed multi-agent systems (Figure 4.2). More precisely, the transition (*delay*) for passing time says that the whole system can delay δ units as long as the updated valuations do not violate the invariant of the current choreography state. The transition rule (*t-sync-act*) replaces the transition (*sync-act*). The changes reflect that, in addition, the new clock valuations should satisfy the invariant associated to the new choreography state in order to allow the transition to take place. We conclude with an illustration of one possible timed trace of a multi-agent system composed of two BUpL agents i_1 and i_2 having the same code described in Section 4.3.2. We consider that the system runs under the timed choreography from Figure 4.3. Recalling that the clock of the choreography was denoted by x_c and assuming that i_1 has a clock y_{1c} , resp. i_2 has a clock y_{2c} , we have the following computation:

⁶In order to have a well-defined interpretation $v^{\mathcal{M}}$ the set of clock variables of individual agents must be disjoint, i.e., each clock variable is unique.

$$\begin{array}{c}
\frac{\langle cs, v \rangle \xrightarrow{\delta} \langle cs, v + \delta \rangle \quad \bigwedge_i ((ms_i, v_i) \xrightarrow{\delta} (ms_i, v_i + \delta)) \quad (v^{\mathcal{M}} \cup v + \delta) \models I(cs)}{\langle (cs, v, \theta_{ch}), \mathcal{M} \rangle \xrightarrow{\delta} \langle (cs, v + \delta, \theta_{ch}), \mathcal{M}' \rangle} \text{ (delay)} \\
\\
\frac{\langle cs, v \rangle \xrightarrow{l := \|\mathcal{J}(j, x_{aj})} \langle cs', v' \rangle \quad \theta'_{ch} \in \text{Sols}(\bigwedge_j (E(ms_j), x_{aj} \theta_{ch})) \quad a_j := x_{aj} \theta'_{ch} \quad \bigwedge_j ((ms_j, v_j) \xrightarrow{a_j} (ms'_j, v'_j)) \quad v^{\mathcal{M}'} \cup v' \models I(cs')}{\langle (cs, v, \theta_{ch}), \mathcal{M} \rangle \xrightarrow{l} \langle (cs', v', \theta'_{ch}), \mathcal{M}' \rangle} \text{ (t-sync-act)}
\end{array}$$

Figure 4.8: Transitions Rules for Timed Multi-Agent Systems

$$((i_1, \text{clean}) \parallel (i_2, \text{move}(3, 1, 0)), x_c = 0, y_{1c} = 0, y_{2c} = 4), \quad (1)$$

$$((i_2, \text{move}(2, 0, 1)), x_c = 6, y_{1c} = 6, y_{2c} = 10), \quad (2)$$

$$((i_1, \text{move}(3, 0, 2)) \parallel (i_2, \text{clean}), x_c = 6, y_{1c} = 6, y_{2c} = 0), \quad (3)$$

where we have ignored substitutions from choreography states since in this particular case all the choreographic labels are ground. The snapshot represented by (1) illustrates that after i_1 performed *clean* both its clock y_{1c} and x_c have been reset to 0 while the clock of i_2 shows how much time has passed, i.e., 4 units. As a short note, we recall that the invariant associated to the choreography state cs_1 is $x_c \leq 5$ and that i_2 could delay at most 10 units before the next *move* action, thus the maximum time that y_{2c} could have shown is 9 units. Snapshot (2) illustrates that after i_2 performs *move*(2,0,1) the agents and the choreography delays for 5 units. This is due to the guard $x_c > 6$ from the transition between cs_2 and cs_3 . Finally, snapshot (3) illustrates that after the last moves the clock of i_1 has been reset due to the specification of the plan *cleanR*.

4.4 A Normative Language

In this section, we present a programming language that facilitates the implementation of multi-agent systems with norms. Individual agents are assumed to be implemented in a programming language, not necessarily known to the multi-agent system programmer, who is assumed to have a reference to the (executable) program of each individual agent. Most noticeably, it is not assumed that the agents are able to reason about the norms of the system since we do not make any assumptions about the internals of individual agents.

Agents perform their actions in an external environment which is part of and controlled by the organisation. The initial state of an environment can be implemented by means of a set of facts. In order to implement the effects of the external actions of individual agents in the

environment, we propose a programming construct by means of which it can be indicated that a set of facts should hold in the environment after an external action is performed by an agent. As external actions can have different effects when they are executed in different states of the environment, we add a set of facts that function as the precondition of those effects. In this way, different effects of one and the same external action can be implemented by assigning different pairs of facts, which function as pre- and postconditions, to the action. The multi-agent system organisation determines the effect of an action by the following mechanism: if the precondition holds in the current state of the environment (the execution of the action is enabled), then the state is updated with the facts which represent the postcondition.

We consider norms as being represented by counts-as rules [Sea95], which ascribe “institutional facts” (e.g. “a violation has occurred”) to “brute facts” (e.g. “the agent is in the train without a ticket”). In our framework, brute facts constitute the factual state of the multi-agent system organisation, which is represented by the environment (initially set by the programmer), while institutional facts constitute the normative state of the multi-agent system organisation. The institutional facts are used with the explicit aim of triggering system’s reactions (e.g. sanctions). As claimed in [GDM07] counts-as rules enjoy a rather classical logical behaviour. In our framework, the counts-as rules are implemented as simple rules that relate brute and institutional facts. It is important to note that the application of counts-as rules corresponds to the triggering of a monitoring mechanism since it signals which changes have taken place and what are the normative consequences of the changes.

Sanctions can also be implemented as rules, but follow the opposite direction of counts-as rules. A sanction rule determines what brute facts will be brought about by the system as a consequence of normative facts. Typically, such brute facts are sanctions, such as fines. Notice that in human systems sanctions are usually brought about by specific agents (e.g. police agents). This is not the case in our computational setting, where sanctions necessarily follow the occurrence of a violation if the relevant sanction rule is into place (comparable to automatic traffic control and issuing tickets). It is important to stress, however, that this is not an intrinsic limitation of the system since we do not aim at mimicking human institutions but rather providing the specification of computational systems.

4.4.1 Syntax

In order to represent brute and institutional facts in our normative multi-agent system programming language, we introduce two disjoint sets of first-order atoms `<b-atoms>` and `<i-atoms>` to denote these facts. Moreover, we use `<ident>` to denote a string and `<int>` to denote an integer. Figure 4.9 presents the syntax of the language in EBNF notation. A normative multi-agent system program `N-MAS_Prog` starts with a non-empty list of clauses, each of which specifies one or more agents. The list of agent specifications is preceded by the keyword `'Agents: '`. Unlike in non-normative MAS programming language, we do not specify the agents’ access relation to environments in these clauses because we assume that the access relations can and should be specified by means of norms and sanctions. In each clause, `<agentName>` is a unique name to be assigned to the individual agent that should be created, `<agentProg>` is the reference to the (executable) agent program that implements the agent, and `<nr>` is the number of such agents to be created (if the number is

```

<N-MAS_Prog> = "Agents:" ( <agentName> <agentProg> [<nr>] )+
               "Facts:" <bruteFacts>
               "Effects:" { <effect> }
               "Counts-As rules:" { <counts-as> }
               "Regimentation rules:" { <regimentation> }
               "Sanction rules:" { <sanction> };

<bruteFacts>   = <b-literals> ;
<effect>       = " { " <b-literals> " } " <actName> " { " <b-literals> " } " ;
<counts-as>    = <literals> "=>" <i-literals> ;
<regimentation> = <b-literals> "=>" viol⊥ ;
<sanction>     = <i-literals> "=>" <b-literals> ;
<agentName>    = <ident> ;
<agentProg>    = <ident> ;
<nr>           = <int> ;
<actName>      = <ident> ;
<b-literals>   = <b-literal> { " , " <b-literal> } ;
<i-literals>   = <i-literal> { " , " <i-literal> } ;
<literals>     = <literal> { " , " <literal> } ;
<literal>      = <b-literal> | <i-literal> ;
<b-literal>    = <b-atom> | "not" <b-atom> ;
<i-literal>    = "viol⊥" | <i-atom> | "not" <i-atom> ;

```

Figure 4.9: The EBNF syntax of normative multi-agent programs

greater than one, then the agent names will be indexed by a number). After the specification of individual agents, the initial state of the environment is specified as a set of first order literals denoting brute facts. The set of literals is preceded by the keyword 'Facts:'. The effects of an external action of an individual agent are specified by triples consisting of the action name, together with two sets of literals denoting brute facts. The first set specifies the states of the environment in which the action can be performed, and the second set specifies the effect of the action that should be accommodated in the environment. The list of the effects of agents' external actions is preceded by the keyword 'Effects:'. A counts-as rule is implemented by means of two sets of literals. The literals that constitute the antecedent of the rule can denote either brute or institutional facts, while the consequent of the rules are literals that denote only institutional facts. This allows rules to indicate that a certain brute or institutional fact counts as another institutional fact. For example, speeding is a violation of traffic law (institutional fact), but this violation together with not paying your fine in time (brute fact) is considered as another violation (institutional fact). The list of counts-as rules is preceded by the keyword 'Counts-As rules:'. A regimentation rule is a special type of counts-as rule. The difference is that the antecedent is defined only on brute facts and the consequent is a specifically designated literal `viol⊥`. Regimentation rules are normative enabling conditions on top of external actions. They function as one look-ahead step, specifying when the execution of an action leads to a forbidden state of the environment, thus preventing it from taking place. The list of regimentation rules is preceded by the keyword 'Regimentation rules:'. Finally, the list of sanction rules can be specified in a normative multi-agent program. The antecedent of a sanction rule consists of literals denoting institutional facts while the consequent of a sanction rule consists of literals denoting brute facts. The list of sanction rules are preceded by the keyword 'Sanction rules:'.

Figure 4.10 presents a normative multi-agent system program that implements a small part of a train system. The program creates from the file `passenger_prog` one agent called `psg`. The `Facts`, which implement brute facts, determine the initial state of the shared environment. In this case, the set of brute facts is empty, meaning, for example, the agent is not at the platform and has no ticket⁷. The `Effects` indicate how the environment can advance in its computation, for instance, `psg` performing `enter` when not at the platform, results in `psg` being at the platform (with or without a ticket). The `Counts-As` rules determine the normative effects for a given state of the multi-agent system. In our case, the only count-as rule states that being at the platform without having a ticket is a specific violation (`viol_ticket(X)`). The rule functions as an *enforcement* mechanism [GDM07] and it is based on the idea of responding to a violation such that the system returns to an acceptable state. However, there are situations where stronger requirements need to be implemented, for example, where it is never the case that `psg` enters the train without a ticket. This is what we call *regimentation* and in order to implement it we consider the literal `viol⊥(X)` by means of regimentation rules. The operational semantics of the language ensures that `viol⊥(X)` can never hold during any run of the system. Intuitively, regimentation can be thought of as placing gates blocking an agent's action. Finally, the aim of `Sanction rules` is to determine the punishments that are imposed as a consequence of violations. In the example the

⁷In our framework we consider the closed world assumption, thus negation as failure.

violation of type `viol_ticket(X)` causes the sanction `fined(X, 25)` (e.g., a 25 EUR fine).

```

Agents:
    psg passenger_prog 1
Facts:
Effects:
    enter(X) = ({not at_platform(X)}, {at_platform(X)})
    buy-ticket(X) = ({not ticket(X)}, {ticket(X)})
    embark(X) = ({at_platform(X), not in_train(X)},
                  {not at_platform(X), in_train(X)})

Counts-As rules:
    at_platform(X), not ticket(X) => viol_ticket(X)
Regimentation rules:
    in_train(X), not ticket(X) => viol_|_(X)
Sanction rules:
    viol_ticket(X) => fined(X, 25)

```

Figure 4.10: An example of a Normative MAS file

4.4.2 Operational Semantics

The state of a normative multi-agent system is an extension of a multi-agent system state as defined in Chapter 4. More precisely, it consists of the state of the external environment, the normative state of the organisation, and the states of individual agents. We recall that we abstract away from the internal configuration of individual agents. The language of design is left to the choice of the programmer as long as it allows reasoning on the observable actions executed by agents.

4.4.1. DEFINITION. [Normative MAS Configuration] Let P_b and P_n be two disjoint sets of first-order atoms denoting brute and normative facts (including a predefined `viol_|_`), respectively. Let ms_i denote the configuration of individual agent i . The configuration of a normative multi-agent system consisting of a set of agents identified by \mathcal{I} is defined as $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$ where $\mathcal{M} = \{(i, ms_i) \mid i \in \mathcal{I}\}$, σ_b is a consistent set of ground literals from P_b denoting the brute state of the multi-agent system, and σ_n is a consistent set of ground literals from P_n denoting the normative state of the multi-agent system. \blacklozenge

Before presenting the transition rules for specifying possible changes between normative multi-agent system configurations, we fix some notation. The normative rules we consider, counts-as or sanction rules⁸, are defined as first order implications, $l = (\Phi \Rightarrow \Psi)$, with Φ and Ψ being sets, or equally conjunctions, of literals. We use $cond_l$ and $cons_l$ are used to indicate the condition Φ and the consequent Ψ of l , respectively. Given a set \mathbf{R} of normative rules and a set σ of ground atoms, we define the set of applicable rules in σ as:

⁸Counts-as and sanctions are usually considered as being context dependent. Our framework can be extended by considering both rule types in a non-monotonic way capturing their context dependence.

$$\text{App1}^{\mathbf{R}}(\sigma) = \{ (\Phi \Rightarrow \Psi)\theta \mid \Phi \Rightarrow \Psi \in \mathbf{R} \wedge \exists \theta \text{ s.t. } \sigma \models \Phi\theta \},$$

with θ being a ground substitution.

The ground closure of σ under \mathbf{R} , denoted as $\text{Cl}^{\mathbf{R}}(\sigma)$, is inductively defined as follows:

$$\text{Base : } \text{Cl}_0^{\mathbf{R}}(\sigma) = \sigma \cup (\bigcup_{l \in \text{App1}^{\mathbf{R}}(\sigma)} \text{cons}_l)$$

$$\text{Inductive Step : } \text{Cl}_{n+1}^{\mathbf{R}}(\sigma) = \text{Cl}_n^{\mathbf{R}}(\sigma) \cup (\bigcup_{l \in \text{App1}^{\mathbf{R}}(\text{Cl}_n^{\mathbf{R}}(\sigma))} \text{cons}_l).$$

We note that such a computation does not always reach a fixpoint. Not to interrupt the flow of reasoning, we postpone this discussion to Section 4.4.5.

We do not make any assumptions about the internals of individual agents. Therefore, for the operational semantics of normative multi-agent system we assume $ms_i \xrightarrow{a} ms'_i$ as being the transition of configurations for individual agent i . Given this transition, we can define a new transition rule to derive transitions between normative multi-agent system configurations.

4.4.2. DEFINITION. Let $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$ be a configuration of a normative multi-agent system. Let \mathbf{R}_c be the set of counts-as rules, \mathbf{R}_r^9 be the set of regimentation rules, \mathbf{R}_s be the set of sanction rules, and a be an external action. The transition rule for the derivation of normative multi-agent system transitions is defined in Figure 4.11. \blacklozenge

$\frac{\begin{array}{l} ms_i \xrightarrow{a} ms'_i \quad a = (\psi, \xi) \quad \theta \in \text{Sols}(\sigma_b, \psi) \quad \sigma'_b := \sigma_b \uplus \xi \theta \\ \text{App1}^{\mathbf{R}_r}(\sigma'_b) = \emptyset \quad \sigma'_n := \text{Cl}^{\mathbf{R}_c}(\sigma'_b) \setminus \sigma'_b \quad S := \text{Cl}^{\mathbf{R}_s}(\sigma'_n) \setminus \sigma'_n \quad \sigma'_b \cup S \not\models \perp \end{array}}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', \sigma'_b \cup S, \sigma'_n \rangle} \quad (ACS)$
--

Figure 4.11: The Transition Rule for Normative Multi-Agent Systems

In the transition (ACS) ms_i is the current state of agent i , i.e., $ms_i \in \mathcal{M}$, \mathcal{M}' reflects the changes with respect to the new states, i.e., $\mathcal{M}' = (\mathcal{M} \setminus \{ms_i\}) \cup \{ms'_i\}$, and \uplus is the function from Section 2.1 used to update the environment of a normative multi-agent system with the effects of an action performed by an agent. The transition rule captures the effects of performing an external action by an individual agent on both external environments and the normative state of the multi-agent system. First, the effect of a on the environment σ_b is computed. Then, the updated environment σ'_b is used to check whether $\text{App1}^{\mathbf{R}_r}(\sigma'_b)$ is empty. This means that an agent can execute an external action only if the execution does not result in a state containing viol_\perp . This captures exactly the regimentation of norms. Thus, once assumed that the initial normative state does not include viol_\perp , it is easy to see that the system will never be in a viol_\perp -state. The updated state of the environment σ'_b is further used to determine the new normative state σ'_n by applying all counts-as rules to σ'_b . The next step consists in adding to the environment state all possible sanctions by applying sanction rules to the new normative state of the system. The result is saved in S . Finally, both σ'_b and S are

⁹We consider \mathbf{R}_c and \mathbf{R}_r as separate sets in order to model regimentation more easily.

used to check whether no contradictory state is reached. This is denoted by the construction $\sigma'_b \cup S \not\models \perp$, with \perp as the logical boolean *false*, which stands for the case where from σ'_b and S one can conclude f and $\neg f$, with f being a ground fact.

4.4.3 Normative Properties

We would like to make sure that our language definitions fulfill some properties. Namely, we would be interested in whether the semantics of the language models the enforcement and the regimentation of norms. We recall that enforcing a norm means that if a violation occurs then a corresponding sanction is applied while regimenting a norm means that the associated violation can never occur.

We can express such concepts as LTL properties, $\text{enforcement}(c, s) = \text{cond}_c \wedge (\text{cons}_c \rightarrow \text{cond}_s) \rightarrow \Diamond \text{cons}_s$ and $\text{regimentation}(r) = \Box \neg \text{cond}_r$. On the one hand, the definition of *enforcement* says that for an arbitrary counts-as rule c with a valid antecedent (cond_c is true) and for a sanction rule s with the antecedent being implied by the consequence of c ($\text{cons}_c \rightarrow \text{cond}_s$) it is the case that the sanctioning will eventually be applied ($\Diamond \text{cons}_s$). On the other hand, the definition of *regimentation* says that for an arbitrary rule r from \mathbf{R}_r ($\text{cons}_r = \text{viol}_\perp$) it is never the case that the antecedent cond_r holds.

It is not difficult to see that the transition (ACS) models regimentation. This is because the execution of an action is performed only when the set of applicable regimentation rules is empty ($\text{App1}^{\mathbf{R}_r}(\sigma'_b) = \emptyset$), which means that no regimentation rule r has a true antecedent ($\neg \text{cond}_r$). However, this is not the case for enforcement and the reason is that the application of a sanction can enable the application of a previously not enabled counts-as rule. This is possible since the antecedents of counts-as are defined on both brute and normative facts. Thus though there is no change in the set of normative facts, the change in brute facts (due to the application of sanctions) might have as a follow-up the enabling of new counts-as rules. We note however, that such scenarios are more peculiar, even hard to implement, and that in general, the semantics models for most scenarios not only regimentation but also enforcement. How we can change the transition (ACS) such that it always models enforcement and other possible variations on the semantics are discussed in the next section.

4.4.4 From Totalism to Liberalism in Operational Semantics

The transition rule (ACS) gives an operational semantics which characterises agent societies implementing *almost* Orwell's like "1984" societies, where each single step is being supervised and faults are being handled accordingly. We say "almost" since there might arise cases when mistakes are being left unpunished, thus the societies are not "completely" vigilant. Consider a traffic scenario where an actor drives through the red light, thus violating the traffic law. Consequently, a fine is applied. Assume that this is done automatically by withdrawing a certain amount of money from the actor's account. It is then the case that not enough money in the account results in a new violation. This is under the supposition that the bank has a regulation specifying that the client must not go below a certain debt level, otherwise the client is added to the bank's black list and has to pay an additional fee. We note that this latter sanction rule can never be applied when the system runs with respect to

the transition (ACS). What happens is that after computing the closure of normative facts under counts-as rules and respectively the closure of brute facts under sanctions, the system changes state with no further check for new counts-as rules which are enabled by the update of brute facts. In the new state, by the definition of the semantics, previous normative facts play no role (this makes sense in most cases).

From the above scenario it follows that in certain circumstances the application of a sanction enables the execution of a new counts-as rule which should be taken into consideration. In order to implement such a requirement, we need to consider, with respect to the applicable norms and sanctions, the sequences defined on σ_n, σ_b satisfying the following recurrent relations:

$$\begin{aligned}\sigma_{n_i} &= \sigma_{n_{i-1}} \cup \bigcup_{l \in \text{App1}^{\mathbf{R}_c}(\sigma_{b_{i-1}})} \text{cons}_l \\ \sigma_{b_i} &= \sigma_{b_{i-1}} \cup \bigcup_{l \in \text{App1}^{\mathbf{R}_s}(\sigma_{n_{i-1}})} \text{cons}_l,\end{aligned}$$

where $i \geq 1$, $\sigma_{n_0} = \sigma_n \cup \bigcup_{l \in \text{App1}^{\mathbf{R}_c}(\sigma_b)} \text{cons}_l$ and $\sigma_{b_0} = \sigma_b$. We denote by σ_n^* (resp. σ_b^*) the limit of the sequence σ_{n_i} (resp. σ_{b_i}). We note that simply considering the closure $\mathbf{CI}^{\mathbf{R}_c \cup \mathbf{R}_s}$ is not enough since we cannot distinguish anymore between brute and normative facts. The new transition (ACS) is reflected in Figure 4.12. This new rule always models enforcement. However, there is also a price to pay since, as we have already mentioned, when computing fixpoints, limits of recurrent sequences in this case, one might run into the problem of non-termination. We further discuss this in Section 4.4.5.

$$\boxed{\frac{\begin{array}{l} ms_i \xrightarrow{a} ms'_i \quad a = (\psi, \xi) \quad \theta \in \text{Sols}(\sigma_b, \psi) \\ \sigma'_b := \sigma_b \uplus \xi \theta \quad \text{App1}^{\mathbf{R}_r}(\sigma'_b) = \emptyset \quad \sigma_b^* \not\models \perp \end{array}}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', \sigma_b^*, \sigma_n^* \rangle} \quad (A(CS)^*)}$$

Figure 4.12: A Totalitarian Semantics

What is distinctive to both (ACS) and $(A(CS)^*)$ is that the application of normative rules is performed in the same step with the execution of actions. We now consider the case when the process of applying normative rules is separated from the one for action execution. This gives rise to new variations on the operational semantics of normative multi-agent systems. The key concept is the scheduling of the monitoring mechanism, i.e., the application of normative rules. To illustrate this, we start with *traces*. We understand traces as sequences of *observables* with respect to the executions of a system. By observables we mean actions and normative rules. In what follows, we use the notation α for denoting an arbitrary action, $\gamma \in 2^{\mathbf{R}_c}$ for the set of counts-as rules which are applicable after the execution of α , and $\varsigma \in 2^{\mathbf{R}_s}$ for the set of sanctions which are applicable after the execution of the counts-as rules from γ . With this, traces of normative multi-agent systems are regular expressions defined on $\alpha, \gamma, \varsigma$. For example, the regular expression which characterises the traces of normative multi-agent systems running with respect to the transition (ACS) is $(\alpha\gamma\varsigma)^*$, after one action,

apply all valid counts-as rules and then all valid sanction rules. On the other hand, when running with respect to $(A(CS)^*)$ the traces are of the form $(\alpha(\gamma\zeta)^*)^*$.

Scheduling strategies give us the freedom to think of more relaxing societies. We show that such societies can be not only imagined but also implemented. For example, it suffices to consider a strategy where the application of sanctions is performed in a transition distinct from the one corresponding to the execution of actions and counts-as rules. This implies that the sanctioning mechanism runs independently, as a separate thread. We would then implement a more liberal society characterised by the scheduling strategy $((\alpha\gamma)^*\zeta)^*$. The illustrative situation is that of a video camera monitoring in a supermarket, or of a radar measuring the velocity of the passing vehicles. In such cases, sanctions do not necessarily follow immediately after the recording of an infraction. To make this transparent from the semantics means that the rule (ACS) splits into two rules depicted in Figure 4.13.

$$\begin{array}{c}
 \frac{
 \begin{array}{l}
 ms_i \xrightarrow{\alpha} ms'_i \quad a = (\psi, \xi) \quad \theta \in Sols(\sigma_b, \psi) \\
 \sigma'_b := \sigma_b \uplus \xi \theta \quad \text{App1}^{\mathbf{R}_r}(\sigma'_b) = \emptyset \\
 \sigma'_n := \text{C1}^{\mathbf{R}_c}(\sigma'_b) \setminus \sigma'_b
 \end{array}
 }{
 \langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{\alpha\gamma} \langle \mathcal{M}', \sigma'_b, \sigma_n \cup \sigma'_n \rangle
 } \quad (AC) \\
 \\
 \frac{
 S := \text{C1}^{\mathbf{R}_s}(\sigma_n) \setminus \sigma_n \quad \sigma_b \cup S \not\models \perp
 }{
 \langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{\zeta} \langle \mathcal{M}, \sigma_b \cup S, \sigma_n \rangle
 } \quad (S)
 \end{array}$$

Figure 4.13: Transition Rules for $((\alpha\gamma)^*\zeta)^*$ -semantics

In Figure 4.13, γ (resp. ζ) represents the set of all counts-as (resp. sanctions) that have been applied during the computation of the closure set $\text{C1}^{\mathbf{R}_c}(\sigma'_b)$ (resp. $\text{C1}^{\mathbf{R}_s}(\sigma_n)$).

Even closer to human societies, we could imagine a scheduling strategy $(\alpha^*\gamma^*\zeta^*)^*$. The corresponding transition rules are illustrated in Figure 4.14.

At this point, we make a few observations. First, as it has been pointed out in the case of the transition rule (ACS) , regimentation is modelled in both transitions (AC) and (A) by means of checking for the emptiness of the set of applicable regimentation rules. Second, $((\alpha\gamma)^*\zeta)^*$ cannot be subsumed by $(\alpha^*\gamma^*\zeta^*)^*$. This is because in the latter case a scenario like taking without paying a product from a supermarket with no camera supervision and bringing it back is possible, while in the first case it is not. Third, when the systems run with respect to (ACS) we know, by definition, that faults are being handled. This is no longer the case when considering the scheduling policies for liberal infinite behaviours. In order to guarantee such requirements (faults are being handled) we need an additional fairness constraint. In our case, fairness means that an active (enabled) normative rule is eventually applied, or equivalently, the transitions (C) , (S) are eventually taken:

$$\text{fairness} = \bigwedge_{l \in \mathbf{R}} (\Diamond \Box \text{enabled}(l) \rightarrow \Box \Diamond \text{taken}(l))$$

$$\boxed{
\begin{array}{c}
\frac{ms_i \xrightarrow{\alpha} ms'_i \quad \alpha = (\psi, \xi) \quad \theta \in \text{Sols}(\sigma_b, \psi) \quad \sigma'_b := \sigma_b \uplus \xi \theta \quad \text{App1}^{\mathbf{R}_r}(\sigma'_b) = \emptyset}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{\alpha} \langle \mathcal{M}', \sigma'_b, \sigma_n \rangle} \quad (A) \\
\\
\frac{N := \text{cl}^{\mathbf{R}_c}(\sigma_b) \setminus \sigma_b}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{\gamma} \langle \mathcal{M}, \sigma_b, \sigma_n \cup N \rangle} \quad (C) \\
\\
\frac{S := \text{cl}^{\mathbf{R}_s}(\sigma_n) \setminus \sigma_n \quad \sigma_b \cup S \not\models \perp}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{\xi} \langle \mathcal{M}, \sigma_b \cup S, \emptyset \rangle} \quad (S)
\end{array}
}$$

Figure 4.14: Transition Rules for $(\alpha^* \gamma^* \zeta^*)^*$ -semantics

where \mathbf{R} is the set of normative rules. The predicates *enabled* and *taken* are defined on normative multi-agent system configurations as:

$$\begin{aligned}
\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \models \text{enabled}(l) & \quad \text{iff} \quad l \in \text{App1}^{\mathbf{R}}(\sigma) \\
\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \models \text{taken}(l) & \quad \text{iff} \quad \langle \mathcal{M}, \sigma_b, \sigma_n \rangle \xrightarrow{ls} \langle \mathcal{M}, \sigma'_b, \sigma'_n \rangle \wedge l \in ls \wedge ls = \text{App1}^{\mathbf{R}}(\sigma)
\end{aligned}$$

where $\sigma = \sigma'_b = \sigma_b$ and ls is the set of applicable counts-as when l is a counts-as rule (a (C) transition has been applied), resp. $\sigma = \sigma'_n = \sigma_n$ and ls is the set of applicable sanctions when l is a sanction (a (S) transition has been applied).

One might wonder why not, instead of having four possible operational semantics, defining a most general one (the latter, in our case, corresponding to the strategy $(\alpha^* \gamma^* \zeta^*)^*$) and only mention the other three $((\alpha\gamma)^* \zeta^*)^*$, $(\alpha\gamma\zeta)^*$, $(\alpha(\gamma\zeta)^*)^*$ as more restrictive, particular cases. This is because we want to implement the strategies directly into the semantics. Transition rules by themselves say nothing about the order in which they should be executed. When more of them are active, one is chosen among them in a non deterministic way. Indeed, when it comes to building an interpreter for a given language a decision needs to be taken with respect to the choice of the scheduling algorithm (for example a Round-robin one) which would implement a given strategy (like $(\alpha\gamma\zeta)^*$). However, we avoid such choices by incorporating the strategies in the semantics. Being more accurate and precise when defining the semantics has the advantage of avoiding possible future errors in the implementations.

4.4.5 A Short Note on Computing Closures

We have mentioned in Section 4.4.2 that the computation of closures does not always terminate. This is the case when computing closures under “malformed” counts-as rules and the main reason lies in the fact that the antecedents of counts-as are defined on both brute and normative facts. Thus it can be that the process consisting of applying a counts-as resulting in a new normative fact which enables the application of a new counts-as can repeat ad infinitum.

We take, as an abstract example, $\sigma_b = \{p(x)\}$ and $R_c = \{p(x) \Rightarrow q(x), q(x) \Rightarrow q(q(x))\}$. It is then the case that $\text{C1}_i^{R_c}(\sigma_b) = \{p(x), q^i(x) \mid i \in \mathbb{N}\}$, thus no m exists such that $\text{C1}_m^{R_c} = \text{C1}_{m-1}^{R_c}$. Since we work with sets, one immediate solution is to restrict facts to terms with depth 1, that is, terms which contain only one functional symbol. However, if one finds such a restriction as being too severe, some “healthiness” conditions can be imposed. Namely, we require that a counts-as $c = (\text{cond}_c \Rightarrow \text{cons}_c)$ is *well-defined* in the sense that (1) there is at least one brute fact in cond_c and (2) $\text{Vars}(\text{cond}_c) = \text{Vars}(\text{cons}_c)$, where Vars denotes the set of variables from a formula. Since we consider that σ_b is finite, the conditions (1) and (2) are enough to guarantee that the computation of the closure always terminates. We take, as an illustration, $\sigma_b = \{p(x), f(x)\}$ and $R_c = \{p(x) \Rightarrow q(x), f(x) \wedge q(x) \Rightarrow q(q(x))\}$, where, for convenience, “ \wedge ” denotes “,” which we interpret as conjunction. It is then the case that $\text{C1}_2^{R_c}(\sigma_b) = \text{C1}_1^{R_c}(\sigma_b) = \{p(x), f(x), q(x), q(q(x))\}$, since due to (1) and (2) the only possible substitution for $p(x) \wedge f(x) \wedge q(x) \wedge q(q(x)) \models f(x) \wedge q(x)$ is $[x/x]$, thus no new elements can be added to the closure.

Heaving healthiness conditions for counts-as rules is, however, not sufficient when it comes to computing the limit of the sequence σ_b^* as introduced in Section 4.4.4. Following the same line of reasoning, it is now the case that the process of applying a sanction results in adding a new brute fact which enables the application of a counts-as rule can be iterated “ad infinitum”.

We reconsider the previous example. If we now take R_s as being $\{s = (q(x) \Rightarrow f(x))\}$ the computation of σ_b^* can never reach its limit since at each step the application of s feeds the set of brute facts with a new $f^i(x)$ which makes it possible to apply the counts-as rule c_2 with the substitution $[x/f^i(x)]$. This is what we call a *productive* rule. A solution for avoiding productiveness is to impose a syntactic condition on sanctions. Namely, we require that for any counts-as rule $c = (\text{cond}_c \Rightarrow \text{cons}_c)$, sanction $s = (\text{cond}_s \Rightarrow \text{cons}_s)$, a fact name f , and ground terms t, t' such that $f(t)$ is in cond_c and $f(t')$ is in cond_s we have that the length of t' is smaller than t .

where t, t' are arbitrary terms. That is, if there exists a brute fact $f(t')$ in the consequence of s (thus heaving the same head as a brute fact from the antecedent of c) then t' is shorter in length than t . This guarantees that no new substitution is generated and this implies that the computation terminates.

4.5 Timed Normative Artifacts

In this section we introduce in a modular way a time extension of the normative language presented in the previous section. A timed normative multi-agent system is a collection of timed agents where the behaviour in time of the individual agents is monitored and normative rules are applied consequently. The choice of agent language is not relevant. However, for the sake of completeness, we consider timed agent languages like the timed extensions of BUnity and BUPL as we have described them in Section 4.3. In this way we can describe in a uniform manner a timed, agent-based framework. We recall that we designed timed agents as agents equipped with clocks and that these clocks can be seen as stop-watches which can be started and checked independently of one another, however they use the same unit to measure

the passing of time. At each moment the clocks' values of any agent can be checked by an external observer. The observer cannot, however, change the agents' clocks values since it is only the agents that manipulate their own clocks by delaying and resetting, actions which are invisible to the environment. The advantage of agents having their own clocks is that the normative system does not need to have a clock on its own. In order to (dis)allow the execution of actions at given instances of time or to punish delays it is sufficient¹⁰ to consult the clocks of the agents.

Timed normative rules extend normative rules by allowing clock constraints in their preconditions. We motivate this design choice by noting that in a timed framework new violations and sanctions can arise due to time delays. For example, not paying a fine in a given amount of time might entail the application of a new violation. It is also the case that a sanction might be cancelled when the expiration time has passed.

The semantics of timed normative multi-agent systems extends the untimed one from Section 4.4.4 as follows. A timed normative multi-agent system state $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$ differs from an untimed state in only one aspect, namely, the recorded states of the constituting agents are timed: $\mathcal{M} = \{(ms_i, v_i) \mid i \in \mathcal{I}\}$ with v denoting clock interpretations, i.e., v_i represents the current clock values of ms_i . The timed version of the transition rules for the untimed normative language change as shown in Figure 4.15.

$$\begin{array}{c}
 \frac{(\phi_c, a = (\psi, \xi)) \quad (ms_i, v_i) \xrightarrow{a} (ms'_i, v'_i) \quad \theta \in Sols(\sigma_b, \phi) \quad v_i \models \phi_c}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', \sigma_b \uplus \xi \theta, \sigma_n \rangle} \quad (t-A) \\
 \\
 \frac{(\phi_c, \Phi \Rightarrow \Psi) \in \mathbf{R}_c \quad \theta \in Sols(\sigma_b \cup \sigma_n, \Phi) \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}, \sigma_b, \sigma_n \uplus \Psi \theta \rangle} \quad (t-C) \\
 \\
 \frac{(\phi_c, \Phi \Rightarrow \Psi) \in \mathbf{R}_s \quad \theta \in Sols(\sigma_b \cup \sigma_n, \Phi) \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}, \sigma_b \uplus \Psi \theta, \sigma_n \rangle} \quad (t-S) \\
 \\
 \frac{(\phi_c, \Phi \Rightarrow viol_{\perp}) \in \mathbf{R}_r \quad Sols(\sigma_b \cup \sigma_n, \Phi) \neq \emptyset \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_b, \sigma_n \rangle \rightarrow \nabla} \quad (t-R)
 \end{array}$$

Figure 4.15: The Transition Rules for Timed NMA

With respect to the previous untimed rule (A) from Figure 4.14, in the transition rule (t-A), the double arrow \xrightarrow{a} denotes the transitive closure not only of τ steps, \rightarrow , for internal actions, but also δ steps, $\xrightarrow{\delta}$, for delay actions. The rule (t-A) has also an additional premise

¹⁰Please note that by definition clocks cannot “break” or have “fake time units”.

saying that agent i is enabled to perform a when the clock constraint ϕ_c is satisfied by the clock valuation v_i . In the case that the transition is possible, the system changes such that it reflects the possibly new clock value v'_i which might have changed while δ steps. As for the meaning of the timed transition rules for the application of norms we note that it only differs from the meaning of the untimed ones in that an additional check if the clock constraint ϕ_c is satisfied by the current clock valuation v^M . The transition (t - R) is new. It expresses that whenever a regimentation rule is applicable the system goes into a deadlock state which is represented by ∇ . This rule is not meant to be executable but only to provide information. We use the information when effectively implementing the different normative semantics from Section 4.4.4 by means of strategies. As we will explain in more detail in Section 4.6.2, strategies use (t - R) only to test that this rule is *not* applicable before allowing the execution of any action. This is in order to forbid action execution in the case where a regimentation rule were applicable.

We take as an illustration a timed variant of the train scenario described in Figure 4.10. With respect to that example, the changes are as follows. First, we see that the two agents

```

Agents:
  psg1 clock1 passenger_prog1  1
  psg2 clock2 passenger_prog2  1
Facts:
Effects:
  enter(X) = ({not at_platform(X)}, {at_platform(X)})
  (clock(X) < 10, buy-ticket(X) = ({not ticket(X)}, {ticket(X)}))
  embark(X) = ({at_platform(X), not in_train(X)},
               {not at_platform(X), in_train(X)})

Counts-As rules:
  at_platform(X) /\ not ticket(X) =>
                                viol_ticket(X)
  ( fined(X, Y) /\ not paid-fine(X),
    clock(X) > 100 ) => viol_fine(X, Y)
Regimentation rules:
  in_train(X) /\ not ticket(X) => viol_|_
Sanction rules:
  viol_ticket(X) => fined(X, 25)
  viol_fine(X, Y) => fined(X, 2*Y)

```

Figure 4.16: A Timed NMAS Program

psg1 and psg2 are declared as having one clock each: `clock1` and `clock2`. The specification of the action `buy-ticket` changes such that buying a ticket is allowed only if this is done at most until `clock(X)`¹¹ shows 10 units of time. The last change regards norms: the clock constraint in the second counts-as rule says that fines which are not paid within 100 units of time entail new violations. In order to analyse this scenario, we assume a specific behaviour for the agents `psg1` and `psg2`, and further, that `psg1` can be dishonest. By

¹¹For our scenario, `clock(X)` can be seen as a “library function” returning the valuation of the clock of agent X .

“specific” we mean that we think of `psg1` and `psg2` as BUpL agents and their plans are illustrated in Figure 4.17.

```

p1  =  ( ((xc < 9), buy-ticket) +
        ((xc >= 9), enter) );
        embark; (true -> (xc < 200)); p'

p2  =  ((yc < 8), buy-ticket, yc := 0);
        enter; (true -> yc < 10); embark

```

Figure 4.17: The plans of `psg1` and `psg2`

The plan `p1` says that `psg1` intends to buy a ticket if its clock `xc` shows less than 9 units, otherwise it will enter without a ticket. It further says that if `psg1` manages to embark, it spends at most 200 units in the train. Depending on the scheduling of normative artifacts, we obtain different timed traces. One such trace is $(10, \text{enter}), (10, \text{embark}), (50, \gamma_1, \zeta_1), (80, \text{pay-fine})$ ¹² illustrating that `psg1` entered the train at time 10, that after 40 units of time the counts-as rule and the corresponding sanction `fined(psg1, 25)` have been applied, and finally that, at time 80, `psg1` paid the fine. Another possible trace is: $(10, \text{in-train}), (50, \gamma_1, \zeta_1), (110, \gamma_2, \zeta_2)$ illustrating that `psg1` did not pay the fine in time and consequently the new norms have been applied resulting in doubling the fine. We discuss in more detail possible executions under different scheduling of the normative artifacts in Section 4.6.

On the other hand, we assume that `psg2` is correct and its plan, `p2`, is to always buy a ticket before entering the platform. Furthermore, it has up to 8 units of time to decide what ticket to buy and it resets the clock after the action is done. The delay `true -> yc < 10` means that `psg2` waits at most 10 units of time before embarking the train. Being that `p2` is deterministic, all possible timed traces of `psg2` differ only in the timings. One example is: $(7, \text{buy-ticket}), (0, \text{enter}), (4, \text{embark})$ illustrating that at time 7 `psg2` bought a ticket, entered the platform at time 0 (after resetting its clock), and after 4 units `psg2` in the train.

4.6 Executable Timed Choreographed Normative Systems

In this chapter we combine under the same multi-agent framework both low and high level timed coordination artifacts. The integration reduces to extending the timed normative language from Section 4.5 by adding the constructions specific to timed choreographies. This new extension makes it possible to design *timed choreographed normative multi-agent systems* (TCNMAS). We describe how we can execute such systems by means of a rewriting logic extension called *real-time rewriting logic* [ÖM02]. We show that the nondeterminism in the application of norms can be still an issue in the new setup and how the same notion

¹²For illustration purposes, we assume that in this particular case the action `pay-fine` is in plan `p'`.

of scheduling policies presented in Section 4.4.4 remains a valid solution. Furthermore, we propose the use of strategies to effectively implement scheduling policies. In this way, there is no dependency between the normative artifact corresponding to a given scheduling policy and the normative language. Consequently, one clear benefit is that changing to a different normative artifact must not be reflected in the semantics of the normative language itself. Moreover, the separation between execution and control makes it simpler to reason in a modular way about normative multi-agent systems. This is useful for example when “debugging”: to find an error in the system one can first verify the normative artifact. Only if this process is unsuccessful further effort is needed to debug one by one the constituting agent programs. We conclude the section with a short discussion of how we can further use strategies at a more theoretical level. If, at a practical level we can use strategies to analyse the execution of normative systems in the context of different normative artifacts, at a more theoretical level we can study possible connections among normative artifacts themselves and how they relate to low level coordination mechanisms like choreographies.

4.6.1 Combining Timed Choreographies and Norms

We recall that the states of the timed normative systems were defined as $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$. In the context of timed choreographies, this is no longer enough since they should reflect also the choreographic states. We denote the new configurations by the notation $\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle$, where the symbol σ_{ch} denotes the triples (cs, v, θ_{ch}) which were introduced in Section 4.3. The semantics of the timed normative language running under the directions of a timed choreography is given by the rules in Figure 4.18. They are a combination of the rules from Figure 4.8 and Figure 4.15. More precisely, the rules for the application of norms reflect the incorporation of choreographic states σ_{ch} , and the rule $(t\text{-sync-act})$ (replacing $(t\text{-A})$) as well as (delay) reflect the incorporation of both brute and normative sets σ_b, σ_n .

4.6.2 Execution by Real-Time Rewriting

In this section we describe the mapping of timed choreographed normative multi-agent systems as *real-time rewrite specifications* [ÖM07] such that we can execute TCNMAS. The execution of the untimed choreographed normative systems follows directly from the mapping of the untimed transitions from Figure 4.18 as rewrite rules. This mapping does not pose difficulties and it follows the same lines described in Section 2.6.

Real-time rewrite specifications are rewrite specifications where some rules, called *tick rules*, model the passing of time, leaving the “ordinary” rewrite rules are instantaneous. More precisely, a real-time rewrite specification T^t is a tuple (T, H, δ^r) where T is a rewrite theory (Ω, E, R) such that:

- H is a morphism $H : \text{TIME} \rightarrow (\Omega, E)$ which interprets the abstract equational theory TIME in the underlying equational theory of T . TIME is a commutative monoid with an order $\text{TIME} = (\text{Time}, 0, +, <)$ and additionally with $\dot{-}$ (standing for $x - y$ when $x > y$ or 0 otherwise) and \leq . The morphism H is a nice solution which avoids fixing a *time domain* for all real-time rewrite specifications. Depending on the application,

$$\begin{array}{c}
\frac{\langle cs, v \rangle \xrightarrow{\delta} \langle cs, v + \delta \rangle \quad \bigwedge_i ((ms_i, v_i) \xrightarrow{\delta} (ms_i, v_i + \delta))}{\langle \mathcal{M}, (cs, v, \theta_{ch}), \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', (cs, v + \delta, \theta_{ch}), \sigma_b, \sigma_n \rangle} \text{ (delay)} \\
\frac{\langle cs, v \rangle \xrightarrow{\|\mathcal{J}(j, x_{aj})} \langle cs', v' \rangle \quad \theta'_{ch} \in Sols(\bigwedge_j (E(ms_j), x_{aj} \theta_{ch}))}{a_j := x_{aj} \theta'_{ch} \quad \bigwedge_j (\phi_{c_j}, a_j = (\psi_j, \xi_j)) \quad \theta \in Sols(\sigma_b, \bigwedge_j \psi_j)} \\
\frac{\bigwedge_j ((ms_j, v_j) \xrightarrow{a_j} (ms'_j, v'_j)) \quad v^{\mathcal{M}} \cup v \models \bigwedge_j \phi_{c_j} \quad v^{\mathcal{M}'} \cup v' \models I(cs')}{\langle \mathcal{M}, (cs, v, \theta_{ch}), \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', (cs', v', \theta_{ch} \theta'_{ch}), \sigma_b \uplus \xi', \sigma_n \rangle} \text{ (t-sync-act)} \\
\frac{(\phi_c, \Phi \Rightarrow viol_{\perp}) \in \mathbf{R}_r \quad Sols(\sigma_b \cup \sigma_n, \Phi) \neq \emptyset \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle \rightarrow \nabla} \text{ (t-R)} \\
\frac{(\phi_c, \Phi \Rightarrow \Psi) \in \mathbf{R}_c \quad \theta \in Sols(\sigma_b \cup \sigma_n, \Phi) \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \uplus \Psi \theta \rangle} \text{ (t-C)} \\
\frac{(\phi_c, \Phi \Rightarrow \Psi) \in \mathbf{R}_s \quad \theta \in Sols(\sigma_b \cup \sigma_n, \Phi) \quad v^{\mathcal{M}} \models \phi_c}{\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}, \sigma_{ch}, \sigma_b \uplus \Psi \theta, \sigma_n \rangle} \text{ (t-S)}
\end{array}$$

Figure 4.18: Transition Rules for Timed Normative MAS and Time Choreographies

the type of domain can be, for example, discrete or dense. When the time domain is discrete, H should interpret the sort *Time* as \mathbb{N} , and the corresponding operators as the usual arithmetic operators. When the time domain is dense, H can interpret *Time* as \mathbb{Q}^+ .

- Ω contains two predefined sorts *System* and *GlobalSystem*, on which only one operator is defined $\{_ \} : \text{System} \rightarrow \text{GlobalSystem}$. This is in order to make it possible to distinguish *instantaneous* from *tick* rules. Namely, to those particular rewrite rules from R which are on terms of sort *GlobalSystem*, i.e., $l : \{t\} \rightarrow \{t'\}$, a duration $\delta^r(l)$ is associated by means of the mapping δ^r . This is in the idea that the rule l is a tick rule and it can be executed in at most δ_l^r time. In this case, the following notation is adopted $T^t \vdash \{t\} \xrightarrow{\delta^r(l)} \{t'\}$. All other rules are instantaneous, i.e., their duration is 0. Their notation is simply $T^t \vdash t \rightarrow t'$.

It is shown in [ÖM02] (and further simplified in [ÖM07]) how real-time rewrite theories map to rewrite theories. The idea is to associate to terms of sort *GlobalSystem* $\{t\}$ with “clocked terms” ($\{t\}$ in time δ). The sort *GlobalSystem* becomes a subsort of *ClockedSystem*, the sort for clocked terms. It is then the case that it can be proved that for any computation path in T^t there is a corresponding path in T^C and vice-versa, as stated in Theorem 4.6.1.

4.6.1. THEOREM ([ÖM07]). *For all terms $\{t\}$, $\{t'\}$ of sort *GlobalSystem* and for any time variables denoted by the terms δ , $\delta^r(l)$ of sort TIME^H , the following equivalences hold:*

$$\begin{aligned} T^t \vdash \{t\} \xrightarrow{\delta^r(l)} \{t'\} &\Leftrightarrow T^C \vdash \{t\} \rightarrow (\{t'\} \text{ in time } \delta^r(l)) & (1) \\ &\Leftrightarrow T^C \vdash (\{t\} \text{ in time } \delta) \rightarrow (\{t'\} \text{ in time } \delta +^H \delta^r(l)) \\ T^t \vdash t \rightarrow t' &\Leftrightarrow T^C \vdash (\{t\} \text{ in time } \delta) \rightarrow (\{t'\} \text{ in time } \delta). \end{aligned}$$

4.6.2. REMARK. Thanks to Theorem 4.6.1, recalling the connection between rewrite theories and Kripke structures, it can be stated that model-checking can be used also for real-time rewrite specifications. This has the convenient implication that the same technique we have used in Section 3.1 for verifying agent refinement works also for timed-agent refinement. ♣

The time variable δ in Theorem 4.6.1 denotes the duration of a tick step, and it is meant to compute the successor time value. In the case of a discrete time domain like \mathbb{N} , δ is by default 1, the unit element of $(\mathbb{N}, 0)$. That is, at each step the time is incremented by 1. In the dense case, δ corresponds to a chosen *time sampling strategy*. For example, δ could be set to 2, meaning that time advances at each step by 2 units. Fixing a time sampling strategy has the consequence that the completeness of (1) is lost. More precisely, (1) holds in only one direction as Proposition 4.6.3.

4.6.3. PROPOSITION ([ÖM07]). *Given a real-time rewrite theory T^t and a time sampling strategy s with the corresponding rewrite theory T^s , the following implication holds for all terms of sort *GlobalSystem* and for all ground terms δ :*

$$T^s \vdash t \xrightarrow{\delta} t' \Rightarrow T^t \vdash t \xrightarrow{\delta} t'$$

4.6.4. REMARK. One needs to be cautious in what concerns model-checking dense time domains. Due to Proposition 4.6.3, the fact that no counterexample is found when model-checking a property for T^s does not imply that there is no counterexample in the original real-time rewrite theory T^l . Instead, when a counterexample is found in T^s , then this is also a counter-example in T^l . ♣

By fixing a time sampling strategy the rewrite rule $\{t\} \rightarrow (\{t'\} \text{ in time } \delta^r(I))$ becomes executable because the strategies assign a particular value to the uninitialised duration variable δ^r . This result we use to effectively execute timed choreographed normative multi-agent systems. To show this, we first describe how Timed BUpL agents are prototyped as real-time rewrite theories. We recall that Timed BUpL configurations are pairs of mental states and valuations of the agent's clocks, i.e., (ms, v) (Section 4.3.2) and that mental states are mapped to terms of sort $BpState$ from the rewrite theory T_{BUpL} (Section 2.6). The natural mapping of (ms, v) to terms in a real-time rewrite theory T_{BUpL}^l consists in defining a new sort $TBpState$ with a constructor (ms, v) taking as parameters a term of sort $BpState$ and a term of sort $Set\{Times\}$. $TBpState$ is defined as being a subsort of $System$ such that its terms can appear in tick rules. More precisely, in T_{BUpL}^l , there is only one tick rule corresponding to the transition $(delay)$ in Figure 4.7:

$$\begin{aligned} delay \quad : \quad & \{(\mathcal{B}, (\phi \rightarrow I); p', v)\} \rightarrow \{(\mathcal{B}, p', v')\} \text{ in time } \delta \\ & \text{if } \mathcal{B} \models \phi \wedge v' := inc(v, \delta) \wedge eval(v', I) = true \end{aligned}$$

where inc is a function which increments the current valuations of the clocks with the duration δ and $eval$ is a function which checks if the invariant I remains true with respect to the new valuations. All the other Timed BUpL rules are instantaneous. We only present the rewrite rule corresponding to $(t-act)$ in Figure 4.7:

$$\begin{aligned} t-act \quad : \quad & (\mathcal{B}, p, v) \rightarrow (update(\mathcal{B}, \xi \theta), p' \theta, reset(v, \lambda)) \\ & \text{if } p = (\phi_c, a, \lambda); p' \wedge a = (\psi, \xi) \wedge \theta = match(\mathcal{B}, \psi) \wedge eval(v, \phi_c) = true \end{aligned}$$

where the function $reset$ works as expected, it resets the clocks from λ .

The next issue to look at before moving at the agent system level is how to prototype time choreographies as real-time theories. This is done in a similar¹³ manner as above. Briefly, the prototyping consists of two steps: (1) declare the sort of timed choreography states as a subsort of $System$; (2) mapping the transition rules from Figure 4.4 to a tick and resp., an instantaneous rule.

Having the prototypes for Timed BUpL and timed choreographies we describe the prototyping of the rules in Figure 4.18. We first declare the sort $TCNState$ of the terms corresponding states of the timed choreographed normative system $\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle$ as being a subsort of $System$. The prototyping of the normative rules $t-R$, $t-C$, and $t-S$ is easy: they are mapped to instantaneous rules. For a more suggestive reading, we make a convention to label these instantaneous rules by *reg*, *counts-as*, and *sanction*. The only difficulty might be in prototyping the synchronisation of actions $\bigwedge_i ((ms_i, v_i) \xrightarrow{a_i} (ms'_i, v'_i))$ in the rule $t-sync-act$. Since the number of agents participating in a synchronisation step is not fixed a priori, but depends on

¹³The reader can also refer to [ÖM02] for the original specification of timed automata as real-time rewrite theories.

the concrete definition of the choreography used at “runtime”, to have a generic rewrite rule means that we need a continuation function which takes as parameter the choreographic label and recursively applies itself until no longer possible. At each application, one agent configuration is rewritten with respect to the choreographic label. To implement this means that we need to force a particular action to take place. We recall that rewriting has big-step semantics thus one solution could be to use a combination of frozen declarations and intermediary configurations as we did in Section 3.1. Another solution is to use meta-functions to control the execution at a meta-level. We present this approach since it is more elegant and it gives the opportunity to illustrate another use of strategies:

$$\begin{aligned}
 t\text{-sync-act} & : \langle \mathcal{M}, (cs, v, \theta_{ch}), \sigma_b, \sigma_n \rangle \rightarrow \\
 & \quad \langle \mathcal{M}', (cs', v', \theta_{ch} \theta'_{ch}), \text{update}(\sigma_b, \text{getPost}(l)\theta), \sigma_n \rangle \\
 (*) & \quad \text{if } [l] \langle cs', v' \rangle \in \text{ch-rule}_2 @ \langle cs, v \rangle \wedge l\theta'_{ch} := \text{matchNames}(l, \theta_{ch}) \wedge \\
 (**) & \quad \mathcal{M}' \in (\text{matchRew } M \text{ by } s(l\theta'_{ch}) @ \mathcal{M}) \wedge \theta \in \text{Sols}(\sigma_b, \text{getPre}(l\theta'_{ch})) \\
 & \quad \wedge v^{\mathcal{M}} \cup v \models \text{getTC}(l\theta'_{ch}) \wedge v^{\mathcal{M}'} \cup v' \models I(cs')
 \end{aligned}$$

To simplify, we assume that the instantaneous rewrite rule implementing the second transition from Figure 8.4 incorporates the corresponding choreographic label as a list in the next state, i.e., $\langle cs, v \rangle \xrightarrow{l} \langle cs', v' \rangle$ where, for convenience, l is a list such as $(i_1, x_{a_1}) \dots (i_k, x_{a_k})$. We briefly explain $t\text{-sync-act}$. First the list l is grounded by replacing all action variables with actions which the agents are enabled to execute. This is what line (*) says. The ground list $l\theta'_{ch}$ has the form $(i_1, a_1) \dots (i_k, a_k)$ and it is used in all *get* functions to recursively collect the relative information. More precisely, *getPre* (resp., *getPost*) returns all pre- (resp., post-) conditions and *getTC* returns all clock constraints from the timed specifications of the actions a_1, \dots, a_k . Line (**) illustrates the use of the strategy *matchRew* from Section 2.6.2 for rewriting the subterms representing the agents i_1, \dots, i_k . The symbol M in the expression of the strategy denotes a variable of the sort for multi-agent states. The function s transforms the ground list $l\theta'_{ch}$ into a list of strategy definitions:

$$\begin{aligned}
 s((i_1, a_1)) & = (ms_{i_1} \text{ by using } (\text{test}(t\text{-act}) ? t\text{-act}[a \leftarrow a_1]; \text{fail} : \text{delay})!) & (1) \\
 s((l_1 l)) & = s(l_1), s(l) & (2)
 \end{aligned}$$

where in (2) “,” is used as a separator for lists of strategy definitions. Line (1) represents the base case, it builds the strategy corresponding to a basic choreographic pair: first it tests if the rule $t\text{-act}$ is enabled; if this is the case then it forces *fail* to break the “repeat until the end” strategy; if not, then it delays for some time and then the whole cycle repeats. Line (2) represents the inductive case: the result of s consists of applying s to the first element l_1 concatenated with the result of applying s to the tail of the list l .

Though $t\text{-sync-act}$ is not explicitly a tick rule itself, time advances because of the delays in the agent programs. The only “true” tick rule is the one corresponding to the rule *delay* in Figure 4.18:

$$\begin{aligned}
 \text{delay} & : \langle \mathcal{M}, (cs, v, \theta_{ch}), \sigma_b, \sigma_n \rangle \rightarrow \langle \mathcal{M}', (cs', \text{inc}(v, \delta), \theta_{ch}), \sigma_b, \sigma_n \rangle \\
 & \quad \text{if } (\langle cs, v \rangle \rightarrow \langle cs', v' \rangle \text{ in time } \delta) \wedge (\mathcal{M} \rightarrow \mathcal{M}') \wedge (v^{\mathcal{M}} \cup \text{inc}(v, \delta) \models I(cs))
 \end{aligned}$$

where by $(\mathcal{M} \rightarrow \mathcal{M}')$ we mean the following rewrite rule:

$$\{(ms, v)\} \cup \mathcal{M} \rightarrow \{(ms', inc(v, \delta))\} \cup \mathcal{M}'$$

if $((ms, v) \rightarrow (ms', inc(v, \delta)) \text{ in time } \delta) \wedge (\mathcal{M} \rightarrow \mathcal{M}')$

which increments all agents' clocks in the system, that is, implements the synchronisation of delays $\bigwedge_i ((ms_i, v_i) \xrightarrow{\delta} (ms_i, v_i + \delta))$.

Having the timed choreographed normative language encoded as a real-time rewrite theory we can execute timed choreographed normative multi-agent systems by real-time rewriting. As an illustration, we reconsider the train scenario described in Section 4.5. There, the “actors” are two agents `psg1` and `psg2`. The agent `psg1` intends to behave dishonestly instead of missing the train. The agent can be “forced” to act correctly by means of a timed choreography like:

$$\begin{aligned} \text{chStrict} = & \text{ (zc} < 4 \text{); (psg1, (zc} \geq 3 \text{, buy-ticket));} \\ & \text{(zc} < 7 \text{); (psg2, (zc} \geq 6 \text{, buy-ticket)).} \end{aligned}$$

In this particular case, one possible resulting computation of the system from Figure 4.16 running under the timed choreography is as follows:

$$\begin{aligned} & ((\text{psg1, buy-ticket}), \text{xc} = \text{yc} = \text{zc} = 3), \\ & ((\text{psg2, buy-ticket}), \text{xc} = \text{zc} = 6, \text{yc} = 0) \end{aligned}$$

illustrating that there was a delay of 3 units before each agent bought a ticket and that `psg2` reset clock `yc` because this action was built in the plan `p2`. The computation is unique “modulo timings”, i.e., with respect to the action names appearing in it. This would not have been the case if, for instance, in the choreographic label $(\text{psg1, (zc} \geq 3 \text{, buy-ticket)})$ the action name were substituted by an action variable. Such a situation might have triggered the application of normative rules and so far as the semantics is concerned with, this process is nondeterministic. For the ease of reference, we denote this more permissible choreography as `chLax`. The issues of nondeterminism and possible scheduling policies of norms were discussed in Section 4.4.4. In this section we describe how to effectively implement scheduling policies by means of strategies. We begin with the strategy corresponding to the scheduling policy $((\alpha\gamma)^*\zeta)^*$ which we discussed in Section 4.4.4:

$$\begin{aligned} \text{vigilant} = & \text{ (test(t-sync-act ; reg) ? fail :} \\ & \text{(t-sync-act ; counts-as ! ; sanction !)) !} \end{aligned}$$

The strategy *vigilant* says that actions are executed only if they do not enable the application of regimentation rules (in which case the strategy fails). After executing an action, counts-as rules are applied until no longer possible. Finally, all corresponding sanctions are applied. This process is iterated until no action can be executed. In our train scenario, if we assume a less restrictive choreography extending `chLax`, then the result of applying this strategy reflects that both agents are in the train with tickets and that previously, `psg1` has been sanctioned. If this were not the case, then the system is in a deadlock state because `psg1` embarks without a ticket, thus enabling the application of the regimentation rule.

A simple change in *vigilant* consisting in substituting “;” by “|” in *counts-as ! ; sanction !* leads to a less restrictive normative system. An illustrative scenario is that of a video camera

monitoring in a supermarket, or of a radar measuring the velocity of the passing vehicles. In such cases, sanctions do not necessarily follow immediately after recording an infraction.

A more restricted society conforms to the scheduling policy $(\alpha(\gamma\zeta)^*)^*$ and is implemented by means of:

$$\begin{aligned} \text{totalitarian} = & \quad (\text{test}(t\text{-sync-act} ; \text{reg}) ? \text{fail} : \\ & (t\text{-sync-act} ; (\text{counts-as} ! ; \text{sanction} !) !) ! \end{aligned}$$

saying that the process of applying counts-as rules followed by sanctions is iterated until it is no longer possible. This characterises scenarios where the application of a sanction enables the application of a new counts-as. We take, for instance, a traffic scenario where an actor drives through the red light, thus violating the traffic law. Consequently, a fine is applied. We assume that this is done automatically by withdrawing a certain amount of money from the actor's account. It is then the case that not enough money in the account results in a new violation. This is under the supposition that the bank has a regulation specifying that the client must not go below a certain debt level, otherwise the client is added to the bank's black list and has to pay an additional fee. We note that this latter sanction rule can never be applied when the system runs with respect to the strategy *vigilant*. However, in the case of the train scenario the result of applying either one of the strategies *vigilant* or *totalitarian* is the same.

A liberal society as described by the scheduling policy $(\alpha^*\gamma^*\zeta^*)^*$ is implemented using the strategy:

$$\begin{aligned} \text{liberal} = & \quad (\text{test}(t\text{-sync-act} ; \text{reg}) ? \text{fail} : t\text{-sync-act})^* ; \\ & (\text{try}(\text{counts-as}) ? \text{try}(\text{sanction}) : \text{idle})^* \end{aligned}$$

This strategy imposes no restrictions on *when* normative rules are applied. One possible result could be that the agent `psg1` was sanctioned because of being at the platform without a ticket. Another solution in a “liberal” agent society could be also the case that `psg1` is at the platform without a ticket and without being fined. Such a scenario would never be possible when using either one of the strategies *vigilant* or *totalitarian*. This observation leads to formulating Proposition 4.6.5 stating that the property *enforcement*, which we defined in Section 4.4.3, is guaranteed to hold in normative systems where the normative artifacts are implemented by either *vigilant* or *totalitarian* strategy. Before presenting the result, we introduce a more convenient definition of the semantics of the strategy language S . Under certain requirements like *monotonicity* and *persistence*, which the interested reader can find in [MOMV09], the set-theoretic semantics of a strategy language has an alternative definition as follows:

$$\{[s@t]\} = \{t' \in T_\Sigma(X) \mid (\exists w). S(T) \vdash s@t \rightarrow^* w \wedge t' \in \text{sols}(w)\}.$$

Equally stated, a computation like

$$s@t \rightarrow w_1 \rightarrow w_2 \rightarrow \dots w_n \rightarrow \dots$$

always exists such that $\{[s@t]\}$ is $\cup_{n \in \mathbb{N}} (\text{sols}(w_n))$. The strategy language that we use, i.e., the one described in Section 2.6.2 fulfils the above mentioned requirements, the proof is presented in [MOMV09]. For convenience, we use this alternative definition.

4.6.5. PROPOSITION. *Let t be the term corresponding to an initial normative multi-agent system state $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$. Given that the associated normative artifact is implemented by the strategy \mathcal{N} being either vigilant or totalitarian, the following statement holds: $\{[\mathcal{N}@t]\}^{14} \models (\forall c \in \mathbf{R}_c)(\forall s \in \mathbf{R}_s)\text{enforcement}(c, s)$.*

Proof. We recall that $\text{enforcement}(c, s)$ is defined as:

$$(\text{cond}_c \wedge (\text{cons}_c \rightarrow \text{cond}_s)) \rightarrow \Diamond \text{cons}_s.$$

We proceed by reduction to the absurd. We assume that there exists $k \geq 1$ such that $\mathcal{N}@t \rightarrow^* w_k$, that w_k can no longer be rewritten (1), that there exists a counts-as rule c and a sanction rule s such that $w_k \models (\text{cond}_c \wedge (\text{cons}_c \rightarrow \text{cond}_s))$ (2) and $w_k \not\models \text{cons}_s$ (3), and that k is the first one with these properties, i.e., for any $i \leq k$, $\text{sols}(w_i) \models (\forall c \in \mathbf{R}_c)(\forall s \in \mathbf{R}_s)\text{enforcement}(c, s)$ (4). We do a case analysis on the possible rewritings $w_{k-1} \rightarrow w_k$ and show that such a k cannot exist. The rewriting $w_{k-1} \rightarrow w_k$ can only be due to the application of (a) *act*, (b) *counts-as* or (c) *sanction*:

- (a) because of (2) + (4) cond_c is due to the action executed in the step from w_{k-1} to w_k . But then *counts-as* is applicable (and allowed by \mathcal{N} , thanks to the normalising strategy “!”) from w_k which contradicts (1).
- (b) there are two possibilities: if cond_c holds in w_{k-1} then one application of *counts-as* can at most add cons_c to the normative facts; otherwise, cond_c is the effect of the application of *counts-as*. In either case at least one *sanction* (and allowed by \mathcal{N} , thanks to the normalising strategy “!”) is applicable from w_k which contradicts (1).
- (c) because of (3), the application of *sanction* from w_{k-1} handled a different counts-as rule. But then another *sanction* is applicable (and allowed by \mathcal{N} , thanks to the normalising strategy “!”) from w_k which contradicts (1).

thus, by absurd, no w_k satisfying (1)-(4) exists. ■

Having a classification of types of normative systems we can systematically study the expressive power for each class separately. By expressive power of a normative artifact we mean the domain of possible resulting behaviours when the multi-agent system runs under the coordination of an artifact. One example of a study is represented by Proposition 4.6.5 which focuses on a liveness property, *enforcement*. In what follows we concentrate on *regimentation* as a safety property. We begin our analysis with the remark that, for instance in totalitarian societies, regimentation is modelled by definition. However, the way in which the normative artifacts ensure that *regimentation* holds is by forcing the system to a deadlock state whenever a regimentation rule is applicable. This is usually not a reasonable solution. Instead, we can use choreographies to avoid reaching a deadlock when this is possible, i.e., when there exist other execution paths with no deadlock states. Proposition 4.6.6 states precisely this: choreographies can be seen as a way to implement regimentation.

¹⁴By abuse of notation, we apply strategies implementing normative artifacts not only to choreographed normative systems but also to those without choreographies. To be formally correct, it is enough to make a simple replacement in the definition of strategies, namely, to replace *t-sync-act* by the usual *act* rule from the semantics of agent languages.

4.6.6. PROPOSITION. *Let t be the term corresponding to an initial normative multi-agent system state $\langle \mathcal{M}, \sigma_b, \sigma_n \rangle$ and let \mathcal{N} be the strategy implementing the normative artifact associated to the system. If there is a computation path $\mathcal{N}@t \rightarrow^* w$ such that $\text{sols}(w) \models (\forall r \in \mathbf{R}_r) \text{regimentation}(r)$, then there exists a choreography ch such that $\{[\mathcal{N}@t']\} \models (\forall r \in \mathbf{R}_r) \text{regimentation}(r)$, where t' is the term corresponding to the choreographed normative state $\langle \mathcal{M}, \sigma_{ch}, \sigma_b, \sigma_n \rangle$.*

Proof. We prove the existence of a choreography by effectively constructing it from the computation path $\mathcal{N}@t \rightarrow^* w$. Let w_1, \dots, w_k be the intermediary terms. The choreography ch is constructed by iterating the steps (1) - (3):

- (1) for any $w_i \rightarrow w_{i+1}$ by applying the rule $t\text{-act}$, where the formal parameter a is instantiated by a ground action a_i , on an agent i , concatenate the choreographic label (i, a_i) ;
- (2) for any delay action with δ units for a clock xc , concatenate the choreographic label $xc \leq \delta$;
- (3) for any other rule application skip.

Since we only consider finite computation paths, the above iteration will eventually stop. The resulting choreography has the nice property that it prevents the system entering a deadlock state. ■

Further connections between types of normative systems can be discussed. We only mention them briefly. As we have already pointed out by means of examples, *enforcement* does not necessarily hold in *liberal* societies. In this case, it might be of interest to synthesise certain choreographies which ensure that *enforcement* holds. Naturally, when the participating agents, either by themselves or forced by a choreography, behave honestly, it trivially follows that all normative artifacts have the same power since there is no need to apply normative rules.

4.7 Multi-Agents Systems Refinement

Having fixed *what* is a multi-agent system, we address the problem of *how* to relate them. We recall that at the level of individual agents the relation was refinement, given our interest in incremental design and verification. It is this refinement relation we would like to extend at a multi-agent setup such that we can reuse the previous work modularly. We mainly focus on multi-agent systems where coordination is achieved by means of choreographies. For the sake of completeness, in Section 4.7.2 we discuss the general case of timed choreographed normative multi-agent systems, however only briefly.

Generalising the refinement relation from individual agents to multi-agent systems in the presence of choreographies requires solving a new problem since choreographies may introduce deadlocks. It can be that though there is refinement at the individual agent level, adding a choreography deadlocks the concrete multi-agent system but not the abstract one. We take, as example, a choreography which specifies a BUP agent to execute an action not

defined in the agent program itself (but only in the BUnity specification). In this situation, refinement as trace inclusion trivially holds since the set of traces from a deadlocked state is empty. Our methodology in approaching this problem consists of, basically, formalising the following aspects. On the one hand, we define the semantics of multi-agent systems with choreographies as the set of *maximal traces*, where we make the distinction between a *success* and a *deadlock*. These traces consist of the parallel agents' executions guided by the choreography. We define multi-agent system refinement as maximal trace inclusion. On the other hand, agent refinement becomes *ready trace* inclusion, where a ready trace records not only the actions being executed, but also those ones which *might* be executed. We show that multi-agent system refinement is *compositional*. More precisely, the main result is that agent refinement implies multi-agent system refinement in the presence of *any* choreography. Furthermore, the refined multi-agent system does not introduce deadlocks with respect to the multi-agent system specification. With respect to changing from traces to ready traces, another aspect we discuss is the proof technique for the refinement of multi-agent systems. As one might expect, “readiness” plays the main role, namely, we show that *ready simulation* is indeed a proof technique for the refinement of multi-agent systems.

4.7.1 A Finer Notion of Refinement

To simplify, we mainly focus on describing our methodology only in the case of untimed multi-agent systems. We provide a brief explanation of how this methodology generalises to timed choreographed normative systems in Section 4.7.2.

The question we address in this section is “what are the conditions which ensure that if the agents (for example BUpL) in a multi-agent system \mathcal{I}_1 are refining the (BUnity) agents in \mathcal{I}_2 then the whole system running under a choreography $A^{ch} \otimes \mathcal{I}_1$ is a refinement of $A^{ch} \otimes \mathcal{I}_2$ ’?”. When refinement is defined as trace inclusion, this is, indeed, the case, as we can shortly prove in Proposition 4.7.1.

4.7.1. PROPOSITION. *Given a choreography as A^{ch} and given two multi-agent systems $\mathcal{I}_1, \mathcal{I}_2$ such that $(\forall i_1 \in \mathcal{I}_1)(\exists i_2 \in \mathcal{I}_2)(ms_{i_1} \subseteq ms_{i_2})$ the following inclusion holds: $A^{ch} \otimes \mathcal{I}_1 \subseteq A^{ch} \otimes \mathcal{I}_2$.*

Proof. Let \mathcal{M}_1 and \mathcal{M}_2 be the initial states of the multi-agent systems \mathcal{I}_1 and \mathcal{I}_2 . Let also cs_0 be the initial state of the transition system A^{ch} associated to the choreography c . It is enough to notice that $Tr((cs_0, \mathcal{M}_1)) = Tr(cs_0) \cap Tr(\mathcal{M}_1)$ and that $ms_{i_1} \subseteq ms_{i_2}$ for all $i_1 \in \mathcal{I}_1$ implies $Tr(\mathcal{M}_1) \subseteq Tr(\mathcal{M}_2)$. ■

However, adding choreographies to a multi-agent system may introduce deadlocks. On the one hand, we would like to be able to infer from the semantics when a multi-agent system is in a deadlock state. On the other hand, we would like to have that the refinement of multi-agent systems does not introduce deadlocks. Trace semantics is a too coarse notion with respect to deadlocks. There are two consequences: neither is it enough to define the semantics of a multi-agent system as the set of all possible traces, nor is it satisfactory to define agent refinement as trace inclusion. We further illustrate these affirmations by means of simple examples.

We consider, for instance, the choreography $ch = (i, move(2, 0, 3))$, where i symbolically points to the BUpL agent from Section 2.3. Looking at the plans and repair rules of the BUpL agent we see that such an action cannot take place. Thus, conforming to the transition rule (mas), there is no possible transition for the product $A^{ch} \otimes \mathcal{I}$. Just by analysing the behaviour (the empty trace) we cannot infer anything about deadlocked states: is it that the agent has no plan, or is it that the choreography asks for an impossible execution? This is the reason why, in order to distinguish between successful and deadlocked executions, we explicitly define a transition label \checkmark different from any other action relations. We then define for the product $A^{ch} \otimes \mathcal{I}$ an operational semantics $\mathcal{O}^{\checkmark}(A^{ch} \otimes \mathcal{I})$ as the set of maximal (in the sense that no further transition is possible) traces, ending with \checkmark when the last state is successful:

$$\{tr\checkmark \mid (cs_0, \mathcal{M}_0) \xrightarrow{tr} (cs, \mathcal{M}) \not\vdash, cs \in F(A^{ch})\} \cup \\ \{tr \mid (cs_0, \mathcal{M}_0) \xrightarrow{tr} (cs, \mathcal{M}) \not\vdash, cs \notin F(A^{ch})\} \cup \{\varepsilon \mid (cs_0, \mathcal{M}_0) \not\vdash\},$$

where tr is a trace with respect to the transition rule (mas), \mathcal{M}_0 (resp. cs_0) is the initial state of \mathcal{I} (resp. A^{ch}) and ε denotes that there are no possible transitions from the initial state.

The definition of the operational semantics \mathcal{O} leads naturally to the following definition of multi-agent systems' refinement.

4.7.2. DEFINITION. [MAS Refinement] Given a choreography ch , we say that two multi-agent systems \mathcal{I}_1 and \mathcal{I}_2 are in a refinement relation if and only if the set of maximal traces of $A^{ch} \otimes \mathcal{I}_1$ are included in the set of maximal traces of $A^{ch} \otimes \mathcal{I}_2$. That is, $\mathcal{O}^{\checkmark}(A^{ch} \otimes \mathcal{I}_1) \subseteq \mathcal{O}^{\checkmark}(A^{ch} \otimes \mathcal{I}_2)$. \blacklozenge

The other problem we mentioned in connection to considering agent refinement defined as trace inclusion is explained as follows. It can be the case that the agents in the concrete system refine (with respect to trace inclusion) the agents in the abstract system, nevertheless the concrete system deadlocks for a particular choreography. We consider, for instance, the BUnity and BUpL agents from Sections 2.2 and 2.3. For the ease of reference, we identify the BUnity agent by i_a (since it is more abstract) and the BUpL agent by i_c (since it is more concrete). We can easily design a choreography which works fine with i_a (does not deadlock) and on the contrary, it deadlocks with i_c . Such a choreography is for example the one mentioned in the beginning of the section, $ch = (i, move(2, 0, 3))$, where, i points now to either i_a or i_c up to a renaming. We recall that i_c is a refinement of i_a . However, we have already mentioned, i_c cannot execute the move (since the move is irrelevant for building the *ABC* tower and at implementation time it matters to be as precise as possible), while i_a can (since in a specification “necessary” is more important than “sufficiency”).

What the above illustration implies is that refinement as trace inclusion, though being a satisfactory definition at individual agent level, is not a strong enough condition to ensure refinement at a multi-agent level, in the presence of an arbitrary choreography. It follows that we need to redefine individual agent refinement such that multi-agent system refinement (as maximal trace inclusion) is compositional with respect to any choreography. In this sense, a choreography is more like a context for multi-agent systems, meaning that whatever the context is, it should not affect the visible results of the agents' executions but restrict them by activating only certain traces (the other traces still exist, however, they are inactive).

In order to have a proper definition of agent refinement we look for a finer notion of traces. The key ingredient lies in *enabling* conditions for actions. Given a mental state ms , we look at all the actions enabled to be executed from ms . We recall that in Section 3.4 as well as in Section 4.2 we denoted them by $E(ms) = \{a \in \mathcal{A} \mid \exists ms' (ms \xrightarrow{a} ms')\}$. To relate $E(ms)$ to the notion of traces which we need we call it a *ready* set. We can now present *ready traces* as possibly infinite sequences $X_1, a_1, X_2, a_2, \dots$ where $ms_0 \xrightarrow{a_1} ms_1 \xrightarrow{a_2} ms_2 \dots$ and $X_{i+1} = E(ms_i)$. We denote the set of all ready traces from a state ms_0 as $RT(ms_0)$. Compared to the definition of traces, ready traces are a much more finer notion in the sense that they record not only actions which have been executed but also sets of actions which are *enabled* to be executed at each step.

4.7.3. DEFINITION. [Ready Agent Refinement] We say that two agents with initial mental states ms and ms' are in a ready refinement relation (i.e., $ms \subseteq_{rt} ms'$) if and only if the ready traces of ms are included in the ready traces of ms' (i.e., $RT(ms) \subseteq RT(ms')$). ♦

We can now present our main result which states that refinement is compositional, in the sense that if there is a ready refinement between the agents composing two multi-agent systems it is then the case that one multi-agent system refines the other in the presence of any choreography.

4.7.4. THEOREM. Let $\mathcal{I}_1, \mathcal{I}_2$ be two multi-agent systems such that $(\forall i_1 \in \mathcal{I}_1) (\exists i_2 \in \mathcal{I}_2) (ms_{i_1} \subseteq_{rt} ms_{i_2})$ and a choreography ch with the associated LTS A^{ch} . We have that \mathcal{I}_1 refines \mathcal{I}_2 , that is, $\mathcal{O}^\vee(A^{ch} \otimes \mathcal{I}_1) \subseteq \mathcal{O}^\vee(A^{ch} \otimes \mathcal{I}_2)$.

Proof. What we need to further prove with respect to Proposition 4.7.1 is that the set of enabled actions is a key factor in identifying failures in both implementation and specification. Assume a maximal trace tr in $\mathcal{O}^\vee(A^{ch} \otimes \mathcal{I}_1)$ leading to a non final choreography state cs . Given cs_0 and \mathcal{M}_1 as the initial states of A^{ch}, \mathcal{I}_1 , we have that $(cs_0, \mathcal{M}_1) \xrightarrow{tr} (cs, \mathcal{M})$ $(cs, \mathcal{M}) \not\xrightarrow{l}$ for all $l = \parallel_{j \in \mathcal{J}} (j, a_j)$ such that $cs \xrightarrow{l} cs'$. By rule *(mas)* this implies that there exists an agent identified by j which cannot perform the action indicated. Thus the corresponding trace of j ends with a ready set X with the property that a_j is not included in it. We know that each implementation agent has a corresponding specification, be it j' , such that j ready refines j' . If we, on the other hand, assume that j' can, on the contrary, execute a_j we would have that in a given state j' has besides the ready set X another ready set Y which includes a_j . This contradicts the maximality of the ready set. ■

As a direct consequence of the above theorem, we are able to infer the absence of deadlock in the concrete system from the absence of deadlock in the abstract one:

4.7.5. COROLLARY. Let $\mathcal{I}_1, \mathcal{I}_2$ be two multi-agent systems with initial states \mathcal{M}_1 and \mathcal{M}_2 . Let ch be a choreography with the associated LTS A^{ch} and initial state cs_0 . We have that if \mathcal{I}_1 refines \mathcal{I}_2 ($\mathcal{O}^\vee(A^{ch} \otimes \mathcal{I}_1) \subseteq \mathcal{O}^\vee(A^{ch} \otimes \mathcal{I}_2)$) and c does not deadlock the specification $((cs_0, \mathcal{M}_2) \models \Box \neg \nabla)$ it is then also the case that c does not deadlock the implementation $((cs_0, \mathcal{M}_1) \models \Box \neg \nabla)$.

As we have already explained in Section 2.7.1, proving refinement by deciding trace inclusion is an inefficient procedure. This is also the case with ready refinement, thus a more

adequate approach is needed. If previously we have adopted simulation as a proof technique for refinement, now we consider *weak ready simulation*.

4.7.6. DEFINITION. [Weak Ready Simulation] We say that two agents with initial mental states ms and ms' are in a (weak) ready simulation relation ($ms \lesssim_{rs} ms'$) if and only if $ms \lesssim ms'$ and the corresponding ready sets are equal ($E(ms) = E(ms')$). ♦

As it is the case for simulation being a sound and complete proof technique for refinement, analogously we can have a similar result for ready simulation. We recall that determinacy plays an important role in the proof for completeness.

4.7.7. PROPOSITION. *Given two agents with initial mental states ms and ms' , where the one with ms is deterministic, we have that $ms \lesssim_{rs} ms'$ if and only if $ms' \subseteq_{rt} ms$.*

4.7.8. REMARK. For the sake of generality, in the definitions from this section we have used the symbolic notations ms, ms' . BUnity and BUpL agents can be seen as (are, in fact) particular choices instantiating ms, ms' . Proposition 4.7.7 relates to Proposition 2.7.8. The only difference is that, for simplification, Proposition 4.7.7 refers directly to ready simulation and not to its modal characterisation, as it was the case for simulation in Proposition 2.7.8. It is not difficult to adapt Definition 2.7.6 to the ready simulation. One needs only to change the condition on the transition (mas) from $(\mathcal{B}, p) \xrightarrow{a}_1 (\mathcal{B}', p')$ to the conjunction $(\mathcal{B}, p) \xrightarrow{a}_1 (\mathcal{B}', p') \wedge E((\mathcal{B}, p)) = E((\mathcal{B}'))$ which checks also the equality on the ready sets. ♣

Recalling the BUpL and BUnity agents i_c and i_a , we note that though i_a simulates i_c it is not also the case that it ready simulates. This is because the ready set of the BUnity agent is always larger than the one of the BUpL agent. One elementary argument is that i_a can always “undo a block move”, while i_c cannot. However, if, instead, we change i_a by replacing the trigger set from Figure 2.4 with the set from Figure 4.19: we obtain a less

$$\begin{aligned} \mathcal{A}' = \{ & \neg on(2,1) \triangleright do(move(2,0,1)), \\ & \neg on(3,2) \wedge on(2,1) \triangleright do(move(3,0,2)), \\ & \neg(on(2,1) \wedge on(3,2)) \triangleright do(move(x,y,0)) \} \end{aligned}$$

Figure 4.19: Adapting i_a to ready simulate i_c

abstract BUnity agent which we briefly explain in what follows. The instantiation from the first two triggers disallows any spurious “to and fro” sequence of moves like $move(x,y,z)$ followed by $move(x,y,z)$ which practically undoes the previous step leading to exactly the previous configuration. This instantiation is obvious when one looks at the final “desired” configuration. The last trigger allows “destructing” steps by moving blocks on the floor. With these changes, the new BUnity agent ready simulates i_c . To see this, it suffices to notice that the only BUpL ready trace is $\{move(3,1,0)\}, move(3,1,0), \{move(2,0,1)\}, move(2,0,1), \{move(3,0,2)\}, move(3,0,2)$ which is also the only BUnity ready trace. The new BUnity agent can be still considered as a specification as it provides no information about the order of executing the moves (we recall that orderings should be cast aside at the abstraction level).

We recall the choreography from Figure 4.1 and we consider a BUnity multi-agent system which consists of two copies of i_a (enabled to execute also *clean*). For either branch, the executions (with respect to the transition (*mas*)) of the multi-agent system are successful (the choreography reaches a final state). Since i_c ready refines i_a , by Corollary 4.7.5 we can deduce that also the executions of a multi-agent system which consists of two i_c copies are successful.

4.7.2 A Short Note on TCNMAS Refinement

We begin by first looking at the refinement of individual timed agents. We recall that the key element in having simulation as a proof technique for individual agent refinement was the determinacy of BUnity agents. To have a similar “timed” result we only need to impose the restriction that clock constraints associated with the same action must be disjoint. This restriction ensures determinacy of timed automata. A weaker restriction (which nevertheless requires a “determinisation” construction) is to require that *each basic action is associated with at most one clock* and that *triggers can only reset the clock corresponding to the basic action being executed*; however, the guards in triggers may consult different clocks. Under the disjointness condition, we have that timed BUnity agents are deterministic, thus the same proof technique as in Section 2.7.1 can be applied. We make the remark that *timed simulation* differs from simulation in only one aspect: we further need to consider simulating δ steps and not only a steps.

Switching to the refinement of timed multi-agent system running under timed choreographies, the methodology described in the previous section can be readily adapted. A priori, one aspect worth paying attention to is the possibility of introducing new deadlocks due to the fact that the transition (*delay*) can take place only if all agents are able to delay. However, this is no real problem because delays are not compulsory and thus no deadlocks are being introduced. With this, in order to define timed refinement it suffices to define the semantics of a timed agent system together with a timed choreography as the set of maximal *timed* computations. To have that timed ready simulation is an adequate proof technique for the refinement of timed multi-agent systems, we only need to require that clock constraints associated with the same action are disjoint such that timed choreographies are deterministic.

We conclude with a short example. We take a timed BUPL multi-agent system consisting of two instances of the agent described in Section 4.3.2 running under the timed choreography from Figure 4.3. It is not difficult to see that this BUPL system is a timed refinement of a timed BUnity multi-agent system consisting of two instances of the agent described in Section 4.3.1 running under the same choreography. Furthermore, both systems are deadlock free. To illustrate this latter affirmation, we present a small experiment in UPPAAL [BDL⁺06], a tool for verifying timed automata. At a more abstract and syntactic level, we model the timed choreography and the timed BUPL agent as timed automata in UPPAAL. We use UPPAAL that the choreography always reaches the final state. In our specific context, this reduces to verifying that the values of the clocks are always greater than 6. Figure 4.20 illustrates the timed BUPL system consisting of two instances of the BUPL agent and the choreography. The BUPL agent is parametrised by `id_b`, a bounded integer variable which is in our case 0 or 1. We note that we had to “approximate” and implement the parallel operator using an

interleaving mechanism (first one agent cleans and after the other one moves *C* on the floor). The synchronisation between the choreography and the BUpL agents is in the CCS style (e.g., `clean[1-e]!` and `clean[1-e]?`).

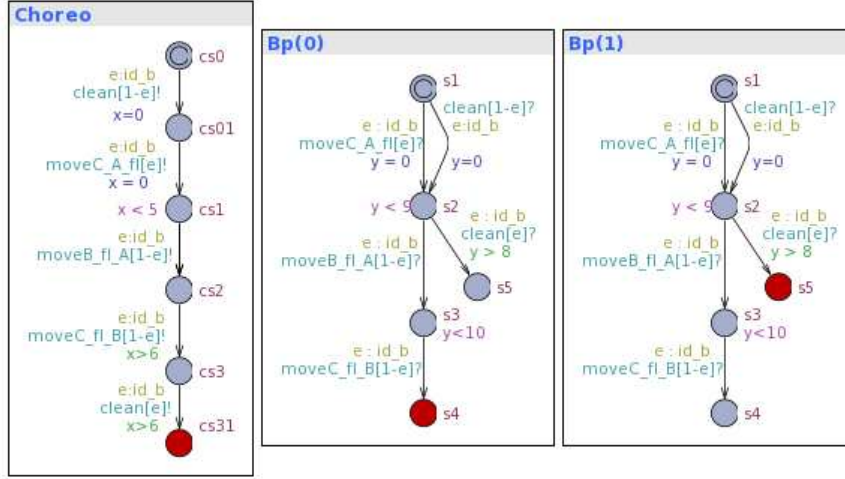


Figure 4.20: A Timed BUpL System Modelled in UPPAAL

Using the UPPAAL `Simulate` command one can experiment with different timed executions of the system. Figure 4.21 represents one of them. The trace shows that the BUpL instance $Bp(0)$ is the first to execute `clean` followed by $Bp(1)$ executing the destructing step (*C* on the floor). From this point $Bp(0)$ finishes the *ABC* tower. Finally, $Bp(1)$ executes, at its turn, the action `clean`.

We conclude by briefly discussing the relevance of the notion of refinement in the context of (timed) choreographed normative multi-agent systems. Intuitively, by fixing a specific normative artifact, the system behaves normatively in a deterministic manner and consequently the above refinement methodology trivially applies. The situation is slightly more complicated when considering the refinement relation between systems with different normative artifacts. One possible approach is to abstract away from the norm application when considering the traces of the system and use the same refinement methodology. This abstraction does not break the definition of refinement, since by refinement we mean “to reduce the non-determinism in agent programs”. Yet the question of what role should the normative rules play in a refinement relation is of particular interest. Any answer could be an adding to the study of the relationship between the normative classes we discussed in Section 4.4.4. In fact, the refinement of normative systems can be a subject on its own and this is why we do not include it in this thesis but rather leave it for future work.

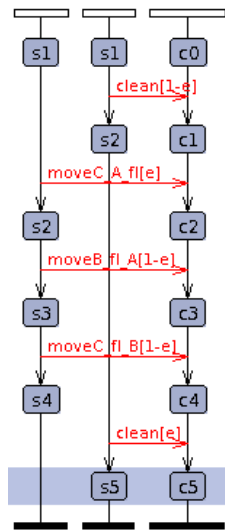


Figure 4.21: A Resulting Timed Trace

Part III

Implementation

In this chapter, we advocate the use of the *Maude* language [CDE⁺09] and its supporting tools for prototyping, verifying, and testing agent programming languages and agent programs. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Maude is a high-performance reflective language and system supporting *equational and rewriting logic specification and programming*. The language has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework through which programming languages with an operational semantics can be implemented in a rigorous way [MOM00b]. Maude comes with an LTL model-checker [EMS02], which allows for verification. Moreover, Maude facilitates the specification of strategies for controlling the application of rewrite rules [EMOMV07].

We will demonstrate how these features of Maude can specifically be applied for developing *agent* programming languages and programs based on solid formal foundations. We use BUpL to illustrate how Maude can be used for prototyping (Section 5.1), model-checking (Section 5.2), and testing (Section 5.3). The complete Maude source code of the implementations discussed in this chapter can be downloaded from <http://homepages.cwi.nl/~astefano/agents/>.

5.1 Prototyping

In this section, we describe how the operational semantics of agent programming languages can be implemented in Maude. The main advantage of using Maude for this is that the translation of operational semantics into Maude is direct [SRM09], ensuring a *faithful implementation*. Because of this, it is relatively easy to experiment with different kinds of semantics, making Maude suitable for rapid *prototyping* of agent programming languages. This is also facilitated by the fact that Maude supports user-definable syntax, offering prototype parsers for free. Another advantage of using Maude for prototyping specifically *logic-based* agent

¹An earlier version of this chapter has appeared as: B.M. van Riemsdijk, L. Aştefănoaei, F. S. de Boer, “Using the Maude Term Rewriting Language for Agent Development with Formal Foundations”, In *Specification and Verification of Multi-Agent Systems/Programs*, Springer, 2010.

programming languages is that Maude has been shown to be suitable not only as a semantic framework, but also as a logical framework in which many other logics can be represented.

We use BUpL to illustrate the implementation of agent programming languages in Maude. BUpL has beliefs and plan revision features, but no goals. We refer to [vRdBDM07] for a description of the Maude implementation of a similar agent programming language that does have goals. While the language of [vRdBDM07] is based on propositional logic, BUpL allows the use of variables, facilitating experimentation with more realistic programming examples. An implementation of the agent programming language AgentSpeak in Maude is briefly described in [FD07].

5.1.1 Introduction to Maude

A rewriting logic specification or rewrite theory is a tuple $\langle \Sigma, E, R \rangle$, where Σ is a signature consisting of types and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the *terms* that form the state of the system. These terms can be rewritten using equations and rewrite rules. Rewrite rules are used to model the dynamics of the system, i.e., they describe transitions between states. Equations form the functional part of a rewrite theory, and are used to reduce terms to their “normal form” before they are rewritten using rewrite rules. The application of rewrite rules is intrinsically non-deterministic, which makes rewriting logic a good candidate for modelling concurrency.

In what follows, we briefly present the basic syntax of Maude, as needed for understanding the remainder of this section. Please refer to [CDE⁺09] for complete information. Maude programs are built from *modules*. A module consists of a *syntax declaration* and *statements*. The syntax declaration forms the signature and consists of declarations for *sorts*, which give names for the types of data, *subsorts*, which impose orderings on data types, and *operators*, which provide names for the operations acting upon the data. Statements are either equations or rewrite rules. Modules containing no rewrite rules but only equations are called *functional modules*, and they define equational theories $\langle \Sigma, E \rangle$. Modules that contain also rules are called *system modules* and they define rewrite theories $\langle \Sigma, E, R \rangle$. Functional modules (system modules) are declared as follows:

```
fmod (mod) <ModuleName> is

    <DeclarationsAndStatements>

endfm (endm)
```

Modules can import other modules, which helps in building up modular applications from short modules, making it easy to debug, maintain or extend.

One or multiple sorts are declared using the keywords `sort` and `sorts`, respectively, and subsorts are similarly declared using `subsort` and `subsorts`. The following defines the sorts `Action` and `Plan` and their subsort relation, which is used for specifying the BUpL syntax.

```
sorts Action Plan . subsort Action < Plan .
```

We can further declare operators (functions) defined on sorts (types) as follows:

```
op <OpName> : <Sort-1> ... <Sort-k> -> <Sort>
                                     [<OperatorAttributes>] .
```

where k is the arity of the operator. For example, the operator declaration below is used to define the BUpL construct plan repair rule. The operator $(_<-_)$ takes a query of sort `Query` that should be tested on the belief base, and a plan, and yields a term of sort `PRrule`. The operator is in mixfix form, where the underscores indicate the positions of its parameters. This also illustrates how Maude can be used to define the syntax of a BUpL language construct.

```
op ( _<-_ ) : Query Plan -> PRrule .
```

Equations and rewrite rules specify how to transform terms. Terms are variables, constants, or the result of the application of an operator to a list of argument terms. Variables are declared using the keywords `var` and `vars`. For example, `var R : PRrule` declares a variable `R` of sort `PRrule`. Equations can be unconditional or conditional and are declared as follows, respectively:

```
eq [<Label>] : <Term-1> = <Term-2> .
ceq [<Label>] : <Term-1> = <Term-2>
               if <Cond-1> /\ ... /\ <Cond-k> .
```

where `Cond- i` is a condition which can be an ordinary equation $t = t'$, a matching equation $t := t'$ (which is true only if the two terms match), a Boolean equation (which contains, e.g., the built-in (in)equality \neq , $=$, and/or logical combinators such as `not`, `and`, `or`), or a membership equation $t : S$ (which means that t is a member of sort S).

For example, the following conditional equation is part of a module for specifying when a formula logically follows from the belief base. The belief base is defined as a commutative sequence of ground belief atoms of sort `Belief`, separated by `#`. The conditional equation specifies that matching term `T` against a belief base containing belief `B` yields substitution `S`, if `match(T, B)` yields a substitution `S` that is different from `noMatch`, the built-in Maude constant to indicate that no substitution has been found.

```
var B : Belief .
var BB : BeliefBase .
var T : Term .
var S : Substitution .

ceq match(T, B # BB) = S if S := match(T, B) /\ S /= noMatch .
```

Operationally, equations can be applied to a term from left to right. Equations in Maude are assumed to be terminating and confluent,¹ i.e., there is no infinite derivation from a term t using the equations, and if t can be reduced to different terms t_1 and t_2 , there is always a term u to which both t_1 and t_2 can be reduced. This means that any term has a *unique normal form*, to which it can be reduced using equations in a finite number of steps.

¹If this is not the case, the operational semantics of Maude does not correspond with its mathematical semantics.

Finally, we introduce rewrite rules. Like equations, rewrite rules can also be unconditional or conditional, and are declared as follows:

```

r1  [<Label>] : <Term-1> => <Term-2> .
cr1 [<Label>] : <Term-1> => <Term-2>
      if <Cond-1> /\ ... /\ <Cond-k> .

```

where `Cond-i` can involve equations, memberships (which specify terms as having a given sort) and other rewrites. We will present several examples in the next section.

5.1.2 Implementing BUpL: Syntax

In this section, we use BUpL to illustrate how the syntax of agent programming languages can be implemented in Maude. We make a distinction between the logical parts of the language and the non-logical parts.

Logical Part

First, we have to define the logical language on which BUpL is based. Logical formulae occur in the belief base (ground atoms), in actions specifications (a formula as precondition, and a set of literals as effects), and in repair rules (a formula as the application condition). For the representation of atoms, the Maude built-in sorts `GroundTerm` and `Term` are used. That is, any Maude (ground) term can be used as an atom of our logical base language. In addition, we define the following sorts to represent also negated (ground) terms and (ground) sets of literals.

```

sorts NegGroundTerm NegTerm GroundLitSet LitSet .

```

The following subsort relations are defined on these sorts. Note that `GroundTerm < GroundLitSet` specifies that any Maude ground term can be a (set of) ground literals, and similarly for `Term < LitSet`.

```

subsorts GroundTerm GroundTermList < GroundLitSet .
subsorts Term NegTerm GroundLitSet < LitSet .
subsort  NegGroundTerm < NegTerm .

```

`GroundLitSet` is defined as a superset of the Maude built-in sort `GroundTermList`, since we use its constant `empty` to represent an empty set of ground literals. The sorts `Belief` and `BeliefBase` are introduced with the subsort relations

```

subsorts Belief < GroundTerm GroundTermList < BeliefBase
      < GroundLitSet .

```

to represent beliefs. The following operators are introduced to syntactically represented (ground) literal sets, belief bases, and negated (ground) terms. The attributes `assoc comm id: empty` declare that the operator is associative and commutative with identity the empty set. The attribute `ctor` declares that the operator is a constructor, which means that it is used to construct terms rather than to apply it as a function and calculate the result. We overload the

operator #, using it for representing both (ground) literal sets and belief bases. The attribute `ditto` specifies that an overloaded operator has the same attributes as the first declaration of the operator (excluding `ctor`).

```
op _#_ : LitSet LitSet -> LitSet [ctor assoc comm id: empty] .
op _#_ : GroundLitSet GroundLitSet -> GroundLitSet [ctor ditto] .
op _#_ : BeliefBase BeliefBase -> BeliefBase [ctor ditto] .

op neg_ : Term -> NegTerm [ctor] .
op neg_ : GroundTerm -> NegGroundTerm [ctor] .
```

We call formulae that are evaluated on the belief base *queries*. The query language is defined over terms as follows. The definition is more general than the DNF of Section 2.3. However, when defining the semantics, formulae are first transformed into DNF.

```
sort Query .
subsort Term < Query .

ops top bot : -> GroundTerm .
op ~_ : Query -> Query [ctor] .
op _/\_ : Query Query -> Query [assoc] .
op _\/_ : Query Query -> Query [assoc] .
```

This completes the specification of the syntax of the logical part of BUpL.

It is important to note that Maude is suitable as a framework in which many logics can be represented, using equations to axiomatise the logic and using rewrite rules as inference rules. This facilitates experimentation with different logics for representing agent beliefs, making the framework flexible.

Non-Logical Part

The non-logical part consists of the specification of actions, plans, procedures, and repair rules. We distinguish between internal and observable actions. This is useful for testing. Actions are specified as functions using equations. The action name is the function name specified as an operator, and applying the equation yields the precondition and effect of the action. Preconditions and effects are defined using the operators $\circ[__, __]$ and $i[__, __]$ for observable and internal actions, respectively. `nilA` is the “empty” action, used to define an empty plan. The code below shows an example specification of the `move` action from the tower of blocks example of Figure 2.6.² The sort `Nat` represents natural numbers.

```
sorts Action I-Action O-Action .
subsorts I-Action O-Action < Action .

ops nilA : -> Action .
op o[_\_, \_] : Query LitSet -> O-Action .
```

²Note that in the specification of the `move` action in Maude, we have added the condition $X \neq Z$, which is easily done using conditional equations.

```

op i[_,_] : Query LitSet -> I-Action .

op on : Nat Nat -> Belief .
op clear : Nat -> Belief .

op move : Nat Nat Nat -> O-Action .

ceq [act] : move(X, Y, Z) = o[on(X, Y) /\ clear(X) /\ clear(Z),
                             neg on(X, Y) # on(X, Z) # clear(Y)
                             # neg clear(Z) # clear(0)]
    if X /= Z .

```

Plans are built from actions, procedure calls (at the end of a plan), sequential composition (`pre`), and non-deterministic choice (`sum`). The operators `pre` and `sum` are declared to be constructors, reflecting the fact that they are used to construct plans. Procedure names are introduced as operators, and a procedure is defined as an equation that yields the plan forming the body of the procedure. For example, the procedure `build` as declared below is used for building a tower of three blocks (321).

```

sort Plan .
subsort Action < Plan .

op pre : Action Plan -> Plan [ctor id: nilA strat (1 0)] .
op sum : Plan Plan -> Plan [ctor comm] .

op build : -> Plan .
eq build = pre(move(2, 0, 1), move(3, 0, 2)) .

```

Note that the operator `pre` has the attribute `strat (1 0)`. This specifies that only its first argument (an action) can be normalised using equations (expressed by the 1), before any equations are applied on the operator `pre` itself (expressed by placing 1 before 0).³ The second argument (a plan) is not normalised using equations. Using this attribute thus changes what a normal form is for the operator `pre`: the normal form is obtained by normalising the operator's first argument and then normalising the operator itself at top level, while leaving the second argument intact. This prevents the continuous application of equations, which would lead to a stack overflow in case a non-terminating procedure is specified. For example, if we would specify a recursive procedure `build` using the equation

```
eq build = pre(move(2, 0, 1), pre(move(2, 1, 0), build)) .
```

without using `strat` in the declaration of `pre`, the continuous application of the equation to normalise `build` as occurring in the right-hand side of the equation would lead to a stack overflow.

Repair rules are defined similarly to procedures, using equations. An operator is introduced to define the name and parameters of the repair rule, and the equation yields the repair rule itself. On the basis of the equations, repair rules can be collected into a repair rule base

³In our implementation, no equations are specified for normalising `pre` itself.

(of sort `PRbase`). The example repair rule `pr` shown below can be used to deal with a failing `move(X, Y, Z)` action. The action fails if `Y` or `Z` are not clear. In this case the repair rule can be applied to move a block to the table (clearing the block on which it was placed), after which it is tried again to build the tower.

```

sorts PRrule PRbase .
subsort PRrule < PRbase .

op ((_<-_)) : Query Plan -> PRrule .
op empty-prb : -> PRbase .
op ___ : PRbase PRbase -> PRbase [assoc comm id: empty-prb] .

ops pr : Nat Nat -> PRrule .
eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                        pre(move(X, Y, 0), build))) .

```

Finally, we define an operator for representing BUPL mental states. The operator takes a label, belief base and plan, and yields a term of sort `LBpMentalState`. The label represents the label of the transitions in the transition system, i.e., it represents which actions have been executed.

```

op <<_,_,_>> : Label BeliefBase Plan -> LBpMentalState .

```

5.1.3 Example BUPL Program

Using the implementation of the BUPL syntax in Maude, one can easily specify BUPL programs in Maude. An example is the following tower building agent, which represents the example agent from Figure 2.6 in Maude. The `move` action and the procedure and plan repair rule have already been introduced above. In addition, the program specifies the initial belief base `bb`, which expresses where blocks are positioned initially and which blocks are clear. Moreover, the initial mental state of the builder agent is specified using the operator `builder`. The initial plan is `build`. Since no actions have been executed yet in the initial mental state, its label is empty. The equation `module-name` is specified to obtain a reference to the module in which the BUPL program is written. This will be used when implementing the semantics.

```

mod AGENT-DATA
  protecting BUPL-SYNTAX .
  protecting NAT .

  eq module-name = 'AGENT-DATA .

  op on : Nat Nat -> Belief .
  op clear : Nat -> Belief .

  op bb : -> BeliefBase .
  eq bb = on(3, 1) # on(1, 0) # on(2, 0) # clear(0) #
          clear(3) # clear(2) .

```

```

op move : Nat Nat Nat -> O-Action .
vars X Y Z : Nat .
ceq [act] : move(X, Y, Z) =
    o[on(X, Y) /\ clear(X) /\ clear(Z),
      neg on(X, Y) # on(X, Z) # clear(Y)
      # neg clear(Z) # clear(0)]
    if X /= Z .

op build : -> Plan .
eq build = pre(move(2, 0, 1), move(3, 0, 2)) .

ops pr : Nat Nat -> PRrule .
eq [pr] : pr(X, Y) = ((on(X, Y) /\ Y > 0 <-
                      pre(move(X, Y, 0), build))) .

op builder : -> LBpMentalState .
eq builder = << bLabel(empty), bb, build >> .
endm

```

5.1.4 Implementing BUpL: Semantics

The implementation of the semantics of BUpL in Maude can again be divided into the implementation of the logical part and of the non-logical part.

Logical Part

Implementing the semantics of the logical part means implementing matching a query against a belief base. Matching takes place both to determine whether an action can be executed, as well as to determine whether a repair rule can be applied. It is defined using the operator `match : Query BeliefBase -> Substitution`, which takes a query and a belief base, and yields a substitution in case the query matches the belief base, and the special substitution `noMatch` otherwise.

This operator is defined by making use of Maude's *reflective* capabilities [Cla00a]. Maude is a reflective logic since important aspects of its meta-theory can be represented at the object level, so that the object level correctly simulates the meta-theoretic aspects. The meta-theoretic aspect that we use here, is matching two terms. Maude continually matches terms when using equations and rewrite rules. This meta-level functionality can be conveniently used to match a term against a belief.

The meta-level operator that can be used for implementing this, is `metaMatch`. This operator takes the meta-representation of a module and two terms, and tries to match these terms in the module. If the matching attempt is successful, the result is the corresponding substitution. Otherwise, `noMatch` is returned. Obtaining the meta-representation of modules and terms can be done using the operators `upModule` and `upTerm`, respectively. The module that we use for this is the module containing the BUpL program, since the belief base is defined there. The name of the module is obtained by defining an equation for the

operator `module-name`, as shown in the example program of Section 5.1.3. The sort `Qid` is a predefined Maude sort for identifiers. The base case for the operator `match`, where a term is matched against a belief, is defined using `metaMatch` as follows.

```
var T : Term .
var B : Belief .

op module-name : -> Qid .

eq match(T, B) =
  metaMatch(upModule(module-name), upTerm(T), upTerm(B)) .
```

Matching a term against a belief base is then defined by making use of the former equation.

```
var S : Substitution .
var BB : BeliefBase .

ceq match(T, B # BB) = S if S := match(T, B) /\ S /= noMatch .
eq match(T, B # BB) = noMatch [otherwise] .
```

For reasons of space, we omit the additional equations for matching composite formulae against a belief base.

Non-Logical Part

As proposed in [VMO03], the general idea of implementing transition rules of an operational semantics in Maude, is to implement them as (conditional) rewrite rules. The premises of a transition rule then form the conditions of the corresponding rewrite rule, and the conclusion forms the rewrite itself.

We illustrate the implementation of transition rules using those for action execution and repair rule application. The transition rule for action execution

$$\frac{a(x_1, \dots, x_n) =_{def} (\varphi, \xi) \in \mathcal{A} \quad a(t_1, \dots, t_n) = (\varphi', \xi') \quad \theta \in Sols(\mathcal{B}, \varphi')}{(\mathcal{B}, a(t_1, \dots, t_n); p') \xrightarrow{a(t_1, \dots, t_n)\theta} (\mathcal{B} \uplus \xi'\theta, p'\theta)} \quad (act)$$

is implemented in Maude as two rewrite rules: one for internal actions and one for observable actions. Here, we present only the rule for observable actions.

```
ops eqSC : -> EquationSet .
eq eqSC = upEqs(module-name, false) .

var OA : O-Action .

crl [exec-OA] : << L:Label, BB, pre(OA, P) >> =>
  << oLabel(getName(OA, eqSC)),
    update(BB, downTerm(substitute(upTerm(effect(OA)), S), 'err)),
    downTerm(substitute(upTerm(P), S), 'err) >>
  if S := match(prec(OA), BB) /\ S /= noMatch .
```

Recall that equations are used to map actions to their specification in terms of preconditions and effects (expressed using the operator $\circ[_, _]$ in case of observable actions). Before Maude applies rewrite rules to a term, it first reduces the term to its normal form using equations. This means that all actions in a plan of a mental state that is rewritten, are first replaced by their preconditions and effects. Any substitutions that are calculated while executing the plan, are therefore applied to these preconditions and effects. This implements the first two conditions of the corresponding transition rule.

In order to implement the third condition, an auxiliary operator `prec` is used, which yields the precondition of an action. The precondition is then matched against the belief base to yield a substitution. The rule can only be applied if a substitution is indeed found, i.e., if the precondition matches the belief base.

Updating the belief base according to the effect of the action is done using the operator `update : BeliefBase GroundLitSet -> BeliefBase`. The ground set of literals, which forms a parameter of this operator, is obtained from applying the calculated substitution S to the effect of the action using the operator `substitute : Term Substitution -> Term`. This operator is general in that it applies a substitution to any term of sort `Term`. In this case, we want to apply the substitution to the effect of an action. This can be done using the operator `upTerm` to obtain the meta-representation of the effect of the action, which is of sort `Term`, and after applying the substitution transforming the term again into its object-level variant using `downTerm`. In a similar way, the calculated substitution is applied to the rest of the plan, according to the transition rule. The operator `getName`, which is used for obtaining the label of the new mental state, retrieves the name of the action (including instantiated parameters) from its precondition/effect specification and the action equations of the BUpL program (obtained using the meta-level built-in Maude function `upEqs`).

The transition rule for applying a plan repair rule

$$\frac{(\mathcal{B}, \alpha; p) \xrightarrow{\alpha\theta'} \varphi \leftarrow p' \in \mathcal{R} \quad \theta \in \text{Sols}(\mathcal{B}, \varphi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \text{ (fail)}$$

is implemented in Maude as the following rewrite rule:

```

cr1 [exec-fail] : << L:Label, BB, pre(A, P) >> =>
  << tLabel, BB, downTerm(substitute(upTerm(P'), S), 'err) >>
  if match(prec(A), BB) == noMatch /\
    (((Q <- P')) PRB) := getPR(eqSC) /\
    S := match(Q, BB) /\ S /= noMatch .

```

The first condition of the rewrite rule checks that the action that is to be executed, cannot be executed (which is the case if no substitution can be found when the precondition of the action is matched against the belief base). This implements the first condition of the transition rule.

The second condition of the rewrite rule implements the second condition of the transition rule as follows. Since repair rules are implemented as equations that yield a repair rule (see Section 5.1.2), we need an operator to collect the rules into a repair rule base. This is done by

`getPR : EquationSet -> PRbase`, which takes the equations corresponding to the repair rules and yields a repair rule base consisting of the rules as defined by the equations.

The third and fourth conditions of the rewrite rule implement matching the condition of the repair rule to the belief base, corresponding to the third condition of the transition rule. The resulting substitution is applied to the plan of the repair rule, which becomes the plan of the next mental state.

5.1.5 Executing an Agent Program

The BUPL example agent from Section 5.1.3 can be executed in Maude using the command `rew builder`. Maude then uses the implemented BUPL semantics to rewrite the term `builder`, which is first reduced to the initial mental state of the builder agent using the equation `eq builder = << bLabel(empty), bb, build >>`, after which other equations and rewrite rules are applied that specify the semantics of BUPL. The Maude output looks as follows.

```
Maude> rew builder .
rewrite in AGENT-DATA : builder .
rewrites: 4722 in 202ms cpu (252ms real) (23264 rewrites/second)
result LBpMentalState:
<< oLabel('move['s_3['0.Zero], '0.Zero, 's_2['0.Zero]]),
clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2), nilA >>
```

This says that the builder finished its execution after moving block 3 onto 2 (the current plan is empty), and that the belief base reflects the current configuration of the blocks, namely the tower 321. The output `'move[...]` is the meta-representation of `move(3, 0, 2)`. For example, `'s_3['0.Zero]` represents the third successor of zero, i.e., 3.

One can also rewrite the builder step by step. For example, the following shows the resulting mental state after one step of rewriting, namely, a τ transition corresponding to handling the failure of action `move(2, 0, 1)` which cannot be executed since block 3 is on top of 1. We can see that the belief base remains unchanged, and the only change is in the current plan. The application of the repair rule `pr` replaces the failing plan by a plan which consists of first executing the action of moving a block (in our case block 3) onto the floor and then trying `build` again. Note that the action is represented by its precondition and effect in the form `o[precondition, effect]`.

```
Maude> rew [1] builder .
rewrite [1] in AGENT-DATA : builder .
rewrites: 4141 in 181ms cpu (228ms real) (22756 rewrites/second)
result LBpMentalState:
<< tLabel,
clear(0) # clear(2) # clear(3) # on(1, 0) # on(2, 0) # on(3, 1),
pre(o[clear(0) /\ (clear(3) /\ on(3, 1)),
neg clear(0) # neg on(3, 1) # clear(0) # clear(1) # on(3, 0)],
build) >>
```

5.2 Model-Checking

In Section 5.1, we have shown how the syntax and semantics of BUPL can be implemented in Maude, and how an example BUPL program can be defined and executed. One of the main advantages of using Maude for agent development is that it supports software development using formal methods. In this section, we show how the Maude LTL model-checker [EMS02] can be used for verifying agent programs. Verification is important in order to ensure that the final agent program is correct with respect to a given specification or that it satisfies certain properties. Properties are specified in *linear temporal logic* (LTL) [MP92] and are verified using a model-checking algorithm. Model-checking *only* works for finite state systems.

We briefly recall some of the LTL concepts which we will refer to in the following sections. The basic LTL formulae are the booleans *true* (\top) and *false* (\perp) and atomic propositions. Inductively, LTL formulae are built on top of the usual boolean connectives like negation and conjunction. Typical LTL operators are *next* (\bigcirc) and *until* (\mathcal{U}). The operator \mathcal{U} can be used to define the connective *eventually*, $\Diamond\phi = \top\mathcal{U}\phi$. The connective \Diamond can be used to further define the connective *always*, $\Box\phi = \neg\Diamond\neg\phi$.

The semantics of LTL formulae is defined in the usual way. The satisfaction of an LTL formula ϕ in a finite transition system S with an initial state s is defined as follows:

$$S, s \models \phi \text{ iff } (\forall \pi \in \text{Paths}(s))(S, \pi \models \phi)$$

which means that the LTL formula ϕ holds in the state s if and only if ϕ holds for any path in $\text{Paths}(s)$, the set of paths in S starting at s . Given a path π , the satisfaction relation for a formula ϕ is defined inductively on the structure of ϕ . We present, as an example, the semantics of the operator “next” and of the connective “until”:

$$\begin{aligned} S, \pi \models_{LTL} \bigcirc\phi & \quad \text{iff} \quad S, \pi(1) \models_{LTL} \phi \\ S, \pi \models_{LTL} \phi\mathcal{U}\psi & \quad \text{iff} \quad (\exists n)(S, \pi(n) \models_{LTL} \psi) \wedge (\forall m < n)(S, \pi(m) \models_{LTL} \phi) \end{aligned}$$

where n, m are natural numbers and $\pi(n)$ denotes the subpath of π starting in the “ n ”-th state on π . Basically, $\bigcirc\phi$ is satisfied in a state if and only if ϕ is satisfied in the successor state. The formula $\phi\mathcal{U}\psi$ holds on a path π if and only if there is a state which makes ψ true and in all the previous states ϕ was true.

Intuitively, a given path π satisfies the temporal formula $\Diamond\phi$ if there exists a state on π which satisfies ϕ . Similarly, π satisfies the temporal formula $\Box\phi$ if there does not exist a state on π which does not satisfy ϕ . By means of these operators, LTL allows specification of properties such as *safety* properties (something “bad” never happens) or *liveness* properties (something “good” eventually happens). These properties relate to the infinite behaviour of a system. We will provide concrete examples in the next sections.

5.2.1 Connecting BUPL Agents and Model-Checker

Maude system modules can be seen as specifications at different levels. On the one hand they can specify *systems* (in our case, BUPL agents), on the other hand they can specify *properties* that we want to prove about a given system. The syntax of LTL is defined in the functional module `LTL` (in the file `model-checker.maude`). The following code, which is a part of

the module `LTL`, shows the declaration of the temporal operators “until” (`U`), “release” (`R`), “eventually” (`<>`) and “always” (`[]`). It further shows the definitions of `<> f` (resp. `[] f`).

```
fmod LTL is
  protecting Bool .
  sort Formula .

  *** primitive LTL operators
  ops True False : -> Formula [ctor ...] .
  op _U_ : Formula Formula -> Formula [ctor ...] .
  op _R_ : Formula Formula -> Formula [ctor ...] .
  ...
  *** defined LTL operators
  op <>_ : Formula -> Formula [...] .
  op []_ : Formula -> Formula [...] .
  ...
  var f : Formula .
  eq <> f = True U f .
  eq [] f = False R f .
  ...
endfm
```

In order to use the Maude model checker, one needs to do two main things: (i) define which sort represents the states of the system that is to be model-checked, and (ii) define the atomic predicates that can be checked on these states. LTL formulae defined over these atomic predicates are then used to specify the property that is to be model-checked.

In our case, the states are the BUpL mental states of sort `LBPmentalState`. In order to express that these are the states of our system, we need the Maude model-checker module `SATISFACTION`, which is defined as follows.

```
fmod SATISFACTION is
  protecting BOOL .

  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

We import this module into our own module `BUPL-PREDS` for defining the BUpL atomic predicates, and declare `subsort LBPmentalState < State` to express that BUpL mental states are to be considered the states of the system that is to be model-checked. Moreover, we use the operator `_|=_` for defining the semantics of the atomic state predicates, which are declared as predicates of sort `Prop`. We define the state predicate `fact(B)` to express that ground atom `B` is believed by the BUpL agent.

```
mod BUPL-PREDS is
  including BUPL-SEMANTICS .
  including SATISFACTION .
  including MODEL-CHECKER .
```

```

including LTL-SIMPLIFIER .

subsort LBpMentalState < State .
op fact : Belief -> Prop .
var B : Belief .
eq << L:Label, B # BB:BeliefBase, P:Plan >> |= fact(B) = true .
endm

```

In the sequel, we will introduce additional state predicates to specify properties of BUPL agents.

5.2.2 Examples

To run the model-checking procedure we need, after loading in the system the predefined file `model-checker.maude`, to call the operator `modelCheck` with an initial state and a formula, specifying the property that is to be checked, as arguments. The result of the algorithm is either the boolean `true` (if the property holds) or a counterexample. The operator `modelCheck` is declared in the system module `MODEL-CHECKER` which is defined in `model-checker.maude`.

```

fmod MODEL-CHECKER is
  including SATISFACTION .
  including LTL .
  subsort Prop < Formula .
  ...
  subsort Bool < ModelCheckResult .
  op modelCheck : State Formula ~> ModelCheckResult [...] .
endfm

```

Recall that `State` and `Formula` are sorts we have already seen declared in the modules `Satisfaction` and `LTL`, respectively (Section 5.2.1).

We can use the predicate `fact` (defined in Section 5.2.1) in order to define safety properties. As an example, we model-check that it is never the case that the agent believes the table is on block 3. The following Maude output shows that the result is the boolean `true`.

```

Maude> red modelCheck(builder, []~ fact(on(0, 3))) .
reduce in AGENT-DATA : modelCheck(builder, []~ fact(on(0, 3))) .
rewrites: 4811 in 196ms cpu (241ms real) (24425 rewrites/second)
result Bool: true

```

The predicate `fact` enables us to express properties of the beliefs of a BUPL agent. In order to express properties of actions, we define another state predicate `taken` using the label of a BUPL state. Recall that the label specifies which action has been executed.

```

mod BUPL-PREDS is
  ...
  op taken : Action -> Prop .
  ceq << oLabel(T), BB:BeliefBase, P:Plan >> |= taken(A) = true
    if T := getName(A, eqSC) .

```

The predicate `taken(A)` is true in a state if the label `T` matches `A`. Note that we cannot match `A` and `T` directly, since `T` is an action name with instantiated parameters, while `A` is an action specified by means of a precondition and effect (that is, it has the form `o[precondition, effect]`). The operator `getName` is used to obtain the name and instantiated parameters of `A` (see Section 5.1.4).

We can use the predicate `taken` to verify that a certain sequence of actions has been executed. For instance, the following Maude output shows that eventually, if block 2 is moved onto block 1 then moving block 3 onto block 2 takes place after this. This is an example of a liveness property.

```
Maude> red modelCheck(builder,
  <> (taken(move(2, 0, 1)) -> 0 taken(move(3, 0, 2)))) .
reduce in AGENT-DATA : modelCheck(builder,
  <> (taken(move(2, 0, 1)) -> 0 taken(move(3, 0, 2)))) .
rewrites: 30 in 1ms cpu (0ms real) (30000 rewrites/second)
result Bool: true
```

We can define more meaningful liveness properties such as *goals* that should be reached from an initial configuration. The equation `g1` defines the predicate `goal321` as being true if the agent believes that block 3 is on block 2 and block 2 is on block 1, expressing that the agent built the tower 321.

```
mod AGENT-DATA-PREDS is
  including BUPL-PREDS .
  including AGENT-DATA .

  op goal321 : -> Prop .
  eq [g1] : goal321 = fact(on(3,2)) /\ fact(on(2,1)) .
endm
```

While the generic BUPL predicates `fact` and `taken` were specified in BUPL-PREDS, the predicate `goal321` is specific to the tower building agent and is consequently specified in the module AGENT-DATA-PREDS.

The following Maude output shows that the result of model-checking `[]<>goal321` is true, meaning that the BUPL agent will always eventually build the tower 321 from the initial configuration.

```
Maude> red modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4816 in 245ms cpu (292ms real) (19580 rewrites/second)
result Bool: true
```

We might be interested in knowing not only that `goal321` is reachable from the initial state, but also in the corresponding trace. For this, it suffices to model-check the negation of `goal321`. This returns a counterexample representing the trace that we want.

```
Maude> red modelCheck(builder, []~ goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []~ goal321) .
```

```

rewrites: 4568 in 188ms cpu (249ms real) (24173 rewrites/second)
result ModelCheckResult: counterexample(
{<< empty-1,..., ... >>,'exec-fail}
{<< tLabel,..., ... >>,'exec-OA}
{<< oLabel('move['s_3['0.Zero'],'s_['0.Zero'],'0.Zero]),
..., ... >>,'exec-OA}
{<< oLabel('move['s_2['0.Zero'],'0.Zero','s_['0.Zero]),
..., ... >>,'exec-OA},

{<< oLabel('move['s_3['0.Zero'],'0.Zero','s_2['0.Zero]),
clear(0) # clear(3) # on(1, 0) # on(2, 1) # on(3, 2),
nilA >>,'deadlock}
)

```

To understand the counterexample we first detail the predefined operator `counterexample`. This operator is declared in the module `MODEL-CHECKER`. It is formed by a pair of transition lists:

```

op counterexample : TransitionList TransitionList ->
ModelCheckResult [ctor] .

```

A transition list is composed of transitions, and a transition records a state and the name of the rule which has been applied from that state.

```

subsort Transition < TransitionList .
op {_,_} : State RuleName -> Transition [ctor] .
op _ : TransitionList TransitionList ->
TransitionList [ctor assoc id: nil] .

```

The first list of `counterexample` represents the shortest sequence of transitions (which record the states being visited) that leads to the first state of a loop. This loop is represented by the second list from `counterexample`. In our example, the first list consists of four transitions. It shows that first the rewrite rule `exec-fail` has been applied from the initial state (for readability, the belief base and plan are omitted), and consequently the label of the next state denotes a τ step. Then, the rule `exec-OA` is applied, which changes the label of the next state into the meta-representation of the action `move(3, 1, 0)`. A similar reasoning applies for the next transition.

The second list of the counterexample (after the white line) consists of only one transition. The initial plan has terminated (the action `nilA` is reached) and the belief base reflects that tower 321 is built. The rule name from this last transition is `deadlock`, a predefined constant which is declared in `MODEL-CHECKER`. It means that from the state that the agent reached, no further rewrite rule is applicable. Thus, the system “cycles” in a deadlock state and this is the loop represented by the second transition list. We note that a Maude deadlock state is, in our case, a termination BUPL state.

5.2.3 Fairness

The BUPL agent we have described always terminates, i.e., all execution paths are finite. Infinite behavior can occur due to recursive abstract plans, and because of the non-determinism

of the operator `sum`. The reason in the latter case is that it is possible that the choice between a failing and a terminating action goes always in favour of the failing one. We call such behavior *unfair*.

In practise, unfair traces are generally prevented from occurring through scheduling algorithms such as round-robin. However, at the level of prototyping BUpL in Maude we would like to abstract from controlling the non-deterministic choices. Rather, non-determinism is reduced at a later phase of design, at a more concrete implementation level. We stress that it is important to abstract from control issues at the prototype level, since the main concern is to experiment with language definitions rather than scheduling algorithms.

Nevertheless, when model-checking BUpL agents one may want to ignore unfair traces and show that the agent satisfies certain properties assuming fairness. Since we work in a declarative framework, our solution is to model-check only the traces that satisfy certain fairness constraints and to define fairness using LTL. To illustrate this, we first introduce the predicate `enabled`. The proposition `enabled(A)` holds in a state if the action `A` can be executed in that state, i.e., if the action's precondition holds.

```
op enabled : Action -> Prop .
ceq << L:Label, BB, P >> |= enabled(A) = true
    if match(prec(A), BB) /= noMatch .
```

Following [MP92], we then define fairness with respect to an action as follows.

```
op fair : Action -> Prop .
eq fair(A) = <>[] enabled(A) -> []<> taken(A) .
```

This says that if an action is continuously enabled it should be infinitely often taken. This requirement casts aside traces where the failing action is always chosen in spite of a terminating action *a* since such traces are unfair with respect to *a*.

For a concrete example where fairness is useful, we modify the BUpL example from Section 5.1.3 such that the initial plan of the agent is `p1`, which is defined as a non-deterministic choice (`sum`) between an always failing action and the plan `build`. We further add an always enabled repair rule `pr1` to handle the case where the failing action has been chosen in `p1`.

```
eq p1 = sum(i[bot, empty], build) .
ops pr1 : -> PRrule .
eq [pr1] : pr1 = (( top <- p1 )) .
...
eq builder = << bLabel(empty), bb, p1 >> .
```

It is now the case that achieving `goal321` is no longer always possible, demonstrated by the following counterexample, which is generated when model-checking the property `[]<> goal321`.

```
Maude> red modelCheck(builder, []<> goal321) .
reduce in AGENT-DATA-PREDS : modelCheck(builder, []<> goal321) .
rewrites: 4875 in 209ms cpu (254ms real) (23217 rewrites/second)
result ModelCheckResult: counterexample(
```

```

{<< empty-1, ..., ... >>, 'sum}
{<< tLabel, ..., ... >>, 'exec-fail},

{<< tLabel, ..., ... >>, 'sum}
{<< tLabel, ..., ... >>, 'exec-fail})

```

The counterexample shows that first the failing action was chosen to be executed, which is then handled by the repair rule `pr1`. In this counterexample, this leads to a loop in which over and over the failing action is chosen and then the repair rule is applied. This loop is represented in the second parameter of `counterexample` (below the white line).

However, if we consider the paths which are fair with respect to `move(3, 1, 0)` then we have that `goal321` is always achieved.

```

Maude > reduce in AGENT-DATA-PREDS :
modelCheck(builder, fair(move(3, 1, 0)) -> []<> goal321) .
rewrites: 9097 in 196ms cpu (231ms real) (46184 rewrites/second)
result Bool: true

```

5.3 Testing

In the previous section, we have illustrated how Maude can be used for model-checking BUpL agents, using the tower builder of Section 5.1.3 as an example. Since the tower builder has a finite number of mental states, verification by model-checking is in principle feasible. However, the state space of agents can also be infinite, making direct model-checking impossible. This issue may be addressed within the context of model-checking, e.g., by investigating abstractions techniques for reducing the state space. In this section, however, we are concerned with a different technique than model-checking, namely *testing*. Testing can be used for identifying failures in infinite state systems or in finite state systems where the state space becomes too large for model-checking.

In this section, we present two kinds of testing that fit Maude very well. The first is testing for satisfaction of invariants by means of search (Section 5.3.1), and the second is testing through the specification of test cases (Section 5.3.3). The latter is implemented by means of Maude strategies, which are used to control the application of rewrite rules on a meta-level.

The running example that we use in this section is a variant of the tower builder introduced previously. Here we consider a tower builder that should respect the specification “the agent should continually construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number”. Since the agent keeps on building higher towers, its state space is infinite. We assume that the programmer decides to refine the specification and tries to implement a BUpL agent that builds towers where the constituting blocks are assigned consecutive numbers, thus 21 and 4321 are examples of “well-formed” towers.

Initially, there is one block and it is on the table. In order to indicate that the agent has finished building a tower of length X , it inserts a predicate `done(X)` in the belief base by means of the action `finish(X, Y)` (where Y is added for technical reasons that we

do not further explain). For indicating that the next tower that is to be built has length X , the agent uses a predicate $\text{max}(X)$. The predicate $\text{length}(X)$ is used to represent the current length X of the tower. The builder agent is executed by rewriting a term of the form $\text{builder}(X, Y)$, where X is the length of the tower that is to be built as the first one, and Y is added for technical reasons that we do not further explain. For illustration purposes, we consider two variants of this tower builder: a correct one and a faulty one that builds odd length towers. Since it is not needed for explaining the techniques presented in this section, we do not provide the code for these tower builders.⁴

5.3.1 Searching

Maude provides a `search` command that can be used, among other things, to test for the satisfaction of invariants. Invariants are defined as properties of states. Search is breadth-first, which means that if there is a state where the invariant does not hold, then the search terminates.

Searching in Maude for invariants can be done using the Maude search command with parameters of the following form.

```
search init =>* x:k such that I(x:k) /= true .
```

Here, `init` is the initial state from which the search starts. It searches for states x of sort k that are reachable from this initial state through zero or more rewrite steps (represented by `=>*`) and for which the invariant I does not hold. This is helpful when verifying safety properties. For example, an invariant for the BUPL builder is the length of the towers, which should always be even. This invariant can be specified by means of a predicate `doneEven` as follows.

```
mod BUPL-BUILDER-INVARIANTS is
  including AGENT-DATA .

  op doneEven : LBpMentalState -> Bool .
  ceq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = true
    if (2 divides X) .
  eq doneEven(<< L:Label, done(X) # BB, P:Plan >>) = false
    [otherwise] .
  var MS : LBpMentalState .
endm
```

If we search, in the faulty implementation, for a state where the property `doneEven(MS) /= true` with `MS` being a variable of sort `LBpMentalState` is satisfied, then we obtain a solution, i.e., a state where the invariant does not hold (`done(3)` appears in the belief base):

```
search in BUPL-BUILDER-INVARIANTS :
  builder(3, 0) =>* MS such that doneEven(MS) /= true .
```

⁴It can be downloaded from <http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php>.

```

Solution 1 (state 11)
states: 12  rewrites: 21030 in 1220ms cpu (1301ms real)
(17226 rewrites/second)
MS --> << ..., clear(0) # clear(3) # length(3) # max(3) #
        done(3) # on(1, 0) # on(2, 1) # on(3, 2),
        ... >>

```

However, this procedure terminates only when the implementation is faulty, since in the correct implementation no state would be found where the invariant does not hold. A possible solution is to bound the search. This can be done by explicitly giving a depth bound, for example 100, as in the following example where the correct implementation is searched.

```

search [1, 100] in BUPL-BUILDER-INVARIANTS :
    builder(3, 0) =>* MS such that doneEven(MS) /= true .

No solution.
states: 10  rewrites: 15266 in 779ms cpu (821ms real)
(19574 rewrites/second)

```

5.3.2 Rewrite Strategies in Maude

We recall that state terms t are BUPL mental states. In order to rewrite `builder(3, 0)` using a strategy E , we only need to input the command `srew builder(3, 0) using E` after loading the Maude file `maude-strat.maude` where the strategy language is defined. If E is a rule name, for example, `exec-IA`, then the result is the mental state after performing an internal action, in this case setting `max(3)` which corresponds to the first parameter of `builder(3, 0)`.

```

Maude> (srew builder(3, 0) using exec-IA .)
rewrites: 1384 in 30ms cpu (55ms real) (44652 rewrites/second)
rewrite with strategy :
result LBpMentalState :
    << iLabel('set-max['s_3['0.Zero], '0.Zero]),
    clear(0) # done(0) # length(1) # max(3) # on(1, 0),
    ... >>

```

Strategies are declared and defined only in strategy modules. Strategy modules have the following syntax:

```

smod <STRAT-MODULE-NAME> is
    protecting <M> .
    including <STRAT-MODULE-NAME1> . ...
    including <STRAT-MODULE-NAMEk> .
    <DeclarationsAndDefinitionOfStrategies>
endsm

```

where M is the module containing the terms which we want to rewrite using strategies defined in the imported strategy modules `STRAT-MODULE-NAME1, ..., STRAT-MODULE-NAMEk`.

Similarly to the declaration of operators, strategies are declared using the following format:

```
strat <STRAT-NAME> : <Sort-1> ... <Sort-m> @ <Sort> .
```

where *Sort* is the sort of the term which will be rewritten using the strategy *STRAT-NAME*. Like equations, strategies can be unconditional or conditional and are defined using the following syntax:

```
sd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> .
csd <STRAT-NAME>(<P1>, ..., <Pm>) := <Exp> if <Cond> .
```

with *Pi* being the parameters of the strategy *STRAT-NAME* and *Exp* being a strategy expression.

5.3.3 Using Maude Strategies for Implementing Test Cases

We now illustrate how the test definitions of Section 3.4.2 can be implemented by means of Maude strategies. First, we show how the syntax of tests can be specified as a Maude functional module. We then describe a generic strategy *test2Strat* which associates to each test a corresponding strategy that implements the test. Finally, we focus on the implementation of the basic test *a*.

The following module defines the syntax of tests, in correspondence with the BNF grammar for tests of Section 3.4.2.

```
fmod TEST-SYNTAX is
  protecting SYNTACTICAL-DEFS .
  sort TestA .
  subsort O-Action < TestA .
  op _;a_ : TestA TestA -> TestA .
  op _+a_ : TestA TestA -> TestA .
  op _*a_ : TestA -> TestA .
endfm
```

The code shows that we first declare a sort *TestA* for denoting tests. In order to express that any observable action is a test we use the subsort relation *subsort O-Action < TestA*. Further, we declare regular expression operators to construct new tests. We use the index *a* in their declaration in order to distinguish them from the regular expression operators defined for Maude strategies.

Given that we have defined the syntax of tests as above, we can define the rewrite strategy *test2Strat* inductively on the structure of tests:

```
strat test2Strat : Test @ LBpMentalState .
var Oa : O-Action . vars Ta1 Ta2 : TestA .
sd test2Strat(Oa) := do(Oa) .
sd test2Strat(Ta1 ;a Ta2) := test2Strat(Ta1) ; test2Strat(Ta2) .
sd test2Strat(Ta1 +a Ta2) := test2Strat(Ta1) | test2Strat(Ta2) .
sd test2Strat(Ta1 *a) := test2Strat(Ta1) * .
```

The strategy `do` is meant to implement the basic test a . Note the natural mapping from tests to the corresponding strategy.

We now focus on describing how to implement the basic test a , i.e., the strategy `do`. We recall that, when applied to a mental state ms , this test succeeds only if after performing some internal steps (corresponding to internal actions, repair rules, and choices among plans) the agent reaches a state where a is enabled. This means that we need to implement a strategy, `tauClosure`, for computing the transitive closure of τ steps. A simple⁵ way to do this is as follows:

```
strat tauClosure : @ LBpMentalState .
sd tauClosure := (sum | exec-fail | exec-IA)! .
```

that is, by non-deterministically applying one of the rules which correspond to τ steps until no longer possible. Given that we have the strategy `tauClosure`, the implementation of the test a is straightforward:

```
strat do : O-Action @ LBpMentalState .
sd do(Oa) := tauClosure ; exec-OA[OA <- Oa] .
```

where `exec-OA[OA <- Oa]` applies `exec-OA` with the variable `OA` from the definition of the rewrite rule being instantiated by the argument `Oa` of the strategy. Note that the strategy `tauClosure` returns precisely those states from which no τ steps are possible, that is, the states where the head of the current plan is an observable action. If this observable action is the one given as argument to the strategy `do` then it succeeds and computes again the transitive closure. Otherwise, it fails. To see how this strategy works in practise, we strategically execute `builder(3, X:Nat)` using `do(move(2, 0, 1))`. This means that we test whether the agent executes `move(2, 0, 1)` as the first observable action.

```
Maude> (srew builder(3,X:Nat) using do(move(2,0,1)) . )
rewrites: 18463 in 1415ms cpu (1417ms real) (13040 rewrites/second)
rewrite with strategy :
result LBpMentalState :
  << oLabel('move['s_2['0.Zero], '0.Zero, 's_['0.Zero]]),
      clear(0)# clear(2)# clear(3)# done(0)# length(1)# max(3)#
      on(1,0)# on(2,1)# on(3,0), ...>>
Maude> (next .)
rewrites: 1210 in 10ms cpu (11ms real) (110020 rewrites/second)
next solution rewriting with strategy :
No more solutions .
```

What we obtain is a state reflecting that the agent moved block 2 onto block 1. This can be seen either from the label of the resulting mental state, or from the fact that `on(2,1)` is in the current belief base. Furthermore, we can also notice that this is the only possible resulting mental state since the command `(next .)` for obtaining other solutions returns `No more solutions`.

⁵The strategy described here does not always terminate. One immediate solution is to bind the number of iterations. For a more detailed discussion, we refer to Section 3.4.

We recall that our purpose is to test whether “bad” states are reachable from the initial configuration of `builder` and that “bad” means odd length towers in our case. Thus, a suitable test is `move(2,0,1);move(3,0,2);finish(3,0)`, meaning that we test whether the agent (in its faulty variant) executes the action `finish(3,0)` after moving block 2 onto 1 and block 3 onto 2:

```
Maude> (srew builder(3,X:Nat) using
      test2Strat(move(2,0,1) ;a move(3,0,2) ;a finish(3, 0)) .)
rewrites: 50421 in 2069ms cpu (2082ms real) (24361 rewrites/second)
rewrite with strategy :
result LBPmentalState :
  << oLabel('finish['s_^3['0.Zero], '0.Zero]),
      clear(0)# clear(3)# done(3)# length(3)# max(3)#
      on(1,0)# on(2,1)# on(3,2), ...>>
```

The output shows that this is indeed the case, meaning that the agent is not safe to this test. Performing the same test on the correct builder yields no possible rewriting, and from this we can conclude that the correct builder agent is safe with respect to the test.

5.4 Executable Normative Multi-Agent Systems

We prototype the language for normative multi-agent systems in two modules: the first one, which we call `SYNTAX`, is a functional module where we define the syntax of the language, and the latter, which we call `SEMANTICS`, is a system module where we implement the semantics, namely the transition rule (ACS).

We recall that the state of a normative multi-agent system is constituted by the states of the agents together with the set of brute facts (representing the environment) and normative facts. The following lines, extracted from the module `SYNTAX`, represent the declaration of a normative multi-agent system and the types on which it depends:

```
sorts BruteFacts NormFacts NMasState .
op <_,_,_> : AgentSet BruteFacts NormFacts -> NMasState .
```

The brute (normative) facts are sets of ground literals. The effects are implemented by means of two projection functions, `pre` and `post` which return the enabling condition and the effect of a given action executed by a given agent:

```
op pre : Action Qid -> Query .
op post : Action Qid -> LitSet .
```

Norms or sanctions are implemented similarly. Both have two parameters, an element of type `Query` representing the conditions, and an element of type `LitSet` representing the consequent. Take, for example, the declaration of `norm(s)`:

```
sorts Norm Norms . subsorts Norm < Norms .
op norm : Query LitSet -> Norm .
op *_ : Norms Norms -> Norms [assoc comm] .
```

The effect of a norm is to update the collection of normative facts whenever its condition matches either the set of brute facts or the set of normative facts:

```
op applyNorms : Norms Norms BruteFacts NormFacts NormFacts
  -> NormFacts .
ceq applyNorms(NS, norm(Q, E) * NS', BF, NF, OldNF) =
  applyNorms(NS, NS', BF, update(NF, E), NF)
  if matches(Q, BF ; NF) /= noMatch .
```

where NS is an auxiliary variable which we need in order to compute the transitive closure of the normative set:

```
ceq applyNorms(NS, empty, BF, NF, OldNF) =
  applyNorms(NS, NS, BF, NF, NF) if NF /= OldNF .
eq applyNorms(NS, empty, BF, NF, NF) = NF [owise] .
```

meaning that we apply the norms until no normative fact can be added anymore.

The application of norms entails the application of sanctions which, in a similar manner, update the brute facts when their conditions match the set of normative facts:

```
ceq applySanctions(SS, sanction(Q, E) * SS', NF, BF, OldBF) =
  applySanctions(SS, SS', NF, update(BF, E), BF)
  if matches(Q, NF) /= noMatch .
```

In a normative multi-agent system certain actions of the agents are monitored. Actions are defined by their pre- (enabling) and their postconditions (effects). We recall the basic mechanism which takes place in the normative multi-agent system when a given monitored action is executed. First the set of brute facts is updated with the literals contained in the effect of the action. Then all possible norms are applied and this operation has as result an update of the set of normative facts. Finally all possible sanctions are applied and this results in another update of the brute facts. The configuration of the normative multi-agent system changes accordingly if and only if it is not the case that `violationReg`, the literal we use to ensure regimentation (corresponding to `violl` in Section 4.4.3), appears in the brute facts. Consequently, the semantics of the transition rule (ACS) is implemented by the following rewrite rule:

```
cr1 [ACS] : < A * AS, BF, NF > =>
  < A' * AS, BF'; BF'', NF' >
  if A => [Act] A'
  /\ S := matches(pre(Act, Id), BF) /\ S /= noMatch
  /\ BF' := update(BF, substitute(post(Act, Id), S))
  /\ NF' := setminus(applyNorms(nS, nS, BF', NF, NF), BF')
  /\ BF'' := setminus(applySanctions(sS, sS, BF', NF', BF'), NF')
  /\ matches(violationReg(Id), NF') == noMatch .
```

where `nS`, `sS` are constants defined as the sets of instantiated norms, sanctions. Please note that we implement negation as *failure* and this implies that our `update` function preserves the consistency of the set of facts.

Given the above, we can proceed and describe how we can instantiate a concrete normative multi-agent system. We do this by creating a system module `PSG-NMAS` where we implement the constructions specified in Figure 4.10:

```

mod PSG-NMAS is
  including SEMANTICS .
  including BUPL-SEMANTICS .
  op psg : Qid BpMentalState -> Agent .
  eq pre(enter, X) = ~ at-platform(X) .
  eq post(enter, X) = at-platform(X) .
  eq pre(buy-ticket, X) = ~ has-ticket(X) .
  eq post(buy-ticket, X) = has-ticket(X) .
  eq pre(embark, X) = at-platform(X) /\ ~ in-train(X) .
  eq post(embark, X) = in-train(X) /\ ~ at-platform(X) .
  ops n r : Qid -> Norm .
  eq [norm] : n(X) = norm(at-platform(X) /\ ~ has-ticket(X),
    ticket-violation(X)) .
  *** ( eq [reg] : r(X) = norm(in-train(X) /\ ~ has-ticket(X),
    violationReg(X)) . )
  op s : Qid -> Sanction .
  eq [sanction] : s(X) = sanction(ticket-violation(X),
    pay-fee-ticket(X)) .
  op nmas-state : Qid -> NMasState .
  eq [init] : nmas-state(X) = < psg(X), nil, nil > .
endm

```

The operator `psg` associates an identity to a BUPL agent. We stress that using BUPL agents is only a choice. Any other agent prototyped in Maude can be used instead. The actions being monitored are `enter`, `buy-ticket`, `embark`, with obvious pre- and postconditions. The equation `norm` defines a norm which introduces a ticket violation and the equation `sanction` introduces a punishment in the case of a ticket violation. The equation `reg` defines the normative enabling condition for the action `enter`, making it impossible for `psg` to be in the train without a ticket. However, it will not be taken into consideration because it is in a comment block and the reason will be clear in the next section. We further consider that `psg` has a plan which consists of a sequence of only two actions, `enter`; `embark`, meaning he tries to embark without a ticket. This gives rise to special situations where model-checking turns out to be useful, as we will see in the following.

In order to model-check the system defined in the module `PSG-NMAS` we create a module `PSG-NMAS-PREDS` where we implement the predicates regimentation and enforcement as introduced in Section 4.4.3. Creating a new module is justified by the fact that state predicates are part of the *property specification* and should not be included in the *system specification*.

```

mod PSG-NMAS-PREDS is
  including PSG-NMAS .
  protecting SATISFACTION .
  extending LTL .
  subsort NMasState < State .
  op fact : Lit -> Prop .

```

```

ceq < AS, BF, NF > |= fact(L) = true if in(L, BF) = true .
ops enforcement regimentation : Qid -> Prop .
eq [enf] : enforcement(X) =
  fact(at-platform(X)) /\ not fact(has-ticket(X))
  -> <> fact(pay-fee-ticket(X)) .
eq [reg] : regimentation(X) =
  [] (fact(in-train(X)) -> fact(has-ticket(X))) .
endm

```

The state predicate `fact(L)` holds if and only if there exists a ground literal `L` in the set of brute facts of the normative multi-agent system. We need this predicate in order to define the properties enforcement and regimentation, which we are interested in model-checking. The equation `enf` defines the predicate `enforcement` such that it holds if and only if any agent `X` which is at the platform and has no ticket (`fact(at-platform(X)) /\ not fact(has-ticket(X))` can be entailed from the brute facts) will eventually pay a fee. On the other hand, the equation `reg` defines the predicate `regimentation` such that it holds if and only if it is always the case that any agent in the train has a ticket.

If we model-check whether enforcement holds for an agent identified by `a1`:

```

Maude> red modelCheck(nmas-state('a1), enforcement('a1)) .
reduce in PSG-NMAS-PREDS :
  modelCheck(nmas-state('a1), enforcement('a1)) .
result Bool : true

```

we obtain `true`, thus the normative structure enforces `a1` to pay a fee whenever it enters without a ticket. This is not the case for regimentation, the result of model-checking is a counter-example illustrating the situation where the agent enters the train without a ticket. However, if we remove the comment of the equation labelled `reg` in `PSG-NMAS` the application of the regimentation rule results in the update of the normative facts with `violationReg('a1)`. Consequently, ACS is not applicable and `nmas-state('a1)` is a deadlock state. The result of the model-checking is `true`, since `in-train('a1)` is not in the brute facts. We note that trivially regimentation would hold if the plan of `psg` consisted in buying a ticket before embarking.

Chapter 6

Coordinating 2APL with Reo Artifacts

As a short illustration of how to practically deal with both communication and coordination in multi-agent systems, we present a tool integration experiment. We take advantage of the existing software Reo and 2APL. The 2APL framework consists of agents and environments to which the agents have access. We understand nodes in a Reo network as a particular environment through which the agents can communicate. By connecting channels in a particular way, we obtain specific infrastructures on top of the nodes, which can be used as a coordination mechanism which, for example, restricts the execution of the agents. We explain how Reo and 2APL are integrated by means of a auction scenario.

The advantages of this approach are as follows. One important feature of the Rem language lies in the concept of “exogenous coordination”. A direct consequence is that there is a clear separation between execution (of agent programs) and control (of executions). This makes analysis, extensions much cleaner and modular. It is also the case that there exists a wreath of tools, “The Eclipse Coordination Tools”¹, which provide facilities for designing and verifying Reo networks.

6.1 A Short Overview of Reo

Reo [Arb04, BSAR06] is a channel-based exogenous coordination language wherein complex coordinators, called *connectors*, are built out of simpler ones. Reo can be understood as a “glue language” for compositional construction of connectors which represent coordination artifacts in component-based systems. The emphasis in Reo is on connectors and their composition, not on the components, which are seen as “black-boxes”. The connectors impose a specific behaviour on the components, without the knowledge of the internal structure of the components.

The mechanism for constructing connectors is channel composition. *Channels* are primitive connectors, with two ends which can be either *source* or *sink*. At a source end data enters the channel by performing a corresponding *write* operation, while at a sink end data leaves the channel by performing a corresponding *read* operation. Reo imposes no restriction on the behaviour of the channels and thus it allows an open-ended set of channel types with

¹The Eclipse Coordination Tools are at <http://homepages.cwi.nl/~koehler/ect/>

user-defined semantics. Figure 6.1 depicts the graphical representation of three basic chan-

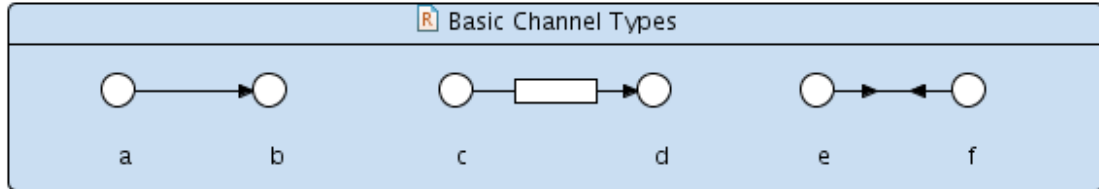


Figure 6.1: Basic Channel Types in Reo

nel types: *ab* is a synchronous channel (*Sync*), *cd* is a one buffer cell asynchronous FIFO channel (*FIFO1*), and *ef* is a synchronous drain channel (*SyncDrain*). Synchronous and FIFO channels have both a source and a sink end each. In a *Sync* channel data is simultaneously accepted at the source end and dispensed at the sink end. In a *FIFO1* channel data is accepted at the source only if the buffer is empty and data is dispensed at the sink end only if the buffer is full. *SyncDrain* channels have two source ends and no sink. In a *SyncDrain* channel data is simultaneously accepted at the source ends and then destroyed.

Channels are composed via a join operation in a node which consists of a set of channel ends. Such a node is either source, sink, or mixed depending on whether all channel ends which coincide on the node are only source, only sink or a combination of source and sink. Source and sink nodes represent input and output ports where components connect to the network. A component can write data to a source node (input port) only if all source ends coincident on the node accept the data, in which case the data is written on each source end. Source nodes, thus, replicate data. A component can obtain data from a sink node (output port) only if at least one of the sink nodes coincident on the node offers data. In the case of more offers one is chosen nondeterministically. Sink nodes, thus, nondeterministically merge data. We take as an example the Reo diagram shown in Figure 6.2. This diagram represents

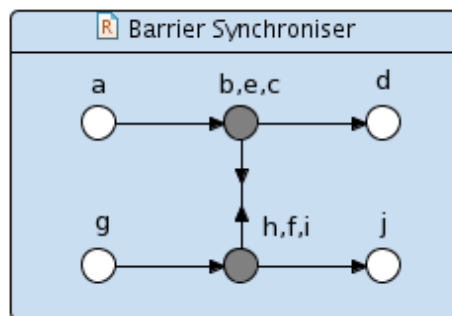


Figure 6.2: A Barrier Synchroniser Connector

a Reo connector which we use later in the paper. It implements a barrier synchronisation: by definition, the SyncDrain channel ef ensures that a data item passes from ab to cd only simultaneously with the passing of a data from gh to ij (and vice-versa). In this section we briefly describe the process of building up a custom Java application which implements a component-based system coordinated by a Reo connector. The design mechanism relies on a bundle of plugins for the Eclipse development environment called “The Eclipse Coordination Tools”.

We take as an illustration a system with two components which simply alternate write and read operations. We assume that the behaviour of one component is to first write to the source node a and then read from the sink node g , and the same for the other one (a write to g is followed by a read from j). We further assume that the writings are controlled by a barrier synchroniser, and in this way, we have a simple mechanism of coordinating the components. Basically, the programmer starts by drawing the Reo connector from Figure 6.2 using the Reo editor. This diagram is automatically converted to the Java code which we denote as `Barrier` in Figure 6.3. Given that the components are implemented as Java threads (`Comp1` and `Comp2` in Figure 6.3), the programmer simply needs to drag and drop their corresponding code and the code for the barrier synchroniser into the Casp editor, which is meant to facilitate the programmer to wire the components to the coordinator. After the linking is completed, the system automatically generates code that implements the whole application (i.e., it generates a Java class where the constructors for `Comp1`, `Comp2`, `Barrier` are properly instantiated and the corresponding threads are started). Note that connectors can be exported as Reo libraries which can be later on reused. A growing collection of commonly useful connectors already exists.

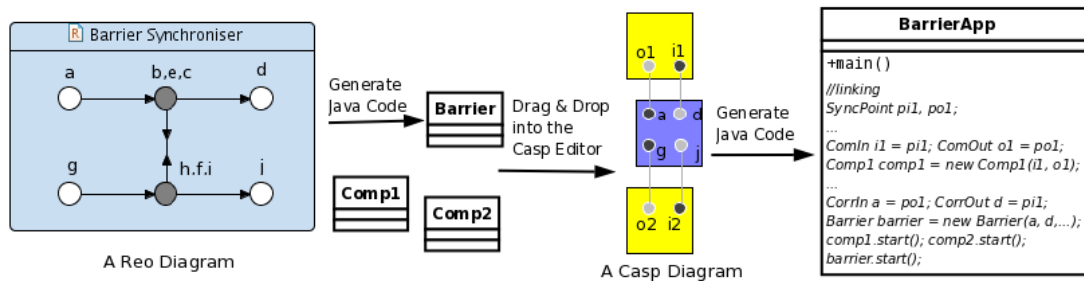


Figure 6.3: Implementing a Component-Based System with a Barrier Synchroniser

6.2 A Short Overview of 2APL

2APL (A Practical Agent Programming Language) is an agent-oriented programming language that provides two distinguished sets of programming constructs to implement both

multi-agent as well as individual agent concepts. At the multi-agent level one can specify which agents should be created and which external environments they have access to. At the individual agent level, one can specify each (to be created) agent in terms of declarative concepts such as beliefs and goals, and imperative concepts such as events and plans. 2APL multi-agent systems run on the 2APL platform², a development tool that is designed to support the implementation and execution of multi-agent systems programmed in 2APL. In this section we briefly describe those parts of the syntax and intuitive semantics of 2APL that are relevant to this paper. For a complete overview of the syntax and formal semantics of the 2APL programming language we refer to [Das08]. The specification at the multi-agent level indicates which type of agents and how many of them constitute a multi-agent system. Moreover, it indicates which external environments are involved in the multi-agent system and which agent can interact with which external environment. The syntax for multi-agent level specification is as follows:

```
agentname$_1$ : filename$_1$ N$_1$ @env$_1^1$, ..., env$_1^n$
$...$
agentname$_p$ : filename$_p$ N$_p$ @env$_p^1$, ..., env$_p^m$
```

Here $agentname_i$ is the name of the agent to be created, $filename_i$ is the file name in which the agent is specified, N_i is the number of such agents to be created (if $N_i > 1$, then the agent names will be indexed by a number), and $@env_i^j$ is the name of the environment that the agents can access and interact with. Each environment env_i^j is specified by a Java class of which one instance is created by the 2APL platform when loading the multi-agent specification. We explain later in this section how such a Java class implements an environment. Next, we will describe the relevant concepts by which a 2APL agent is specified. A 2APL agent has beliefs and goals. The beliefs of an agent represent the information the agent has (about itself and its environments). The goals of an agent specify the situation the agent wants to realise. The agent's beliefs are stored in its belief base, which is implemented as a Prolog program. The agent's goals are stored in its goal base as conjunctions of ground atoms. A 2APL agent has only goals it does not believe to have achieved. Consider, for example, the belief and goal base of an auction agent with the goal of having a bike.

```
Beliefs:
    bid(100).    step(30).    maximalBid(400).
Goals:
    have(bike)
```

In this example, the agent believes its current bid to be EUR 100 (at the first round this is its initial bid), and will increase its bid each round with EUR 30 to a maximum of EUR 400. As soon as the agent believes to have bought the bike (i.e., `have(bike)` can be derived from the agent's beliefs) this goal is removed from the goal base. To achieve its goals an agent needs to act. Actions that a 2APL agent can perform include actions to update its beliefs, actions to query its belief and goal base and external actions to interact with its environment. The belief base of the agent can be updated by the execution of a belief update

²The 2APL Platform can be downloaded at <http://cs.uu.nl/2apl>

action. Such belief updates are specified in terms of pre- and postconditions. Intuitively, an agent can execute a belief update action if the precondition of the action is derivable from its belief base. After the execution of the action, the beliefs of the agent are updated such that the postcondition of the action is derivable from the agent's belief base. The belief update `UpdateBid(R)` for updating the current bid `bid(X)` to `bid(R)`, for example, is specified as:

```
{bid(X)} UpdateBid(R) {not bid(X), bid(R)}
```

A test action can be used to test whether the agent has certain beliefs and/or goals. A test action is a conjunction of belief and goal tests. Belief test actions are of the form $B(\phi)$ in which ϕ is a (Prolog) query to the belief base. Similarly, goal test actions are of the form $G(\phi)$ in which ϕ is a query to the goal base. A goal test action $G(\phi)$ is successful if ϕ is derivable from the agent's goal base. A (belief or goal) test action can result in a substitution for variables that occur in ϕ . For example, given the above belief base, the belief test action $B(bid(X))$ is successful resulting in the substitution $X/100$. A test action can be used in a plan to (1) instantiate variables in the subsequent actions of the plan (if the test can be entailed by the agent's beliefs and goals), or (2) to block the execution of the plan (in case the test cannot be entailed by the agent's beliefs and goals). A 2APL agent performs external actions to act upon its environment. In 2APL environments are implemented as Java classes of type `Environment`. External actions that can be performed in this environment are then to be implemented as methods of this class having a predefined signature:

```
Term actionName(Term t$_1$, ..., Term t$_n$)
```

in which `Term` is the Java equivalent of 2APL terms such as constants (numbers and ids) or variables. External actions in the 2APL programming language are then of the form:

```
@env(actionName(t$_1$, ..., t$_n$), R)
```

with `actionName(t1, ..., tn)` corresponding to the signature of the Java method implementing the action, `R` is the return value that can be used to capture (a part of) the result of the action, and `env` being the unique identifier of the `Environment` object that implements the environment. The performance of an external action then boils down to invoking the method specifying the external action and binding the return value of this method to `R`.

A 2APL agent adopts plans to achieve its goals. These plans are the recipes that describe which actions the agent should perform to reach the desired situation. In particular, plans are built of basic actions and can be composed (amongst others) by a sequence operator (i.e., `;`) and a conditional choice operator. Conditional choice operators are of the form `if ϕ then π_1 else π_2` . The conditional part of these expressions (ϕ) is a conjunction of belief tests $B(\phi)$ and goal tests $G(\phi)$ that are evaluated on the agent's beliefs and goals. Such a condition thus expresses that the agent should perform plan π_1 in case ϕ can be entailed by the agent's beliefs and goals and otherwise plan π_2 . Note that the conditional part of such an expression may result a substitution that binds some variables in the π_1 part of the expression.

An agent possibly has more than one plan to reach its goals. Which plans are the best usually depends on the current situation. Planning goal rules are used to generate plans based on the agent's goals and beliefs. Such a rule is typically of the form `head \leftarrow guard |`

body. The head of the rule is a goal expression indicating whether the agent has a certain goal. The guard of the rule is a belief expression indicating whether the agent has a certain belief, and the body of the rule is the plan that can be used to achieve the goal as stated by the head. A planning goal rule can be applied if the head and guard of the rule can be entailed by the agent's beliefs and goals, respectively. As an example, consider the following planning goal rule of our auction agent:

```
have(X) <- not finished | { ... }
```

indicating that the plan between the brackets can be used to achieve the goal of having product *X* in case the agent believes the auction is not finished.

6.3 Integrating Reo Connectors into the 2APL platform

The mechanism for integrating Reo connectors into the 2APL platform is as follows. For this we consider a particular environment `reo`. The execution of any 2APL external action in the environment `reo` is a read from or a write to a given sink or source node, respectively. It is the task of the MAS programmer to setup the links between 2APL action names and Reo nodes. This should be done in a configuration class, which, in this section, we call `ReoCustom`. This class should be understood as an interface between the Reo network and 2APL agents. The MAS programmer should bear in mind that the association of an action name to a source node is to be interpreted as a write operation to the node. Similarly, the association of an action name to a sink node is to be interpreted as a read operation from the node. We take, for instance, the following setup. We assume that the MAS programmer creates a MAS file with the specification `bidder1 : bidder1.2apl @reo` and that the 2APL code for the agent `bidder1.2apl` contains the external action call `@reo (bid(100), _)`. This means that there exists a corresponding node in the Reo network. Let this node be a source node `p4`. Under these assumptions, if the MAS programmer wants to implement that `@reo (bid(100), _)` is a write on `p4`, then he or she needs to associate the action `bid` of `bidder1` to `p4`. This association is done in the `ReoCustom` configuration class by the following statement:

```
addSourceNode('bidder1', 'bid', p4)
```

where the parameters are the name of the agent, the name of the 2APL action and a source node. Similarly, the association of an action name with a sink node is done by calling `addSinkNode`. These functions are implemented in `ReoEnvironment`, a specific environment. Besides providing functions which facilitate the MAS programmer to make the associations between action names and nodes, `ReoEnvironment` has a further use as well. Please note that the `@reo` external actions have a generic execution mechanism (either a read from or a write to a given node). It follows that it is desirable that the MAS programmer is spared the trouble of implementing them (as it is the case with external actions in general). `ReoEnvironment` is designed especially to make the implementation of `@reo` external actions transparent to the MAS programmer.

We note that, so far, we have left the “wiring” up to the MAS programmer. However, we can imagine other options through which the interface is created automatically. For instance,

we could think of scripting languages, where one could even design mechanisms which support parametrised MAS files as input. It should also be possible to use the Casp editor (see Section 6.1) as such an alternative. For this, MAS files should specify for each agent its interface to the Reo network. For example, the MAS file

```
bidder1 : bidder1.2apl @reo (bid p4) (readMax p3)
```

specifies that `bidder1` can perform the external actions `bid` and `readMax` in the environment `reo`. Furthermore, it specifies that the action `bid` is associated with the node `p4`, and `readMax` with `p3`. Such an approach has the advantage that a node could be associated with more than one action.

To picture the above mechanism, we propose an auction scenario illustrating the use of Reo based coordination artifacts in a 2APL system. We assume that we have a set of agents taking part in a sealed-bid auction. Each agent has its own maximal bid and its own strategy of increasing the bid. All participants submit their initial bid at the same time, then they wait for a response with the highest bid. If they want to continue they increase the highest bid with their chosen amount, otherwise they submit a default value 0. The auction ends when all minus one of the participants submit 0. The winner is the one with a non-zero bid. Typically, the planning goal rule of such a bidder is implemented in 2APL as follows:

```
have(X) <- highestBid(H) and maximalBid(Max) and step(S) and
  bid(C) and oldBid(O) and not finished | {
  if B(Max > H + S)
  then { @reo(bid(H + S), _); UpdateBid(H + S) }
  else { @reo(bid(0), _) };
  @reo(readMax(nil), NH); UpdateHighestBid(NH);
  if B(highestBid(0) and oldBid(Y) and bid(Y))
  then Bought(X) else if B(highestBid(0)) then Finish() }
```

where `updateBid(X)`, `updateHighestBid(X)`, `Bought(X)`, `Finish()` are the internal actions of the bidder agent, defined simply as belief updates, and `bid(X)`, `readMax(X)` are the only external actions that bidders can perform in the environment `reo`. Assume that `auction.mas` is the MAS file describing two bidders, and assume that the bidding agents are implemented in `bidder1.2apl` and `bidder2.2apl`:

```
bidder1 : bidder1.2apl @reo
bidder2 : bidder2.2apl @reo
```

We implement the mechanism of the auction as a Reo connector. Whenever a bidder submits a bid, a writing to the corresponding node occurs. We ensure that the bids happen simultaneously by using the barrier synchroniser described in Section 6.1, as it can be noticed in Figure 6.4.

Adding components to a multi agent system has the advantage of making our approach more powerful, generic and modular. We advocate the use of components whenever a standard task, with a clear meaning, needs to be implemented. This is why we choose to implement the auctioneer as a component, `Max`, which basically takes the data from its input nodes and forwards the maximum value. The value computed by `Max` is broadcast to `readMax1`

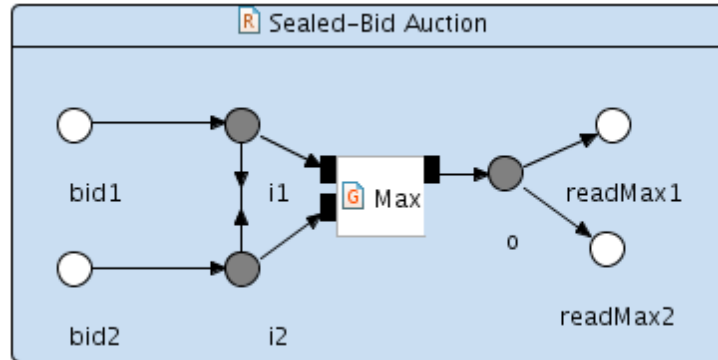


Figure 6.4: A Reo Connector Implementing an Auction

and `readMax2`, which coincide with the sink nodes associated to the action `readMax` of the bidders, thus the bidders can read the value of the highest bid and continue the auction.

Given that we generated from the Reo diagram in Figure 6.4 a Java class `Auction` by the mechanism described in Section 6.1, we can now proceed with filling in the missing information in the interface we referred to as `ReoCustom` in Section 6.3. Since it is application dependent, we name it `ReoAuction`. This is the place where we setup the links between the nodes of the coordinator and the nodes of all other components, in our case, the bidding agents and `Max`. This is partially done by generating code from the Casp diagram (see Figure 6.5). We further need to setup the associations between the external actions and the nodes of the bidders. Take, as an illustration, the function call `addSinkNode("bidder1", "readMax", p3)`, where `p3` is a synchronisation point representing, on the one hand, coordinator's source node `readMax1`, and on the other hand, `bidder1`'s sink node `readMax`. This establishes that whenever `bidder1` performs a `readMax` action it reads the data written to `readMax1`.

We assume that the first bidder has an initial bid of EUR 150, and that he is willing to increase the highest bid with the amount of EUR 10, until it reaches an upper limit of EUR 300. Similarly, we assume that the second bidder has an initial offer of EUR 100, that the increasing step is of EUR 30, and that the maximal bid is EUR 400. We also assume that both bidders have the goal of buying a bike. The implementation of the bidders' planning goal rule suggests that these are naive, as we can easily foresee the winner. The auction stops when the bidder with the smallest upper limit submits 0, in our case after `bidder1` bids EUR 290. This means that `bidder2` wins the bike for EUR 310. Running the application confirms our expectations.

For the sake of clarity, the above scenario is on purpose simple. However, we could further make it more complex. For example, we could implement a component which validates the bids submitted by the agents: here we assume that the bidders always submit a higher bid than the previous one, however, this is a particular case. It is possible that the bidders have

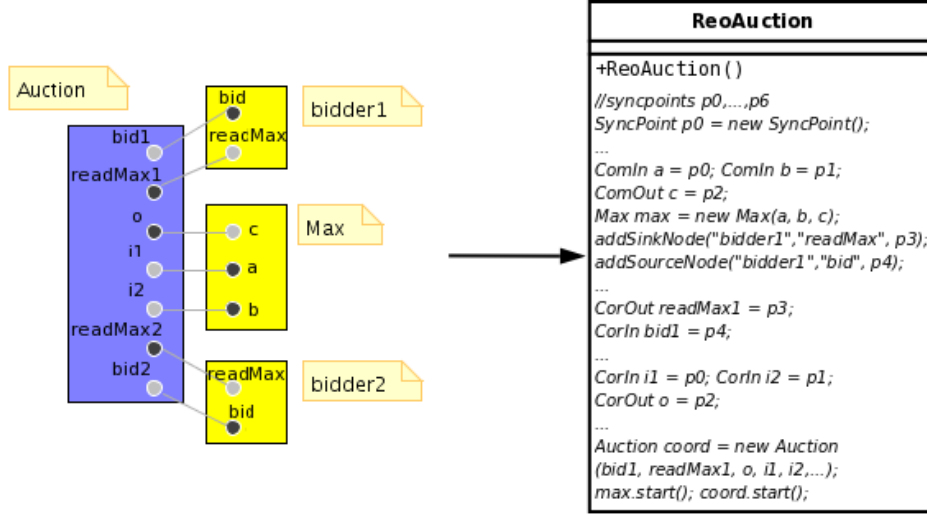


Figure 6.5: ReoAuction: The Interface between the Bidders and the Connector Auction

a different implementation, and that we might not even have access to the source code. It follows that it is desirable to impose a validation step in the Reo connector implementing the auction. Figure 6.6 shows the result of adding a validator component for `bidder1`. There, the component `Validator` simply compares the bid submitted by the agents with the previous ones. In order to record previous bids and input them to the `Validator` we basically create a new node `cache` and a `Sync` channel connecting the input of `Max` to `cache`, such that each time `bidder1` submits a valid bid this value is fed to `cache`. The node `cache` is connected to the input of `Validator` through a full `FIFO1`, which initially, at the first round contains value 0. The `SyncDrain` channel ensures that the `Validator` component can fetch data only when the bidder submits a bid. Only if the bid is greater than the value read from the `cache` is the `cache` updated by inserting the bid, otherwise not. Note that in such a situation the flow of data through the connector is stopped.

The Eclipse Coordination Tools are useful not only in designing Reo connectors but also in verifying them, as they contain an animation and a model-checker tool. The Reo animation is handy in the design phase. It helps the programmer to better understand the data flow in the Reo connector. The Reo model-checker [KB07] can be used in model-checking whether properties expressed in Branching Time Stream Logic (BTSL) are valid for the designed Reo coordination artifacts.

BTSL combines features of CTL [CES86], PDL [FL79] and time data stream logic (TDSL) [ABdBR04]. We can therefore express properties like $\forall \langle bid_1 \wedge bid_2 \rangle true$. This means that for all executions, there exists a prefix which satisfies the constraint $bid_1 \wedge bid_2$ such that it reaches a state where *true* holds. The constraint $bid_1 \wedge bid_2$ denotes that the operations on the nodes bid_1 and bid_2 happen simultaneously. The constraints in PDL-like formulae can also be defined on the data passing through nodes. A property like $\forall \langle d_{bid_1} = d_{i_1} \rangle true$

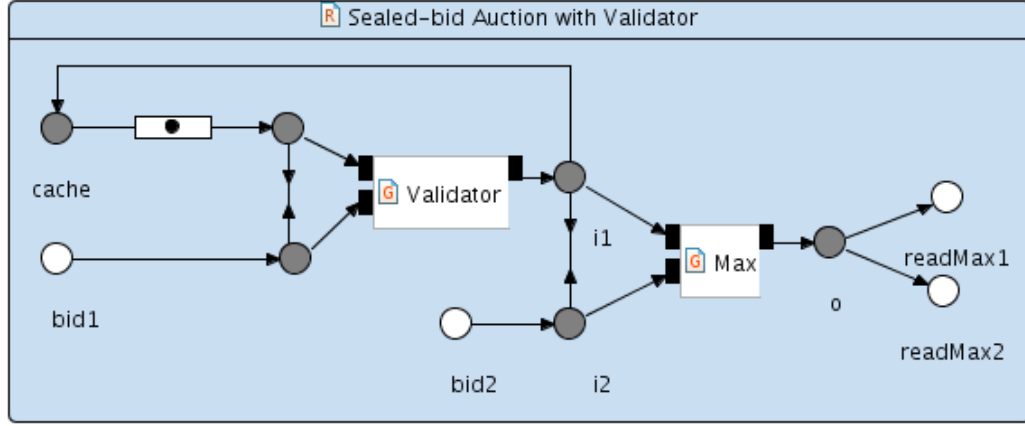


Figure 6.6: The Auction Connector with a Validator for bidder1

expresses that for all paths there exists a prefix where it holds that the data written at the node bid_1 (symbolically denoted by d_{bid_1}) is the same with the data d_{i_1} read at the node i_1 .

We can further extend our example by adding a sequencer connecting the sources and the sink of `Max`, as it can be noticed in Figure 6.7. For the sake of illustration, Figure 6.8 shows a two place sequencer. It consists of two `FIFO1` and three `Sync` channels, with the leftmost `FIFO1` being initialised with a data item (the value is irrelevant). The sequencer ensures that the read operations succeed only in the strict order from left to right. The construction is generic, one just needs to insert some more pairs of `Sync` and `FIFO1` channels in order to obtain a k -Sequencer. Given the connector described in Figure 6.7 we can model-check that it can never be the case that the component `Max` outputs a higher bid before receiving the actual value of the highest bid: $\neg \exists \langle readMax_1; bid_1 \rangle true$. The regular expression $readMax_1; bid_1$ has precisely the meaning of “an operation on $readMax_1$ is followed by an operation on bid_1 ”. As one might expect, all the properties defined above hold for the connector from Figure 6.7.

Currently, no model checking tools exist for 2APL programs. However, using Reo encourages a compositional approach to the verification of systems, where (1) the externally observable behaviour of each 2APL agent is represented by a constraint automaton; (2) the system is verified as the product of the constraint automata of its agents and Reo connectors; and (3) the compliance of each individual agent with its constraint automaton model is verified separately. We note that properties involving the effective data values (numerical) passing through the connector could be integrated by making use of `Filter` channels. Such channels have predicates as labels which enables data to pass. The predicates can typically be (in)equalities between functions defined on the values from the source (sink) nodes. In this way we could replace the `Sync` channel connecting the sink node of `Validator` to the source `cache` by a filter with the predicate $d_{bid_j} > d_{cache}$ as the corresponding label.

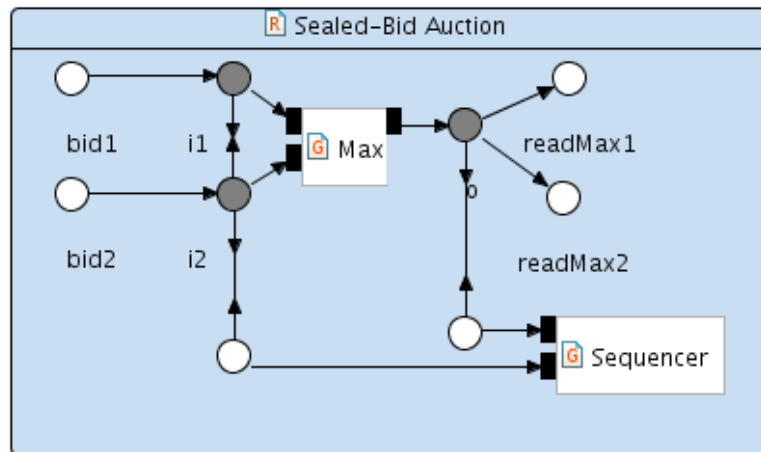


Figure 6.7: The Auction Connector with a Sequencer

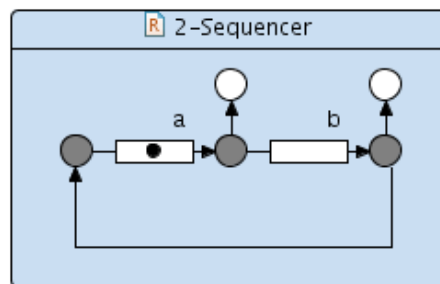


Figure 6.8: A 2-Sequencer Connector

In this thesis we presented an executable theory of multi-agent systems. In the first part of the thesis we focused upon individual agent refinement. The results can be grouped in two classes. The first class relates the design of agent languages. There, the main message is to design from abstract specifications to more concrete implementations by means of refinement. The second class relates the verification of the correctness of agent programs where an agent is “correct” with respect to a specification if it refines the specification. As a concluding picture, we provide the following overview together with possible directions for future work.

Concerning the design of agent languages, we introduced BUnity as an abstract specification language, BUpL as a more concrete agent language which refines BUnity by adding control structures, and BUnity^K as a more expressive agent language which allows to define more sophisticated queries by making use of knowledge bases. We proposed a theory of control refinement illustrated by means of BUnity and BUpL. As future work, we discuss the following aspects. With respect to BUnity^K there are two main directions. The first one is to use knowledge bases in verification for *abstraction*, under the slogan “equations are abstractions”. The second direction could explore *data* refinement between BUnity^K and BUnity. One possible idea is to define data refinement as answers’ inclusion, in the sense that one agent is more “data” concrete if the querying mechanism provides more answers.

Concerning the verification of agent programs, first we showed how to model-check refinement. We then showed that even simple agent languages as BUnity are Turing complete with the consequence that a decidable algorithm for the refinement of infinite state agents does not exist. We further presented two orthogonal approaches to the verification of infinite agent programs. One is a symbolic approach based on a weakest precondition calculus, and the other one is based on testing. As possible extensions relating verification aspects, new results may be obtained on the connection between our application of the weakest precondition calculus for invariants, induction, and strengthening. With respect to testing, further extensions concern the automatic generation of test cases.

The connection between the two classes lies in our choice to use rewriting logic as a suitable semantical and logical framework for prototyping agent languages and verification techniques. We promoted rewrite strategies for a clear separation between agent executions at object-level and control of executions at meta-level. With respect to strategies, future work may concern their usefulness and applicability in the design of agent deliberation cycles,

repair mechanisms and synthesising reconfiguration schemata.

In the second part of the thesis we extended the refinement relation to timed choreographed normative multi-agent systems. To summarise, the main messages are as follows. With respect to coordination, this can be achieved at two orthogonal levels by means of actions and norms. The action-based artifact that we proposed is represented by choreographies which enforce action synchronisation between agents. The norm-based artifact that we described forces the multi-agent system to be in a normative state. Choreographies and norms concern with separate issues, however, we shown that their integration is possible and that furthermore, coordinated multi-agent systems can be extended to timed coordinated multi-agent systems by means of timed automata.

As future work we see some directions relating the implementation of the normative mechanisms at meta-level via strategies. Two aspects which have not been discussed are termination and deadlock. With respect to termination, we note that because of “malformed” counts-as rules, e.g., recursive, the application of *vigilant* may not always terminate. It is also the case that “circularities” can lead to non-terminating *totalitarian* strategies. Thus it may be of interest to determine conditions or to find special classes of normative rules such that the termination of the strategies is guaranteed. With respect to deadlock, we recall that the current mechanism for handling the case when a regimentation rule is applicable, is to “send” the system to a deadlock state. This is not an optimal exception handling mechanism. If an agent tries to do an action which leads to the application of *reg* but it can also do a permitted action *a*, it should not be the case that the system enters a deadlock state but it constrains the agent to execute *a*. We view such mechanisms as a sort of self repairing mechanisms. Their investigation and formalisation could be a subject of future work.

The last part of the thesis was concerned with more practical aspects of, on the one hand, implementing the theory of refinement in Maude, and on the other hand, describing the integration of a more powerful coordination tool Reo in a more advanced agent platform 2APL. There are two main messages that become transparent. The first one is that Maude offers a homogeneous framework for prototyping agent languages, executing agent programs and verifying properties of the agents. The second one is that by means of Reo connectors 2APL applications are modular, distributed and multitasking.

As for future work, with respect to our Maude implementation, a short term project may focus on automatically generating Java code from the Maude agents’ prototypes. This is in the idea of providing a plugin in Eclipse which would facilitate the prototyping of agent languages through a friendly interface while ensuring the efficiency of executing Java code. The main benefit of this approach would consist in the fact that the correctness of the Java implementation follows from the correctness of the translation, thus whatever agent program we execute in Java we can be sure that it is a correct implementation of the Maude prototype. Along the same line, another aspect that might deserve attention is to make a case study on the use of Rascal [KvdSV09] for prototyping and verifying agents. Rascal is a domain specific language for source code analysis and manipulation a.k.a. meta-programming. Though we used Rascal only for some short experiments, these turned out to be exciting and compensating. The main adding to what we focused upon in this thesis is that Rascal would be a syntactic approach to verification. Furthermore, besides being a standalone application, Rascal is provided also as an Eclipse plugin and this may constitute a possible link to the above

mentioned project.

With respect to the integration of Reo and 2APL, we remark that the coordination patterns imposed by Reo connectors are not suitable for expressing organisational concepts like norms or sanctions. Thus, it may be of interest for future work to extend the 2APL platform such that conceptually different coordination artifacts are incorporated into 2APL systems. Another future project may concern scalability aspects. Currently, the number of the agents specified in a multi-agent system file is a priori fixed, and the same holds for the elements of a Reo network. However, there is current work on dynamic reconfiguration of Reo networks which could be integrated such that the network copes with agents entering and leaving a multi-agent system.

Bibliography

- [AAdB⁺08] Farhad Arbab, Lacramioara Astefanoaei, Frank S. de Boer, Mehdi Dastani, John-Jules Ch. Meyer, and Nick A. M. Tinnemeier, *Reo connectors as coordination artifacts in 2APL systems*, Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA), LNCS, Springer, 2008, pp. 42–53.
- [AB10] Lacramioara Astefanoaei and Frank S. Boer, *The refinement of multi-agent systems*, Specification and Verification of Multi-agent Systems (Mehdi Dastani, Koen V. Hindriks, and John-Jules Charles Meyer, eds.), Springer US, 2010, pp. 35–65.
- [ABdBR04] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten, *Models and temporal logics for timed component connectors*, Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM) (Washington, DC, USA), IEEE Computer Society, 2004, pp. 198–207.
- [AdB08] Lacramioara Astefanoaei and Frank S. de Boer, *Model-checking agent refinement*, Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2008, pp. 705–712.
- [AdBD09] Lacramioara Astefanoaei, Frank S. de Boer, and Mehdi Dastani, *The refinement of choreographed multi-agent systems*, Proceedings of the 9th International Workshop on Declarative Agent Languages and Technologies (DALT), LNCS, Springer, 2009, pp. 20–34.
- [AdBD10] ———, *Strategic executions of choreographed timed normative multi-agent systems*, Proceedings of the 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2010, pp. 965–972.

- [AdBvR09] Lacramioara Astefanoaei, Frank S. de Boer, and M. Birna van Riemsdijk, *Using rewrite strategies for testing BUpL agents*, Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), LNCS, Springer, 2009, pp. 143–157.
- [ADMdB09] Lacramioara Astefanoaei, Mehdi Dastani, John-Jules Meyer, and Frank S. de Boer, *On the semantics and verification of normative multi-agent systems*, Journal of Universal Computer Science (J.UCS) **15** (2009), no. 13, 2629–2652, http://www.jucs.org/jucs_15_13/on_the_semantics_and.
- [Alu99] Rajeev Alur, *Timed automata*, Proceedings of the 11th International Conference on Computer Aided Verification (CAV), LNCS, Springer, 1999, pp. 8–22.
- [Apt88] Krzysztof R. Apt, *Introduction to logic programming*, Tech. report, Austin, TX, USA, 1988.
- [Arb04] Farhad Arbab, *Reo: a channel-based coordination model for component composition*, Mathematical Structures in Computer Science **14** (2004), no. 3, 329–366.
- [ASP09] Alexander Artikis, Marek J. Sergot, and Jeremy V. Pitt, *Specifying norm-governed computational societies*, ACM Trans. Comput. Log. **10** (2009), no. 1, 1–42.
- [ATW06] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier, *Observations on determinization of büchi automata*, Theor. Comput. Sci. **363** (2006), no. 2, 224–233.
- [BBC⁺09] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmal Desai, Viviana Patti, and Munindar P. Singh, *Choice, interoperability, and conformance in interaction protocols and service choreographies*, Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2009, pp. 843–850.
- [BBL08] Patricia Bouyer, Ed Brinksma, and Kim Guldstrand Larsen, *Optimal infinite scheduling for multi-priced timed automata*, Formal Methods in System Design **32** (2008), no. 1, 3–23.
- [BBM⁺07] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, Viviana Patti, and Claudio Schifanella, *Service selection by choreography-driven matching*, Proceedings of the 2nd Workshop on Emerging Web Services Technology (WEWST), CEUR Workshop Proceedings, CEUR-WS.org, 2007.
- [BBvdT08] Guido Boella, Jan Broersen, and Leendert van der Torre, *Reasoning about constitutive norms, counts-as conditionals, institutions, deadlines and violations*, Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA), LNCS, Springer, 2008, pp. 86–97.

- [BDL⁺06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkan-
son, Paul Pettersson, Wang Yi, and Martijn Hendriks, *UPPAAL 4.0*, Pro-
ceedings of the 3rd International Conference on the Quantitative Evaluaion
of Systems (QEST), IEEE Computer Society, 2006, pp. 125–126.
- [BDM⁺98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tri-
pakakis, and Sergio Yovine, *Kronos: A model-checking tool for real-time sys-
tems*, Proceedings of the 10th International Conference on Computer Aided
Verification (CAV), LNCS, Springer, 1998, pp. 546–550.
- [BFVW06] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge,
Verifying multi-agent programs by model checking, Autonomous Agents and
Multiagent Systems **12** (2006), no. 2, 239–256.
- [BJvdM⁺06] Tibor Bosse, Catholijn M. Jonker, Lourens van der Meij, Alexei Sharpan-
skykh, and Jan Treur, *Specification and verification of dynamics in cognitive
agent models*, IAT, 2006, pp. 247–254.
- [BKdV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer (eds.), *Terese. Term
Rewriting Systems*, Cambridge Tracts in Theoretical Computer Science,
vol. 55, Cambridge University Press, 2003.
- [BM03] Roberto Bruni and José Meseguer, *Generalized rewrite theories*, ICALP’03:
Proceedings of the 30th international conference on Automata, languages
and programming (Berlin, Heidelberg), Springer-Verlag, 2003, pp. 252–266.
- [BM07] Aaron R. Bradley and Zohar Manna, *The calculus of computation: Deci-
sion procedures with applications to verification*, Springer-Verlag New York,
Inc., Secaucus, NJ, USA, 2007.
- [BM08] Aaron R. Bradley and Zohar Manna, *Property-directed incremental invari-
ant generation*, Formal Asp. Comput. **20** (2008), no. 4-5, 379–405.
- [Bör87] Egon Börger, *Unsolvable decision problems for prolog programs*, Computa-
tion Theory and Logic, LNCS, vol. 270, Springer, 1987, pp. 37–48.
- [Bra87] M. Bratman, *Intentions, plans, and practical reason*, Harvard University
Press, 1987.
- [Brz64] Janusz A. Brzozowski, *Derivatives of regular expressions*, J. ACM **11**
(1964), no. 4, 481–494.
- [BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten, *Modeling
component connectors in reo by constraint automata*, Sci. Comput. Program.
61 (2006), no. 2, 75–113.
- [BvdT07] Guido Boella and Leendert W. N. van der Torre, *A game-theoretic approach
to normative multi-agent systems*, Normative Multi-agent Systems, Dagstuhl

- Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [BvdT08] Guido Boella and Leendert van der Torre, *Substantive and procedural norms in normative multiagent systems*, J. Applied Logic **6** (2008), no. 2, 152–171.
- [BvdTV07] Guido Boella, Leendert W. N. van der Torre, and Harko Verhagen (eds.), *Normative multi-agent systems, 18.03. - 23.03.2007*, Dagstuhl Seminar Proceedings, vol. 07122, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
- [CDE⁺07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (eds.), *All about Maude - a high-performance logical framework, how to specify, program and verify systems in rewriting logic*, LNCS, vol. 4350, Springer, 2007.
- [CDE⁺09] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott, *Maude manual (version 2.4)*, 2009.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Program. Lang. Syst. **8** (1986), no. 2, 244–263.
- [Cla00a] Manuel Clavel, *Reflection in rewriting logic: Metalogical foundations and metaprogramming applications*, The University of Chicago Press, Chicago, 2000.
- [Cla00b] Manuel Clavel, *Reflection in rewriting logic: Metalogical foundations and metaprogramming applications*, CSLI Publications, Stanford, CA, USA, 2000.
- [CM88] K. Mani Chandy and Jayadev Misra, *Parallel program design: A foundation*, Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.
- [CM02] Manuel Clavel and José Meseguer, *Reflection in conditional rewriting logic*, Theor. Comput. Sci. **285** (2002), no. 2, 245–288.
- [CMP07] Manuel Clavel, José Meseguer, and Miguel Palomino, *Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic*, Theor. Comput. Sci. **373** (2007), no. 1-2, 70–91.
- [CN00] Ana Cavalcanti and David A. Naumann, *A weakest precondition semantics for refinement of object-oriented programs*, IEEE Trans. Softw. Eng. **26** (2000), no. 8, 713–728.
- [Con04] Evelyne Contejean, *A certified ac matching algorithm*, Rewriting Techniques and Applications (Vincent van Oostrom, ed.), Lecture Notes in Computer Science, vol. 3091, Springer Berlin / Heidelberg, 2004, pp. 70–84.

- [DAdB05] Mehdi Dastani, Farhad Arbab, and Frank de Boer, *Coordination and composition in multi-agent systems*, Proceedings of the 4th international joint conference on Autonomous Agents and Multiagent Systems (AAMAS) (New York, NY, USA), ACM Press, 2005, pp. 439–446.
- [Das08] Mehdi Dastani, *2APL: a practical agent programming language*, Autonomous Agents and Multiagent Systems **16** (2008), no. 3, 214–248.
- [dBHvdHM07] Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Ch. Meyer, *A verification framework for agent programming with declarative goals.*, Journal of Applied Logic **5** (2007), no. 2, 277–302.
- [DG08] Nachum Dershowitz and Yuri Gurevich, *A natural axiomatization of computability and proof of church’s thesis*, Bulletin of Symbolic Logic **14** (2008), no. 3, 299–350.
- [DGMT08] Mehdi Dastani, Davide Grossi, John-Jules Ch. Meyer, and Nick Tinnemeier, *Normative multi-agent programs and their logics*, Proceedings of the 1st Workshop on Knowledge Representation for Agents and Multi-Agent Systems (KRAMAS), LNCS, Springer, 2008, pp. 16–31.
- [Dig03] V. Dignum, *A model for organizational interaction*, Ph.D. thesis, Utrecht University, 2003.
- [Dij76] Edsger W. Dijkstra, *A discipline of programming*, Prentice-Hall, 1976.
- [DvRDM03] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer, *A programming language for cognitive agents goal directed 3apl*, Proceedings of the 1st International Workshop on Programming Multi-Agent Systems (PROMAS), LNCS, Springer, 2003, pp. 111–130.
- [EGZ09] Jörg Endrullis, Herman Geuvers, and Hans Zantema, *Degrees of undecidability in term rewriting*, Proceedings of the 23rd international Workshop Computer Science Logic (CSL), LNCS, Springer, 2009, pp. 255–270.
- [Eke02] Steven Eker, *Single elementary associative-commutative matching*, J. Autom. Reason. **28** (2002), no. 1, 35–51.
- [EMOMV07] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo, *Deduction, strategies, and rewriting*, Electronic Notes in Theoretical Computer Science (ENTCS) **174** (2007), no. 11, 3–25.
- [EMS02] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan, *The Maude LTL model checker*, Proceedings of the 4th Workshop on Rewriting Logic and its Applications (WRLA) (Fabio Gadducci and Ugo Montanari, eds.), ENTCS, vol. 71, Elsevier, 2002.

- [ERRAA04] Marc Esteve, Bruno Rosell, Juan A. Rodríguez-Aguilar, and Josep Lluís Arcos, *Ameli: An agent-based middleware for electronic institutions*, Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IEEE Computer Society, 2004, pp. 236–243.
- [FD07] Berndt Farwer and Louise Dennis, *Translating into an intermediate agent layer: A prototype in Maude*, Proceedings of Concurrency, Specification, and Programming (CS&P), 2007, pp. 168–179.
- [FGM03] Jacques Ferber, Olivier Gutknecht, and Fabien Michel, *From agents to organizations: An organizational view of multi-agent systems*, Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering (AOSE), Lecture Notes in Computer Science, Springer, 2003, pp. 214–230.
- [FL79] Michael J. Fischer and Richard E. Ladner, *Propositional dynamic logic of regular programs*, J. Comput. Syst. Sci. **18** (1979), no. 2, 194–211.
- [Flo67] Robert W. Floyd, *Assigning meanings to programs*, Proceedings of a Symposium on Applied Mathematics, Mathematical Aspects of Computer Science, vol. 19, AMS, 1967, pp. 19–31.
- [Fra86] Nissim Francez, *Fairness*, Springer-Verlag New York, Inc., New York, NY, USA, 1986.
- [Gal85] Jean H. Gallier, *Logic for computer science: foundations of automatic theorem proving*, Harper & Row Publishers, Inc., New York, NY, USA, 1985.
- [GDM07] Davide Grossi, Frank Dignum, and John-Jules Ch. Meyer, *A formal road from institutional norms to organizational structures*, Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2007, p. 89.
- [GGvdT08] Davide Grossi, Dov M. Gabbay, and Leendert van der Torre, *A normative view on the blocks world*, Proceedings of the 3rd International Workshop on Normative Multiagent Systems (NORMAS), 2008, pp. 128–142.
- [GM87] J. A. Goguen and J. Meseguer, *Models and equality for logical programming*, II and Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87: Advanced Seminar on Foundations of Innovative Software Development (New York, NY, USA), Springer-Verlag New York, Inc., 1987, pp. 1–22.
- [GR08a] Guido Governatori and Antonino Rotolo, *Changing legal systems: Abrogation and annulment part i: Revision of defeasible theories*, Proceedings of the 9th International Conference on Deontic Logic in Computer Science (DEON), LNCS, Springer, 2008, pp. 3–18.

- [GR08b] ———, *Changing legal systems: Abrogation and annulment. part ii: Temporalised defeasible logic*, Proceedings of the 3rd International Workshop on Normative Multiagent Systems (NORMAS), 2008, pp. 112–127.
- [GSAdBB05] Juan Guillen-Scholten, Farhad Arbab, Frank de Boer, and Marcello Bonsangue, *Mocha-pi, an exogenous coordination calculus based on mobile channels*, Proceedings of the 2005 ACM symposium on Applied computing (SAC) (New York, NY, USA), ACM Press, 2005, pp. 436–442.
- [GZ97] David Gelernter and Lenore D. Zuck, *On what linda is: Formal description of linda as a reactive system*, Proceedings of Coordination Languages and Models, Second International Conference (COORDINATION), LNCS, Springer, 1997, pp. 187–204.
- [HdBvdHM99] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer, *Agent programming in 3APL*, Autonomous Agents and Multi-Agent Systems **2** (1999), no. 4, 357–401.
- [HK99] Miki Hermann and Phokion G. Kolaitis, *Computational complexity of simultaneous elementary matching problems*, J. Autom. Reason. **23** (1999), no. 2, 107–136.
- [HL78] Gérard Huet and Dallas Lankford, *On the uniform halting problem for term rewriting systems*, Tech. Report 283, INRIA, 1978, Technical Report.
- [HLM⁺08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou, *Testing real-time systems using UPPAAL*, Formal Methods and Testing, LNCS, Springer, 2008, pp. 77–117.
- [HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman, *Introduction to automata theory, languages and computability*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [Hoa69] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580.
- [Hoa72] ———, *Proof of correctness of data representations*, Acta Inf. **1** (1972), 271–281.
- [HSB02] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier, *Moise+: towards a structural, functional, and deontic model for mas organization*, Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS), ACM, 2002, pp. 501–502.
- [Hue76] Gérard Huet, *Résolution d'équation dans les langages d'ordre 1, 2, ..., ω* , Ph.D. thesis, Université Paris VII (France), 1976.
- [Jac02] Bart Jacobs, *Weakest precondition reasoning for java programs with jml annotations*, Journal of Logic and Algebraic Programming **58** (2002), 2004.

- [JKR89] C. S. Jutla, E. Knapp, and J. R. Rao, *A predicate transformer approach to semantics of parallel programs*, PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing, ACM, 1989, pp. 249–263.
- [KB07] Sascha Klüppelholz and Christel Baier, *Symbolic model checking for channel-based component connectors*, Electron. Notes Theor. Comput. Sci. **175** (2007), no. 2, 19–37.
- [KK99] Claude Kirchner and Helene Kirchner, *Rewriting, solving, proving*, A preliminary version of a book available at www.loria.fr/~ckirchne/rsp.ps.gz, 1999.
- [Klo92] Jan Willem Klop, *Term rewriting systems*, Handbook of Logic in Computer Science (Samuel Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, eds.), vol. 2, Oxford University Press, Inc., New York, NY, USA, 1992, pp. 1–116.
- [KMS09] Marcus Kracht, John-Jules Ch. Meyer, and Krister Segerberg, *The logic of action*, 2009, <http://plato.stanford.edu/entries/logic-action/>.
- [KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju, *Rascal: A domain specific language for source code analysis and manipulation*, Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2009, pp. 168–177.
- [LPSS09] A. Lomuscio, W. Penczek, M. Solanki, and M. Sreter, *Runtime monitoring of contract regulated web services*, Proceedings of the 12th International Workshop on Concurrency, Specification and Programming (CS&P09), 2009, to appear.
- [MA07] Sun Meng and Farhad Arbab, *Web services choreography and orchestration in Reo and constraint automata*, Proceedings of the ACM Symposium on Applied Computing (SAC), ACM, 2007, pp. 346–353.
- [Mcc62] John McCarthy, *Towards a mathematical science of computation*, In IFIP Congress, North-Holland, 1962, pp. 21–28.
- [Mes92] José Meseguer, *Conditional rewriting logic as a unified model of concurrency*, Theor. Comput. Sci. **96** (1992), no. 1, 73–155.
- [Mes97] ———, *Membership algebra as a logical framework for equational specification*, WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques (London, UK), Springer-Verlag, 1997, pp. 18–61.
- [Mey08] Bertrand Meyer, *Seven Principles of Software Testing*, IEEE Computer **41** (2008), no. 8, 99–101.

- [Min61] Marvin L. Minsky, *Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines.*, 437–455.
- [Mis04] Jayadev Misra, *A programming model for the orchestration of web services*, Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM), IEEE Computer Society, 2004, pp. 2–11.
- [MOM00a] Narciso Martí-Oliet and José Meseguer, *Rewriting logic as a logical and semantic framework*, Electronic Notes in Theoretical Computer Science (J. Meseguer, ed.), vol. 4, Elsevier, 2000.
- [MOM00b] ———, *Rewriting logic as a logical and semantic framework*, Electronic Notes in Theoretical Computer Science (J. Meseguer, ed.), vol. 4, Elsevier Science Publishers, 2000.
- [MOMV09] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo, *A rewriting semantics for maude strategies*, Electr. Notes Theor. Comput. Sci. **238** (2009), no. 3, 227–247.
- [MP92] Zohar Manna and Amir Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag New York, Inc., New York, NY, USA, 1992.
- [MS95] David E. Muller and Paul E. Schupp, *Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of rabin, mcnaughton and safra*, Theor. Comput. Sci. **141** (1995), no. 1-2, 69–107.
- [MY60] R. McNaughton and H. Yamada, *Regular expressions and state graphs for automata*, IEEE **9** (1960), 39–47.
- [Nip89] Tobias Nipkow, *Combining matching algorithms: The regular case*, Rewriting Techniques and Applications (Nachum Dershowitz, ed.), LNCS, vol. 355, Springer, 1989, pp. 343–358.
- [NO79] Greg Nelson and Derek C. Oppen, *Simplification by cooperating decision procedures*, ACM Trans. Program. Lang. Syst. **1** (1979), no. 2, 245–257.
- [NORCR07] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *Challenges in satisfiability modulo theories*, In Proceedings of the 18th International Conference on Term Rewriting and Applications (RTA), LNCS, Springer, 2007, pp. 2–18.
- [NPT07] Duy Cu Nguyen, Anna Perini, and Paolo Tonella, *A Goal-Oriented Software Testing Methodology*, Agent Oriented Software Engineering (AOSE), 2007, pp. 58–72.
- [ÖM02] Peter Csaba Ölveczky and José Meseguer, *Specification of real-time and hybrid systems in rewriting logic*, Theor. Comput. Sci. **285** (2002), no. 2, 359–405.

- [ÖM07] Peter Csaba Ölveczky and José Meseguer, *Semantics and pragmatics of real-time maude*, Higher Order Symbol. Comput. **20** (2007), no. 1-2, 161–196.
- [ÖM08] Peter Csaba Ölveczky and José Meseguer, *The Real-Time Maude tool*, Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer, 2008, pp. 332–336.
- [Plo81] G. D. Plotkin, *A structural approach to operational semantics*, Tech. Report DAIMI FN-19, University of Aarhus, 1981.
- [PR98] Dino Pedreschi and Salvatore Ruggieri, *Weakest preconditions for pure prolog programs*, Inf. Process. Lett. **67** (1998), no. 3, 145–150.
- [RAB10] Maria Birna Riemdsijk, Lacramioara Astefanoaei, and Frank S. Boer, *Using the maude term rewriting language for agent development with formal foundations*, Specification and Verification of Multi-agent Systems (Mehdi Dastani, Koen V. Hindriks, and John-Jules Charles Meyer, eds.), Springer US, 2010, pp. 255–287.
- [Rei01] Raymond Reiter, *Knowledge in action: Logical foundations for specifying and implementing dynamical systems*, The MIT Press, Cambridge, Massachusetts, 2001.
- [RL07] Franco Raimondi and Alessio Lomuscio, *Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams*, J. Applied Logic **5** (2007), no. 2, 235–251.
- [RVO07] Alessandro Ricci, Mirko Viroli, and Andrea Omicini, *Give agents their artifacts: the A&A approach for engineering working environments in mas*, Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2007, p. 150.
- [Saf89] Shmuel Safra, *Complexity of automata on infinite objects*, Ph.D. thesis, Rehovot, Israel, 1989.
- [Sea95] John R. Searle, *The construction of social reality*, The Penguin Press, London, 1995.
- [SRM09] Traian-Florin Serbanuta, Grigore Rosu, and José Meseguer, *A rewriting logic approach to operational semantics*, Inf. Comput. **207** (2009), no. 2, 305–340.
- [SS63] John C. Shepherdson and Howard E. Sturgis, *Computability of recursive functions*, J. ACM **10** (1963), no. 2, 217–255.
- [TDM08] Nick Tinnemeier, Mehdi Dastani, and John-Jules Meyer, *Orwell’s nightmare for agents? programming multi-agent organisations*, Proceedings of the 6th

- International Workshop on Programming Multi-Agent Systems (ProMAS), LNCS, Springer, 2008, pp. 56–71.
- [TDM09] Nick A. M. Tinnemeier, Mehdi Dastani, and John-Jules Ch. Meyer, *Roles and norms for programming agent organizations*, Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2009, pp. 121–128.
- [Tur54] A. M. Turing, *Solvable and unsolvable problems*, Science News (Penguin Books) **31** (1954), no. ??, 7–23.
- [VFC05] Francesco Viganò, Nicoletta Fornara, and Marco Colombetti, *An event driven approach to norms in artificial institutions*, Proceedings of the International Workshop on Agents, Norms and Institutions for Regulated Multi-Agent Systems (ANIREM), LNCS, Springer, 2005, pp. 142–154.
- [vG90] Rob J. van Glabbeek, *The linear time-branching time spectrum (extended abstract)*, Proceedings of Theories of Concurrency: Unification and Extension (CONCUR), LNCS, Springer, 1990, pp. 278–297.
- [VMO03] Alberto Verdejo and Narciso Martí-Oliet, *Executable structural operational semantics in Maude*, Tech. report, Universidad Complutense de Madrid, Madrid, 2003.
- [vRdBDM06] M. Birna van Riemsdijk, Frank S. de Boer, Mehdi Dastani, and John-Jules Ch. Meyer, *Prototyping 3apl in the maude term rewriting language*, Proceedings of the 5th international joint conference on Autonomous Agents and Multiagent Systems (AAMAS) (New York, NY, USA), ACM Press, 2006, pp. 1279–1281.
- [vRdBDM07] M. Birna van Riemsdijk, Frank S. de Boer, Mehdi Dastani, and John-Jules Ch Meyer, *Prototyping 3APL in the Maude term rewriting language*, Proceedings of the 7th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA), LNAI, vol. 4371, 2007, pp. 95–114.
- [Woo97] M. Wooldridge, *Agent-based software engineering*, IEEE Proceedings Software Engineering **144** (1997), no. 1, 26–37.
- [WS00] Pawel T. Wojciechowski and Peter Sewell, *Nomadic pict: Language and infrastructure design for mobile agents*, IEEE Concurrency **8** (2000), no. 2, 42–52.
- [ZTP08] Zhiyong Zhang, John Thangarajah, and Lin Padgham, *Automated unit testing intelligent agents in PDT*, Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, 2008, pp. 1673–1674.

Samenvatting

Het gebruik van complexe applicaties zoals incident management, sociale simulaties, productie applicaties, elektronische veilingen, e-instellingen, en business to business applicaties is groeiende en zeer belangrijk geworden. De agent-georiënteerd methodologie is een volgende abstractie die door software ontwikkelaars gebruikt kan worden om dergelijke applicaties te modelleren en ontwikkelen. Voor alle ontwerp methodologieën geldt, in het algemeen, dat control structures in latere stadia van het ontwerp moeten worden toegevoegd in een natuurlijke top-down manier, van specificatie naar implementatie, door middel van refinement. Te veel detail in de specificaties (bij voorbeeld, ten behoeve van efficiëntie) heeft vaak negatieve gevolgen. Om D.E. Knuth te parafraseren: "Premature optimization is the root of all evil"(geciteerd in "De Unix Programming Environment" van Kernighan en Pine, blz. 91).

Het doel van dit proefschrift is het aanpassen van bestaande formele technieken voor de agent-georiënteerde methodologie ten behoeve van een executeerbare refinement theorie. De rechtvaardiging hiervoor is het leveren van agent-based software wiens correctheid reeds gegarandeerd wordt door het ontwikkelproces. Het onderliggende logische framework van de theorie die we introduceren is gebaseerd op rewriting logic. Hierdoor is onze theorie op dezelfde manier executeerbaar als rewriting logic is. De verhaallijn is als volgt. We beginnen met een motivatie voor en beschrijving van de onderdelen van bestaande agent talen die we gekozen hebben voor de representatie van zowel abstracte als concrete ontwerp-niveaus. Vervolgens stellen we een definitie van refinement tussen agenten, die geschreven zijn in deze talen, voor. Deze notie van refinement verzekert dat concrete agenten correct zijn ten opzichte van hun abstractie. Het voordeel van deze definitie is dat het de formulering van een bewijstechniek voor refinement eenvoudig maakt met behulp van de klassieke notie van simulatie. Dit maakt het mogelijk om refinement effectief te verifiëren middels model-checking. Daarnaast stellen we een weakest precondition calculus voor als een op asserties gebaseerde deductieve methode waarmee de correctheid van agenten met een oneindige state space kan worden bewezen. We generaliseren de refinement theorie naar multi-agent systemen zodanig dat concrete multi-agent systemen hun abstracties verfijnen. We zien multi-agent systemen als verzamelingen van gecoördineerde agenten, en we beschouwen coördinatie artefacten alsof ze gebaseerd zijn op acties of normatieve regels. We integreren deze twee orthogonale mechanismen voor coördinatie binnen dezelfde theorie, die we uitgebreid hebben tot een framework waarin ook tijd een rol speelt. Tot slot bespreken we implementatie aspecten.

Abstract

Complex applications such as incident management, social simulations, manufacturing applications, electronic auctions, e-institutions, and business to business applications are pervasive and important nowadays. Agent-oriented methodology is an advance in abstraction which can be used by software developers to naturally model and develop systems for such applications. In general, with respect to design methodologies, what it may be important to stress is that control structures should be added at later stages of design, in a natural top-down manner going from specifications to implementations, by refinement. Too much detail (be it for the sake of efficiency) in specifications often turns out to be harmful. To paraphrase D.E. Knuth, “Premature optimization is the root of all evil” (quoted in ‘The Unix Programming Environment’ by Kernighan and Pine, p. 91).

The aim of this thesis is to adapt formal techniques to the agent-oriented methodology into an executable theory of refinement. The justification for doing so is to provide correct agent-based software by design. The underlying logical framework of the theory we propose is based on rewriting logic, thus the theory is executable in the same sense as rewriting logic is. The storyline is as follows. We first motivate and explain constituting elements of agent languages chosen to represent both abstract and concrete levels of design. We then propose a definition of refinement between agents written in such languages. This notion of refinement ensures that concrete agents are correct with respect to the abstract ones. The advantage of the definition is that it easily leads to formulating a proof technique for refinement via the classical notion of simulation. This makes it possible to effectively verify refinement by model-checking. Additionally, we propose a weakest precondition calculus as a deductive method based on assertions which allow to prove correctness of infinite state agents. We generalise the refinement relation from single agents to multi-agent systems in order to ensure that concrete multi-agent systems refine their abstractions. We see multi-agent systems as collections of coordinated agents, and we consider coordination artefacts as being based either on actions or on normative rules. We integrate these two orthogonal coordination mechanisms within the same refinement theory extended to a timed framework. Finally, we discuss implementation aspects.

Titles in the IPA Dissertation Series since 2005

- E. Ábrahám.** *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-*. Faculty of Mathematics and Natural Sciences, UL. 2005-01
- R. Ruimerman.** *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02
- C.N. Chong.** *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03
- H. Gao.** *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04
- H.M.A. van Beek.** *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05
- M.T. Ionita.** *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06
- G. Lenzini.** *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07
- I. Kurtev.** *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08
- T. Wolle.** *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09
- O. Tveretina.** *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10
- A.M.L. Liekens.** *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11
- J. Eggermont.** *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12
- B.J. Heeren.** *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13
- G.F. Frehse.** *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14
- M.R. Mousavi.** *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15
- A. Sokolova.** *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16
- T. Gelsema.** *Effective Models for the Structure of π -Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17
- P. Zoetewij.** *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18
- J.J. Vinju.** *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19
- M.Valero Espada.** *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20
- A. Dijkstra.** *Stepping through Haskell*. Faculty of Science, UU. 2005-21

- Y.W. Law.** *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22
- E. Dolstra.** *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01
- R.J. Corin.** *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02
- P.R.A. Verbaan.** *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03
- K.L. Man and R.R.H. Schiffelers.** *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04
- M. Kyas.** *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05
- M. Hendriks.** *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06
- J. Ketema.** *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07
- C.-B. Breunesse.** *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08
- B. Markvoort.** *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09
- S.G.R. Nijssen.** *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10
- G. Russello.** *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11
- L. Cheung.** *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12
- B. Badban.** *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13
- A.J. Mooij.** *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14
- T. Krilavicius.** *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15
- M.E. Warnier.** *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16
- V. Sundramoorthy.** *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17
- B. Gebremichael.** *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18
- L.C.M. van Gool.** *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19
- C.J.F. Cremers.** *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

- J.V. Guillen Scholten.** *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21
- H.A. de Jong.** *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01
- N.K. Kavaldjiev.** *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02
- M. van Veelen.** *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03
- T.D. Vu.** *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04
- L. Brandán Briones.** *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05
- I. Loeb.** *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06
- M.W.A. Streppel.** *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07
- N. Trčka.** *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08
- R. Brinkman.** *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09
- A. van Weelden.** *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10
- J.A.R. Noppen.** *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11
- R. Boumen.** *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12
- A.J. Wijs.** *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13
- C.F.J. Lange.** *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14
- T. van der Storm.** *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15
- B.S. Graaf.** *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16
- A.H.J. Mathijssen.** *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17
- D. Jarnikov.** *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18
- M. A. Abam.** *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

- W. Pieters.** *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01
- A.L. de Groot.** *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02
- M. Bruntink.** *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03
- A.M. Marin.** *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04
- N.C.W.M. Braspenning.** *Model-based Integration and Testing of High-tech Multidisciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05
- M. Bravenboer.** *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06
- M. Torabi Dashti.** *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07
- I.S.M. de Jong.** *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08
- I. Hasuo.** *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09
- L.G.W.A. Cleophas.** *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10
- I.S. Zapreev.** *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11
- M. Farshi.** *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12
- G. Gulesir.** *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13
- F.D. Garcia.** *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14
- P. E. A. Dürr.** *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15
- E.M. Bortnik.** *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16
- R.H. Mak.** *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17
- M. van der Horst.** *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18
- C.M. Gray.** *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19
- J.R. Calamé.** *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engi-

neering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenbergh. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty

of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäuser. *Model Checking Non-deterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component*

Connectors. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04