DOCTOR OF SCIENCES

On Reasoning about Socio-Technical Systems
the Multi-Bach Coordination Model and its Workbench Anemone

Barkallah, Manel

Award date:
2025

Awarding institution:
University of Namur

On Reasoning about Socio-Technical Systems:

the Multi-Bach Coordination Model and its Workbench Anemone

Manel Barkallah

Jury

Prof. Wim Vanhoof
University of Namur, Belgium

Prof. Laura Bocchi
University of Kent, United Kingdom

Prof. Stefano Mariani
UNIMORE University, Italy

Prof. Katrien Beuls
University of Namur, Belgium

Prof. Pierre-Yves Schobbens
University of Namur, Belgium

Prof. Jean-Marie Jacquet
University of Namur, Belgium

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Science, option Computer Science

Supervised by Prof. Jean-Marie Jacquet

University of Namur
Focus - Namur Digital Institute

UNIVERSITÉ DE NAMUR
FACULTÉ D'INFORMATIQUE

Cover design: © Presses universitaires de Namur
Cover illustration: Human–AI collaboration

Rue Grandgagnage 19
B – 5000 Namur (Belgium)

Printed in Belgium.
ISBN: 978-2-39029-215-9
Registration of copyright: D/2025/1881/8

“The Most Dangerous Phrase in The Language is:
"We’ve Always Done It This Way” ”

— Grace Murray Hopper

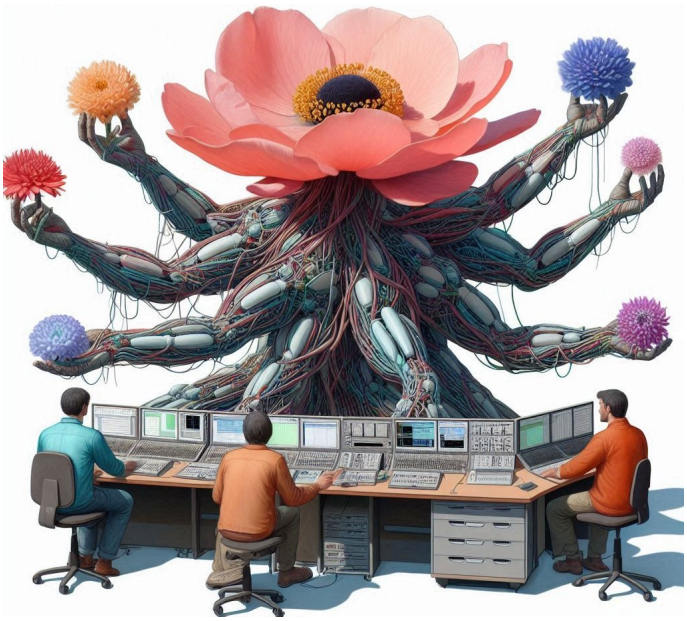
ABSTRACT

The spreading of internet-based technologies since the mid-90s has led to a paradigm shift from monolithic centralized information systems to distributed information systems based upon the composition of software components, interacting with each other and of heterogeneous natures. The popularity of these systems is nowadays such that our everyday life is touched by them.

Classically concurrent and distributed systems are coded by using the message passing paradigm according to which components exchange information by sending and receiving messages. In the aim of clearly separating computional and interactional aspects of computations, Gelernter and Carriero have proposed an alternative framework in which components interact through the availability of information placed on a shared space. Their framework has been concretized in a language called *Linda*. A series of languages, referred to nowadays as *coordination languages*, have been developed afterwards. In addition to providing a more declarative framework, such languages are well-suited for applications like Facebook, LinkedIn and Twitter, in which users share information by adding it or consulting it in a common place. Such systems are in fact particular cases of so-called *socio-technical systems* in which humans interact with machines and their environments through complex dependencies. As coordination languages nicely meet social networks, the question naturally arises whether they can also nicely code socio-technical systems. However, answering this question first requires examining how well programs written in coordination languages can reflect what they are assumed to model.

This thesis aims to address these two questions. To that end, we will use the *Bach* coordination language developed at the University of Namur as a representative of Linda-like languages. We will extend it into a language named *Multi-Bach* to enable the coding and reasoning about *socio-technical systems*. We will also introduce a workbench, *Anemone*, to support the modelling of such systems. Finally, we will demonstrate the interest of our approach through the coding of several socio-technical systems.

Keywords: Coordination, Bach, Anemone, Socio-Technical Systems



The above picture has been generated by the artificial intelligence tool "Microsoft Bing" to illustrate our work. Concurrent information systems are depicted by various computers driven by three persons. This shows that computer systems not only help humans in their tasks but also impact their decisions and consequently their behaviours. Hence, the introduction of socio-technical systems in which both humans and machines interact on an equal basis and impact each other. Moreover the computers share a common place, which reminds the shared space used by coordination languages. Finally, the Anemone flower is depicted in a central and important position in the aim of linking and understanding all the components, as we aim to achieve on our workbench.

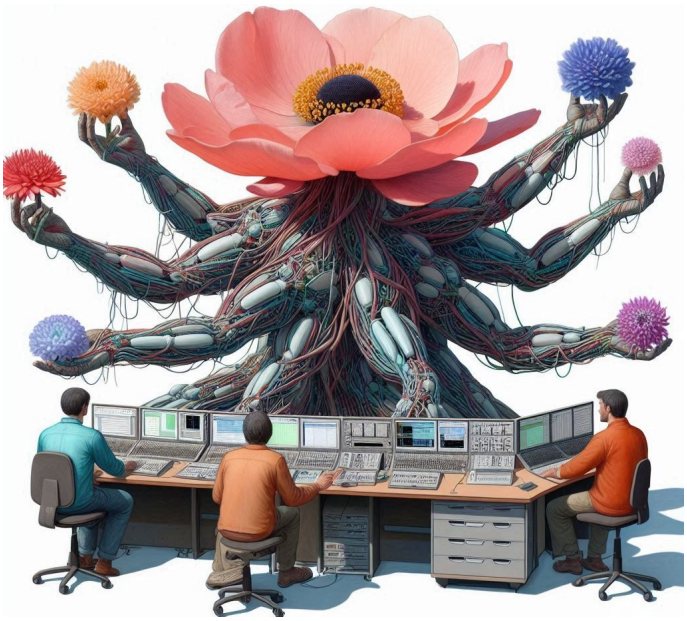
RÉSUMÉ

Depuis le milieu des années 90, le déploiement des technologies de sur l'internet a conduit à un changement de paradigme, faisant passer les systèmes d'information de logiciels monolithiques et centralisés à des systèmes distribués obtenus par composition de fragments hétérogènes de logiciels interagissant les uns avec les autres. La popularité de ces systèmes est aujourd'hui telle que notre vie quotidienne en est grandement impactée.

Traditionnellement, les systèmes concurrents et distribués sont programmés selon le paradigme de passage de messages, où les composants coopèrent en s'échangeant des messages. Pour mieux séparer les aspects de calcul pur et d'interaction, Gelernter et Carriero ont proposé une approche différente, où les composants coopèrent au travers d'un espace commun d'informations. Cette idée a donné naissance à un langage appelé *Linda*. Plusieurs langages appelés « *langages de coordination* » ont été développés à la suite. Ces langages offrent un cadre plus déclaratif et s'adaptent bien à des applications comme Facebook, LinkedIn ou Twitter, où les utilisateurs partagent et consultent des informations dans un espace commun. Ces systèmes sont en réalité des exemples typiques de ce qu'on appelle les systèmes socio-techniques, où humains, machines et environnements interagissent selon des relations complexes. Comme les langages de coordination sont bien adaptés aux réseaux sociaux, une question se pose naturellement : peuvent-ils aussi bien représenter les *systèmes socio-techniques* ? Pour répondre à cette question, il convient d'abord évaluer dans quelle mesure ces langages peuvent modéliser fidèlement ce qu'ils prétendent représenter.

Notre thèse a pour but d'explorer ces questions. Pour ce faire, nous utiliserons le langage de coordination *Bach*, développé à l'Université de Namur, comme exemple représentatif des langages développés à la suite de Linda. Nous l'étendrons en créant un langage nommé *Multi-Bach* pour coder et raisonner sur les *systèmes socio-techniques*. Nous introduirons aussi un atelier, *Anemone*, pour faciliter la modélisation de ces systèmes. Enfin, nous montrerons l'intérêt de notre approche en codant differents exemples de systèmes socio-techniques.

Mots-clés : Coordination, Bach, Anemone, Systèmes Socio-Techniques



L'image ci-dessus a été générée par l'outil d'intelligence artificielle "Microsoft Bing" pour illustrer notre travail. Des systèmes d'information concurrents sont représentés par divers ordinateurs pilotés par trois personnes. L'objectif est de suggérer que les systèmes informatiques ne se contentent pas d'aider les humains dans leurs tâches, mais qu'ils influencent également leurs décisions et, par conséquent, leurs comportements. D'où l'introduction de systèmes socio-techniques dans lesquels les humains et les machines interagissent sur un pied d'égalité et ont un impact les uns sur les autres. De plus, les ordinateurs partagent un lieu commun, ce qui rappelle l'espace partagé utilisé par les langages de coordination. Enfin, la fleur d'anémone est représentée dans une position centrale dans le but de relier et de comprendre tous les composants, comme nous souhaitons le faire avec notre atelier Anemone.

ACKNOWLEDGEMENTS

The journey of completing this thesis has been a long one, filled with challenges, learning experiences, and valuable encounters. Today, it is essential for me to express my deepest gratitude to all those who, directly or indirectly, contributed to this adventure.

First and foremost, I would like to thank the first person who believed in me and supported me tirelessly throughout these years: my supervisor, Prof. Jean-Marie Jacquet. For six years, his rigorous and thoughtful guidance, academically and within the faculty and university, has been a cornerstone of my journey. His steady presence and outstanding mentorship provided not only scientific direction but also invaluable personal support. I am profoundly grateful for the opportunity he gave me, and for allowing me to grow both professionally and personally.

I extend my sincere thanks to the members of my advisory committee: our Dean, Prof. Wim Vanhoof, Prof. Stefano Mariani, and Prof. Pierre-Yves Schobbens. Their thoughtful feedback, enriching discussions, and insightful ideas significantly shaped and deepened my reflections throughout the development of this thesis. I truly appreciated their availability, kindness, and commitment. I would also like to thank the two women who joined my thesis jury: Prof. Katrien Beuls and Prof. Laura Bocchi. Both are inspiring figures in their respective fields and have personally served as role models to me. I had the chance to engage in enriching conversations with them related to my thesis topic, which broadened my perspective and opened up new avenues for reflection. Their presence, thoughtful insights, and encouragement were truly valuable and motivating.

My heartfelt thanks also go to our Rector, Prof. Annick Castiaux, and the Vice-Rectors — Prof. Laurent Schumacher (Education and Sustainable Development), Prof. Carine Michiels (Research and Libraries), and Prof. François-Xavier Fievez (Social and Student Affairs, Gender, Sports, and Culture). I had the honour of collaborating with them, especially within the "Women and Girls in Science – UNamur" committee. Their openness and dedication played a key role in the success of many initiatives that benefited both the faculty and the university.

My deepest gratitude also goes to my family, my unwavering foundation.

To my parents, Taher and Zakia, my sister Fatma, my brother Mohamed Ayoub: I owe you so much. Mom, Dad — it is thanks to your unconditional love, your patience, and the silent sacrifices you made for me, my sister, and my brother that I was able to follow this path. You've always encouraged me to believe in myself, to dream big, and to keep going, even in moments of doubt. Your emotional and moral support has been my greatest strength. Without it, without the values and resilience you instilled in me, I simply wouldn't be here today. I truly hope you are proud of me, just as you have always said — because I am deeply proud to be your daughter. To my cousins Narjes, Imen and Asma Barkallah, and all the Barkallah family who have always supported me with so much love and encouragement; to my grandparents Mabrouka and Said Barkallah, whose guidance and wisdom have always inspired me; and to my late grandparents Fatma and Salem Barkallah, I haven't forgotten you and never will. Thank you for your unconditional love, your incredible strength, and the lasting legacy you left behind, which continues to guide and inspire me every single day. I also want to thank my best friend, Asma Bouchoucha, for all the love and support she has given me over years and years, and for always pushing me forward with warmth and faith.

A huge thank you to my second family, the one I found here in Belgium.

My colleagues at the Faculty of Computer Science became so much more than coworkers; they truly became like family. Together, we went through intense and sometimes difficult times, especially during the Covid period, but also shared many joyful moments, laughter, and unforgettable memories, often over a coffee or a pizza: Babette Di Guardia, Xavier Devroey, Benoît Vanderose, Gilles Perrouin, Denis Darquennes, Karin Derochette, Gonzague Yernaux, Guillaume Nguyen, Sophie Fortz, Paul Temple, Moussa Armani, Yasmine Akaichi, Liesbet De Vos, Doha Ouardi, Mikel Vandeloise, Lucas Berg, Sereysethy Touch, Charline Dardenne, Sacha Corbugy, Jerome Fink, Martin Belfroid, Jules Dejaeghere, Hosam Elkoulak, Lionel Goffaux , Antoine[4]: A. Clarinval, A. Sion, A. Hubermont and A. Gratia, and all the amazing people in the administration, the professors I have shared classes (and even just a coffee break) with, my students, research teams, and all the UNamur faculty staff: thank you for your kindness, your friendship, and for always being there.

And when I speak of family, I also think of the one I built through the various groups and committees at the university: the International Researchers Group, and my wonderful Belgian family: Clarice Colleen Manuel, Pierre Laurent, Julia Bagatin, Kawther Idouz and Julian, Cilya Oulmas and Renaud, Tarcus and Tais Ramos, Sophie Ayama Canden, Rohan Arora, Aishwarya Saxena, as well as the "Women and Girls in Science – UNamur" committee, and many others. To all the friends from around the world who crossed my path and left their mark on this journey, thank you for making my time at UNamur so rich, meaningful, and truly human.

To all of you, thank you, from the bottom of my heart♡ !

CONTENTS

List of Figures

LIST OF TABLES

Preface

PREFACE

Distributed systems constitute a hidden infrastructure powering many services ranging from cloud storage to online gaming. Whether you are streaming a video, making a bank transaction, or using a ride-sharing app, distributed systems allow for seamless coordination and data exchange between devices across vast networks. Their ubiquity makes them essential for supporting the demands of today's interconnected digital world.

It is however not the first time in the history of computer science that such a synergy between technological artefacts and humans is put forward. In 1981, E. Trist already identified *socio-technical systems (STS)* as systems involving complex dependencies between humans, machines and the environment within which they operate [118]. B. Whitworth further claims that STS are invoked each time social interaction is mediated by information technology rather than by the natural world [126]. In view of what precedes, one may even claim that our modern society is ruled by collections of STS.

As evidenced in [80], *engineering STS* is difficult for two main reasons. On the one hand, STS should be built with special concerns for interaction. On the other hand, STS has to marry two different worlds, the social and technical worlds, driven by different concerns. Following [81], we think that these two challenges may be faced by properly handling contexts in *coordination languages*. Indeed, by clearly separating the interactional and computational aspects of software components, they have been proved to be well suited to program the interaction of conventional distributed systems, and in particular to model service-oriented applications (see eg [26, 68, 116]). Their expressiveness [28, 43, 25] is such that it is to be expected that they also offer suitable means to handle other forms of socio/technical interactions mentioned above.

In the aim of introducing our work, we subsequently present what coordination languages and models are as well as what socio-technical systems are. We then state our thesis and describe how it is substantiated in this manuscript. Finally, we list and comment the publications on which this work rest.

Context

Coordination Languages and Models

As exemplified by the Erlang language [124] and the Akka Actor model [3], message passing is a very popular framework to code concurrent and distributed applications. However, in the aim of building interactive distributed systems, a clear separation between the interactional and the computational aspects of software components has been advocated by Gelernter and Carriero in [54]. Their claim has been supported by the design of a model, Linda ([34]), originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace. A number of other models, now referred to as coordination models, have been proposed afterwards. At the University of Namur, Prof. Jacquet's team has contributed to that trend of research in many directions: language design, implementations, expressiveness and semantical studies, and programming methodologies (see e.g. [29, 64, 28, 74, 76, 75, 66, 41, 44, 42, 43]). In particular, the Bach language has been proposed as a variant of the Linda language. More generally, according to [95, 3], two main categories of coordination models can be distinguished:

- **Channel-Based Models:** Using communication channels for message exchange between processes, as seen in the CSP (Communicating Sequential Processes) model.
- **Data-Driven Models:** Utilizing tuple spaces or shared data structures, such as in the Linda model, to coordinate activities among distributed processes.

These languages are part of the center stage in various domains including distributed systems, embedded systems, distributed computing, and web services, where efficient coordination of processes is essential for ensuring system performance and reliability.

In summary, coordination languages can be seen as specialized tools for modelling and managing complex interactions between components within a computational system. In contrast to traditional programming languages that focus on computation and data manipulation, coordination languages concentrate on scheduling, synchronization, and communication among processes or actors within a system.

Socio-Technical Systems

Socio-technical systems are defined as systems where technical subsystems (machines, tools, software) and social subsystems (people, organizational structures, processes) closely and interdependently interact. Originating from the work of the Tavistock Institute of Human Relations in the 1950s [59], this concept emphasizes the need to jointly design and manage both technical and social components.

Researchers like Eric Trist and Ken Bamforth highlighted that enhancing work system performance requires [118], addressing both technical efficiency and social dynamics within organizations. The socio-technical approach promotes key principles to enhance these systems:

- **Autonomy of work groups**: Granting teams autonomy in managing their processes, fostering innovation and accountability.
- **Employee participation in decision-making:** Actively involving organizational members in strategic and operational decisions to enhance engagement and satisfaction.
- **Adaptation of technologies to human needs:** Designing technologies that meet users' needs and capabilities while enhancing efficiency and workplace safety.

These principles are crucial across various domains such as work management, software development, workplace health and safety, and organizational innovation. By seamlessly integrating social and technical aspects, socio-technical systems aim to optimize overall performance while improving the well-being of individuals within organizations.

As we shall see later, it turns out that data-driven coordination languages are well-suited for coding distributed applications, enabling processes to synchronize through publishing, retrieving, and consulting information. This offers benefits like spatial and temporal decoupling, as well as distributed sharing. In distributed systems with a social dimension, known as socio-technical systems, it is crucial to combine technical and social elements. These systems need interaction mechanisms that bridge the gap between socially expressed objectives and the technical actions required. While coordination languages seem suitable for these tasks, it is essential not only to develop mechanisms for coding these systems but also to provide ways to model and reason about them to build trust, especially in the context of the trusted artificial intelligence. To address this, the thesis proposes an extension of a process algebra (based on the Bach language, an extension of the Linda model) that is expressive enough to model both distributed and socio-technical systems, along with a working environment for reasoning about these models.

Thesis Statement

Coordination languages can be extended to **encode** and **reason** about socio-technical systems. By encoding we refer to the translation of socio-technical systems in programs written in coordination languages. By reasoning we do not mean to simulate intelligent behaviours or to make inferences, as Artificial Intelligence would suggest, but rather we address the process of thinking on programs in order to make sure that they correctly represent the socio-technical systems under study.

This manuscript contributes to the study of coordination languages and socio-technical systems in two key ways. First, it proposes an enriched process algebra based on the Bach coordination language, which captures key concepts while ab-

stracting from implementation details. This framework is expressive enough to model both distributed and socio-technical systems, where technical and social aspects are closely intertwined. Second, it introduces a workbench to reason about models written in Bach and its extension. This workbench allows users to understand instructions by executing them step by step, to better grasp the modelling of real-life systems through animations, and to verify system properties through the verification of reachability formulas. This leads us to our two thesis claims:

Claim 1. *Enriching the Bach coordination language with primitives for dealing with processes as active data, for relating data and for animating computations creates a model suitable to code socio-technical systems.*

Claim 2. *A workbench that allows (i) to execute coordination programs step by step, (ii) to animate computations by graphical means and (iii) to check properties by verifying formulae is a good tool to reason about socio-technical systems.*

Thesis Outline

This thesis is structured into five main parts. **Part I** serves as an introduction to the background material and encompasses two chapters. Chapter 1 illuminates the shift from distributed systems to socio-technical systems in the realm of coordination languages. Chapter 2 provides an overview of the Bach coordination language, the foundation of this thesis, and includes an example of an intelligent restaurant to model the language's concepts.

As shown in Figure 1, **Part II** introduces and explores extensions to this language and consists of two chapters. Chapter 3 defines our initial language, Anim-Bach, and introduces the Scan workbench to showcase the language's utility. The primary feature of Anim-Bach is its capability to help users understand instructions, connect agents to animations for system evolution modelling, and facilitate property verification through model verification and trace generation. Chapter 4 introduces our second language, Multi-Bach, along with its associated workbench, Anemone. Multi-Bach extends Anim-Bach by introducing variables, generalized choices, primitives for processing processes as active data, manipulation of blackboard rules, and a revised scene description allowing for multiple scenes, incorporating widgets as key elements. Anemone, as a workbench, has been refined to accommodate these new features and provide additional facilities.

In **Part III**, we present potential optimizations for our Multi-Bach coordination model, adapting the Anemone workbench to accommodate these extensions. The first chapter of this part, Chapter 5, explores the guarded list concept in Multi-Bach, addressing the state space explosion problem in Anemone. The second chapter, Chapter 6, discusses the challenges of correctness verification of programs in data-driven coordination languages by introducing a guarded list construction to mitigate state-space explosion, improve performance and enhance the expressiveness of these languages. In **Part IV**, we show how our approach can be applied to the modelling and coordination of Socio-Technical Systems. Chapter 7 presents

a methodology for encoding socio-technical systems using the Multi-Bach coordination language and the Anemone workbench introduced in previous chapters. Chapter 8 demonstrates the modelling and application of our Multi-Bach coordination model using the Anemone workbench on a range of socio-technical systems, including games, intelligent systems, IoT, and even computer security. In **Part V**, the final part, we draw our conclusions and outline future work.

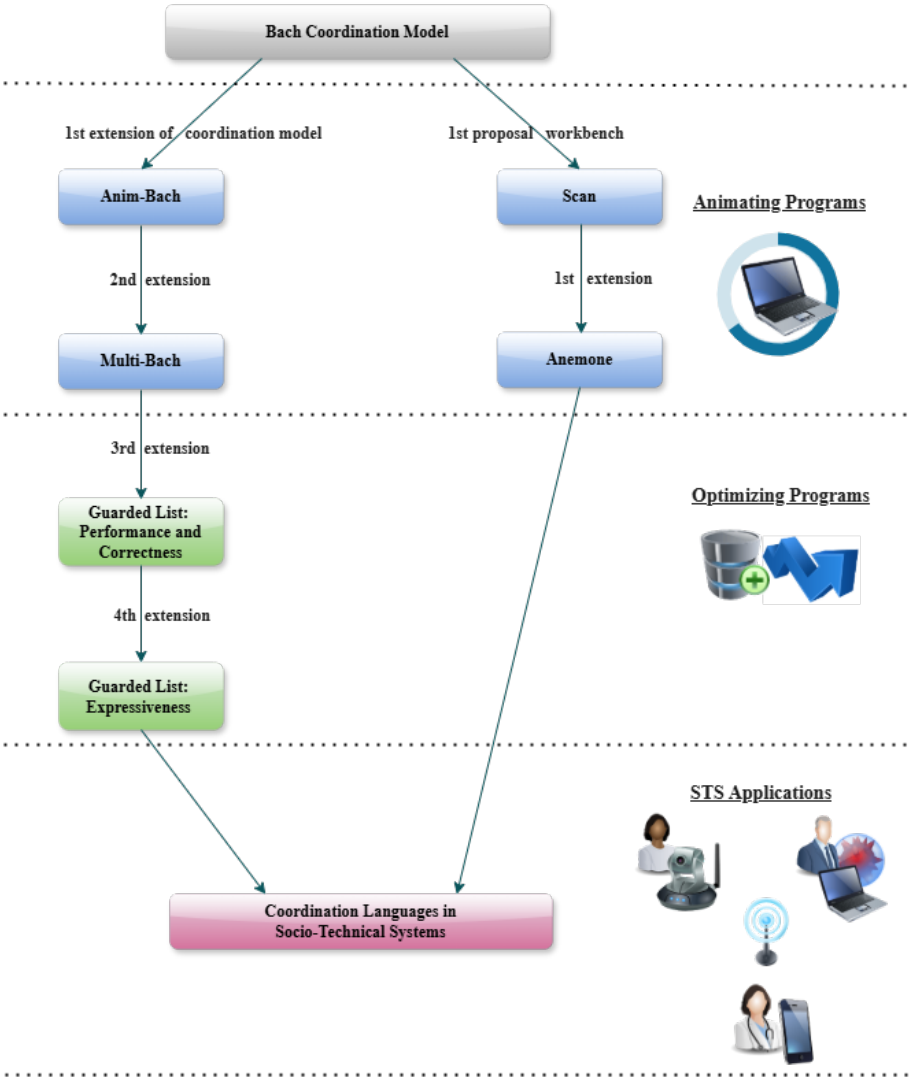


Figure 1: Thesis Outline Diagram

Publications

Parts of this thesis have been published and have been reviewed by international experts.

Conferences and Workshops

Scan: A Simple Coordination Workbench

Jean-Marie Jacquet and Manel Barkallah. Scan *A Simple Coordination Workbench* In Proceedings of the 21st International Conference on Coordination Models and Languages. Vol. 11533. Lecture Notes in Computer Science. Springer, 2019, pp 75–91 (cit. on pp. 18, 19, 21) [63].

In this paper, we propose a workbench that describes concurrent systems using the Bach language, an extension of Linda, to animate them and to reason about them using a linear temporal logic fragment. The key feature is that it maintains a direct relationship between what is written by the user and its internal representation in the workbench. Another feature is the production of replayable examples of traces when LTL formulas are satisfied.

On the Introduction of Guarded Lists in Bach: Expressiveness, Correctness, and Efficiency Issues

Manel Barkallah and Jean-Marie Jacquet. *On the Introduction of Guarded Lists in* Bach *Expressiveness, Correctness, and Efficiency Issues.* In Proceedings of the 16th Workshop on Interaction and Concurrency Experience (ICE'23), Electronic Proceedings in Theoretical Computer Science, volume 383, pp. 55–72, 2023 (cit. on p. 19) [16].

In the aim of improving the performance of the Scan model-checker, we introduce, in this paper, a new mechanism called "guarded list". We show that it does not only improves the performance of the model-checker but also that it increases the expressiveness of the resulting coordination language.

B2Scala tool: Integrating Bach in Scala with Security in Mind

Doha Ouardi, Manel Barkallah and Jean-Marie Jacquet. *B2Scala tool: Integrating Bach in Scala with Security in Mind.* In Proceedings of the 17th Workshop on Interaction and Concurrency Experience (ICE'24), to appear in Electronic Proceedings in Theoretical Computer Science, 2024 [94].

This article proposes an alternative approach to verifying security protocols using asynchronous communication via a shared space. It embodies Bach as a Domain Specific Language (DSL) in Scala, enabling programs to be tested while taking advantage of the Scala ecosystem. A variant of Hennessy-Milner logic is used to restrict executions to programs that meet certain conditions. The Needham-Schroeder protocol is used to illustrate the interest of our approach. In particular

we have been able to rediscover the “man-in-the-middle” attack on this protocol initially discovered by G. Lowe.

Journals

Anemone: A workbench for the Multi-Bach Coordination Language

Jean-Marie Jacquet and Manel Barkallah. *Anemone: A workbench for the* Multi-Bach *coordination language* In Science of Computer Programming (2021), p. 102579 (cit. on pp. 20, 21) [62].

In this article, we propose an extension of the [63] paper. This Anemone model proposes a workbench to describe concurrent systems using a Linda-like language, to animate them and to reason about them using a reachability logic. It is based, in particular, on the construction of programming environments to analyse and reason about programs written in these languages.

On the Expressiveness and Efficiency of Guarded Lists in Bach

Manel Barkallah and Jean-Marie Jacquet. *On the Expressiveness and Efficiency of Guarded Lists in Bach.* Journal of Logical and Algebraic Methods in Programming (2025): 101017 [15].

This paper is an extension of the article [16]. It expands it by adding program transformations techniques, by adding corresponding efficiency results and by investigating in a deeper manner the expressiveness of guarded lists.

Part I

Background

CHAPTER 1

FROM DISTRIBUTED SYSTEMS TO SOCIO-TECHNICAL SYSTEMS

Distributed systems are present in our daily life. Although they are very often studied from a computer science viewpoint, they often involve humans and thus lead to more general systems, called socio-technical systems. It turns out that they can be nicely modelled by the Linda model introduced by D. Gelernter [53], and more generally by the so-called coordination models developed afterwards. This chapter aims at introducing coordination models and socio-technical systems.

1.1 Opening

With the constant evolution of communication technologies, distributed systems have become ubiquitous. They are for instance central to connecting to a bank account, booking a flight, or even participating in vaccination campaigns. From a more scientific point of view, distributed programming is also a key field of software engineering.

A *Distributed System* is a system where the components are located on different computers in a network. In order to achieve a common goal, these components communicate and coordinate their actions in general by sending messages to each other. The *Distributed System* operates as a single entity with high performance, thanks to these three key properties. First, the unlimited horizontal scalability: where machines can be added whenever necessary. Second, the low latency: having machines geographically located closer to the users, will reduce the time needed to serve the users. Third, the fault tolerance: if one server (or data center) fails, the other servers can still serve the service users. With such nice properties,

it is not very surprising that distributed systems are used widely. To start with several industries having to face real-time constraints use distributed systems, both locally and globally. The "*Uber*" application is an interesting example. It divides the map-data into small grid units (e.g. 4 km) and gives a unique identifier to each unit. This is a way to distribute data in the real-time distributed system [109] and store it easily. *Distributed Artificial Intelligence (DAI)* is a second field of application of distributed systems. An illustration is provided by the traffic management of autonomous or semi-autonomous vehicles in complex transport networks. More generally, so-called *multi-agent systems* aim at making independent entities interact with themselves and their environment in a smart way. They allow for applications that bring speedy distribution, optimized storage, complex planning, and decisions.

Distributed programming has received considerable attention in the literature. Among others, many languages have been presented. Most of the time, however, coordinating the distributed entities is achieved by message passing. Erlang [124] and Actor-based models, such as Akka [127], are seminal examples. However, so-called *generative communication*, promoted by Gelernter ([53]) is also very suited to that end and will be central to our thesis.

We will present it in the following, and in particular, will highlight its characteristics and its incarceration in Linda. Then, we will present how such ideas lead to synchronization based on the information and will relate it to constraint-based concurrent programming [110] and socio-technical systems. Given that background and before giving the structure of this thesis, we present the purpose of this work. The following chapter focuses on the core of this thesis, the Bach language.

1.2 Generative Communication

As observed in [53], quoting [6], the three major mechanisms of concurrent programming are (i) monitors or shared variables, (ii) message passing, and (iii) remote operations. By proposing a shared space accessed by a few primitives, Gelernter has proposed a fourth mechanism, which he called *generative communication*. To grasp it, one needs to understand the two key notions on which it relies: on the one hand, *the tuple space* and, on the other hand, *the primitives*.

Tuples and the tuple space. Data to be communicated are represented in the form of tuples and are stored in a shared space, called the tuple space, subsequently also referred to as *TS*.

The primitives. A few primitives are introduced to allow processes to produce, consume, and observe tuples stored in tuple space. They are: "*out()*" to add a tuple to "*TS*", "*in()*" to remove one tuple, and "*read()*" to read one tuple without removing it. Although they are of a very limited number, these primitives are at the core of the

Linda coordination language, which has been shown to be of great expressiveness to code distributed applications.

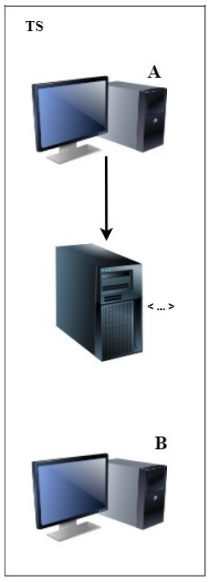


Figure 1.1: A generates a tuple, to send to the process B

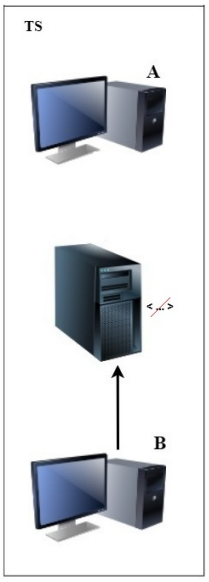


Figure 1.2: B deletes the tuple from TS

Figures 1.1 and 1.2 illustrate generative communication. Let us assume that we want to code the equivalent of a process A producing data d to be given to process B. One would traditionally have A send d to B in the form of a message. Instead, here A generates d, encapsulates it in a tuple, and puts it on the tuple space by means of the out primitive. This is what is depicted in Figure 1.1. Then process B takes data d by using the in primitive. It is worth noting from this example that, once it has been put on TS by A and until it is taken by B, the tuple associated with d has an existence independent of any process. It could for instance be read in the meantime by any other process C. This is the reason why this form of communication has been called *generative*.

Although it was introduced in 1985, much before social networks, it is worth observing that generative communication is well suited to model such networks. For instance, consider *Bob* posting a message on *Alice*'s wall on Facebook. This message will be seen by Alice's friends or even, if the settings allow for that, by any user of Facebook. If she does not like it, Alice may also remove Bob's message. This situation is in fact identical to what is described for processes A, B and C above, A playing the role of Bob, B the role of Alice and C the role of any other user of Facebook. Obviously, this comparison extends also to other social networks like LinkedIn. More generally, our view of modern information systems is that the synchronization of agents is more and more based on the availability of information instead of messages being sent and received. As an evidence, although the appli-

cation WhatsApp was first built to exchange messages from one user to another, discussion lists have been introduced to share information between members of groups of participants.

This new approach to concurrency will allow us to have a fresh look at socio-technical later in this chapter. From a scientific point of view, it is also worth observing that a similar mechanism has been independently introduced by Saraswat in order to cope with *concurrency in constraint programming* [111]. In his framework, concurrently executing agents communicate by placing and checking constraints in a shared space called a store. The two major primitives to control parallel agents are, on the one hand, atomic tell, thanks to which an agent instantaneously places constraints provided they are consistent with constraints that have already been placed, and, on the other hand, blocking ask, thanks to which an agent must block when it checks a constraint that is not yet known to hold. Although less well known than message passing, generative communication thus looks like a promising paradigm for coding distributed applications. In fact, *Linda, the first language* developed along this idea has led to a whole family of language proposals, nowadays referred to as coordination languages and models, and has since the first international conference on the subject in 1996 generated a whole body of research. In the following section, we will present Linda and highlight the nice properties of generative communication.

Linda, the first coordination language

Designed as a *distributed coordination language* [53, 54], *Linda* incorporates *generative communication.* As such, one of the important characteristics of the Linda model is the notion of the tuple, which is an ordered sequence of data.

They are located in the shared tuple space and facilitate asynchronous communication between the agents. A producer agent sends tuples and collects them in the tuple space. On the other hand, a consumer agent can read or re-read them, or retrieve them from the tuple space. The communication between agents thus arises from tuples being shared in a commonplace. As a result, each agent is able to communicate in a broadcast manner with all the other agents, in contrast to the message-passing paradigm in which one producer communicates to one consumer. Another feature of Linda is that it relies on a limited set of primitives:

- **“out”:** to enable processes to produce information, by inserting tuples in the tuple space.
- **“in”:** to enable processes to consume and retrieve information from the tuple space.
- **“rd”:** to enable processes to read information from the tuple space.
- **“eval”:** to enable processes to compute results. To that end, the function given as an argument of eval is evaluated by a parallel process and the result is put as a tuple on the tuple space, yielding thereby a passive tuple.

Finally, as announced previously, Linda features generative communication and consequently benefits from two fundamental properties: *communication orthogo-*

nality and *free naming*. Communication, in Linda, is *orthogonal* in the sense that the receiver has no prior knowledge of the sender, and similarly, the sender has no information about the receiver. Two important consequences are *space uncoupling* and *time uncoupling*. Moreover, *distributed sharing* in space and time is a consequence of the first two [53].

(i) *Space uncoupling*. Message-based programming languages generally allow a reception statement to accept data originating from any address space. Linda allows in addition a sender to send data to any address space. This makes Linda fully distributed in space. This is what is referred to as space uncoupling.
(ii) *Time uncoupling*. Linda allows communication between time-disjoint processes in that process "A" in which an "out()" instruction is present and can run to completion before process "B", in which the corresponding "in()" instruction appears, is charged. This contrasts with message passing, for which classically the sender and the receiver should perform their sending and receiving operations simultaneously.
(iii) *Distributed sharing*. Linda allows "n" disjoint processes in the address space to share a variable "v" by dropping it into "TS". The TS operator definitions ensure that "v" will be maintained atomically. With Linda, a shared variable is not necessary to be implemented by a process or a module. Linda allows "n" processes disjoint in address space to share a variable "v" by dropping it into TS.

The second characteristic of generative communication is the *free naming* [53]. This property can be represented in two points:

(i) *Support for continuation passing*. Consider "continuation passing", which represents the sending of data from one process to another. This process gets stuck waiting for a response and is eventually continued by a third (or nth) process. Linda supports continuation passing in a simple and flexible way by allowing values of the type name to be tuple components and to be stored in arbitrary data structures in the same way as values of any other type.
(ii) *Structured naming*. Linda allows "in()" and "read()" statements to restrict, and "out()" statements to widen, their focus. They also allow one name to generate a family of others.

In conclusion, the Linda coordination model enhances asynchronous communication among distributed agents by using tuples, allowing producers and consumers to operate independently. Its features, such as space and time uncoupling, enable seamless interactions, while free naming and continuation passing support dynamic communication. Overall, Linda streamlines distributed coordination, empowering developers to create more adaptable systems. Consequently, the next section will explore various coordination models and languages inspired by or creatively utilizing Linda.

1.3 Coordination Languages and Models

1.3.1 Basic Concepts of the Coordination Language

Having presented Linda, we now review coordination languages that were developed afterwards. Gelernter himself defined coordination as the process of building programs by gluing together active pieces [53]. The article [31] gives an interesting characterization of coordination in terms of three basic concepts:

(i) *The coordinables (agents or processes):* Coordinables are typically active computing entities programmed in many languages. The coordination of agents should not require their reprogramming but rather interoperate them as a whole.

(ii) *The coordination medium:* Instead of the simple approach of considering a few low-level primitives like send and receive to transmit messages over channels, the communication proposed by Linda is based on a shared data space, accessible to any agent.

(iii) *The coordination rules:* Coordination rules govern the relationship between the *coordinating agents* and the *coordinating medium*. They can be expressed either operationally or in more detail. Linda proposes a minimal set of primitives, implemented as library routines invoked by some host programming languages. However, other coordination languages, such as Tucson [92]), allow programming the reaction of the coordination medium to in and out primitives.

In their seminal paper [54], Gelernter and Carriero postulate that the computations necessary for the tasks to be performed and the communication required to coordinate the components, are orthogonal and as a result, do not need to be incorporated in the same model. This property has been summed up by them by the equation:

$$Programming = Coordination + Computation$$

Coordination languages and models thus favour the separate development and the eventual merging of these two major development phases [3]. It is also worth noting that the concept of coordination is by no means limited to the field of computer science but plays an essential role in many diverse disciplines such as economics, biology, operations research, business organization theory, and social media. In that sense, coordination is of a multidiscipline nature.

1.3.2 A Survey of Coordination Models and Language

Overview of Coordination

In [95], Papadopoulos and Arbab have presented an interesting classification of a number of coordination models and languages proposed in the literature. This

classification identifies two categories: the first category is called *"data-driven"* and the second one is *"control-driven"*.

Let's start with "*data-driven*". Languages of this category are characterized by the fact that the state of the computation at any time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components [54, 95]. In other words, a coordinated process is responsible for both examining and manipulating data as well as for coordinating either itself and/or other processes by invoking the coordination mechanism each language provides. This means that there exists a mixture of coordination and computation code within a process definition. More exactly, with respect to the above equation, [34], the separation between computation and coordination is done at the level of functionality. On the other hand, "*control-driven*" languages are characterized by the fact processes are of two categories. So-called worker processes input or output data on special ports. Other processes, called managers, are in charge of linking these ports so as to create flows of data. As a result, in opposition to the data-driven family, the value of the data being manipulated by the processes is of no more importance in the definition of the state of the computation, which is now to be defined in terms of the coordinated patterns allowing dataflows. In this thesis, we will focus on the first category of this classification, the *"data-driven"* one.

Survey of Data-driven Coordination Models and Language

We now review several key coordination models and languages closely linked to our research. We refer the reader to the article [37] for a detailed overview of coordination languages and models.

Gamma. Gamma [12] is a coordination model based on a *chemical metaphor:* it considers the elements of the data space as molecules that move freely in chemical solutions. Reactions between these molecules can occur, when they come into contact and satisfy certain constraints. Gamma programming sees the atomic values grouped in a single bag, and calculates the results of their individual interactions, following the principle of locality. The latter stipulates the independence of reactions between values to produce new ones. More explicitly, a Gamma program is a *pair(R, A)*, where *R* is a reaction condition, namely a Boolean function on multisets of data. The *A* component is a rewrite action, namely a function from multisets to multisets of data. Following this language, any group of molecules satisfying the reaction condition can be rewritten according to the corresponding rewrite action.

LINC. LINC [78] has been conceived as the natural evolution of the "Coordination Language Facility" (CLF) [5] model developed to deploy *distributed applications*. It is inspired by the Gamma reaction model for implementing reaction rules but also incorporates Linda's core primitives (out, in, rd) in a framework where each "bag"(i.e. tuple space) can be implemented differently. This constitutes a practical opportunity when it comes to physical mechanisms, in the case of deploying IoT

scenarios for instance, or to existing systems, in the case of databases. LINC also provides transactions to relieve developers of the load of backing up actions in case of error. The LINC architecture has revised CLF to fit the new landscape defined by the combination of cloud, Internet of Things, and cyber-physical systems. The main differences in this model are as follows:

(i) The coordination engine, in LINC, has been improved both in terms of CPU usage and memory footprint. It is integrated into each object, whereas CLF relied on more complex dedicated objects. This allows to distribute the coordination by delegating some parts to smaller CPUs.
(ii) The rule language and the environment have been extended to improve its expressiveness and the reexecution of the postmortem to debug the initial execution.

MESSENGERS. Messengers [52] is a paradigm based on the principles of *autonomous messages*, which assume their own behaviour in the form of a program. These messages allow them to freely navigate in the supporting computer network, to communicate with each other, and to call pre-compiled functions resident in the nodes. The coordination in this programming model facilitates the communication of this autonomous messages, at two levels [52]:

(i) The *intra-object coordination*: where the Messengers coordinate the invocation and exchange of data between the different functions distributed in the network, both in time and in space.
(ii) The *inter-object coordination*: where Messengers, each representing a high-level entity, can coordinate their behaviours among themselves.

Using a study of the collective behaviour of fish in the field of biology, the authors showed an interest of the approach through the coding of an application composed of autonomous and mobile entities whose behaviours can change dynamically and which can coordinate their actions with each other. In particular, they evidence the great flexibility in the interaction of entities, the ease of manipulation of the application at runtime as well as the performance obtained. This model was extended in the article [123]. This new version is a hierarchical model for the coordination of casts and communication of casts of concurrent activities based on the grouping of actors. The casts are used for naming, migration, and synchronization, while the Messengers are the actors used to send messages with special behaviour across casts.

Klaim. Klaim [87] or "Kernel Language for Agents Interaction and Mobility" is another coordination model featuring *multiple tuple spaces, mobility of code and agents*. This language is designed to manage the physical distribution of processes across a wired network. Klaim is able to manage the physical distribution of processes on networks, as well as to control changes in their configurations. To do this, Klaim considers a finite set of physical locations and associates nodes with these

locations. These nodes are composed of three elements: a *site*, a *process*, and an *allocation environment*. Internally,

(i) each *site* element is associated with a tuple space.
(ii) as for the *allocation environment*, its function is to associate certain (logical) localities with (physical) sites.

Klaim extends the classical Linda primitives *out, in, rd* and *eval*, as they can not only be invoked locally on the local tuple space, but also on the tuple space of a specified remote site by providing the locality of a node at the invocation of the primitive. Klaim also provides a specific primitive *newloc(u)* to bind a new site with a variable *u*. The implementation of the mobile code paradigms is achieved by combining *newloc(u)* with a remote invocation of primitives such as *eval*.

Limone. Limone [51] for "Lightly-coordinated Mobile Networks" is another coordination model addressing *mobility*. This model aims at facilitating the development of applications on ad hoc networks consisting of logically mobile agents and physically mobile hosts. It considers an agent-centered coordination perspective, by allowing each agent to define its knowledge policy, and limiting all interactions between agents. Agents are stored in a knowledge list, which is maintained by the system. Limone adapts Linda-like primitives by eliminating remote blocking and complex group operations. It provides timeouts for all distributed operations and reactions that allow asynchronous communication with agents in the knowledge list. Finally, it minimizes the granularity of atomic operations and the set of assumptions about the environment.

TuCSoN. TuCSoN [92] is a coordination model adopting Linda at its core, for mobile agent-based Internet applications, and shows its utility in distributed information search. With a focus on mobile agent coordination, TuCSoN also addresses the coordination of *Internet applications* based on mobile information agents. In order to deal with mobility, A. Omicini and F. Zambonelli introduced their approach by saying that a network node that wants to accept mobile agents must define its own local communication space, used by the agents to interact with other agents as well as with the local execution framework. In summary, the principle of this coordination language is:

(i) each local communication space consists of a multiplicity of global (with respect to the node) communication abstractions, called tuple centers. A tuple center is an improved tuple space, denoted by a name that is locally (with respect to a node) unique.
(ii) mobile agents can access each tuple center via a Linda-like communication interface. Such primitives include the Linda-like *"out, in"* and *"rd"* as well as their non-blocking versions *"inp"* and *"rdp"*.

In TuCSoN, coordination is also expressed by specifying the reactions of the tuple center to certain events, based on a logic-based language that specifies these

reactions: ReSpecT or "Reaction Specification Tuples" [91]. Besides introducing multiple tuple spaces, TuCSoN offers the possibility to program them as a key feature. The consequences of these extensions are respectively the parametrization of classical primitives at the particular tuple center accessed, and the processing of requests by the support, in addition to its access by the agents.

Bach. Bach [65] is an extension of Linda, developed at the University of Namur, on which our research is based. This language is based on four primitives to access a tuple space:

(i) the **"tell(t)"** primitive places an occurrence of tuple "t" in the tuple space;
(ii) the **"ask(t)"** primitive checks for the presence of tuple "t" in the tuple space;
(iii) the **"get(t)"** primitive deletes an occurrence of tuple "t" in the tuple space;
(iv) the **"nask(t)"** primitive checks for its absence.

It is interesting to note that the "*tell*" primitive always succeeds, while the last three primitives suspend as long as the presence/absence of tuple "t" is not satisfied. Moreover, the tuple space is seen as a multiset of tuples, which naturally leaves room for multiple occurrences. *Bach* has been shown to be expressive enough to code mobile applications in ad hoc networks. It is less expressive than Gamma (see [28]) but as we shall see later in the thesis expressive enough to code socio-technical systems.

Why Choose Bach? Insights into its utility compared to other languages

We provide in Table 1.1 an overview of various coordination languages, detailing their main features and suitability for different systems. Key elements in each column include:

- **Language**: the name of the coordination language.
- **Key Features**: main characteristics, such as coordination concepts, primitives, or metaphors used.
- **Concurrent Systems (C.S.)**: indicates if the language supports concurrent systems.
- **Asynchronous Communication (A.C.)**: specifies whether the language enables asynchronous communication, essential in distributed environments.
- **STS**: shows if the language supports communication via a shared tuple space, a common structure for storing and exchanging information.
- **Level of Understanding (L.U.)**: the difficulty level for users, rated as "Easy" or "Moderate."

Bach offers specific advantages over other languages discussed in this chapter and in Table 1.1, particularly for socio-technical systems and concurrent processes. Its simplicity, comparable to languages like Gamma, MESSENGERS, and Linda, is crucial for developers implementing coordination solutions without the burden of complex syntax. Additionally, Bach enhances the original Linda model with robust tuple management, making it well-suited for ad hoc networks and distributed

environments. Unlike LINC and Klaim, which introduce unnecessary complexity through their focus on distributed applications and mobility management, Bach emphasizes straightforward coordination of mobile agents. This simplicity facilitates rapid development and efficient interactions within socio-technical systems. While TuCSoN and Limone offer programmable tuple centers and agent-centered coordination, they may present a steeper learning curve for less experienced developers.

Language	Key Features	C.S	A.C	STS	L.U
Linda	Basic coordination language with simple primitives (‘out‘, ‘in‘, ‘rd‘) for tuple management.	✓	Yes	✓	Easy
Gamma	Chemical metaphor; supports parallel, independent reactions in data.	✓	Yes	✓	Easy
LINC	Adapted for distributed applications (IoT, cloud); efficient transactions and embedded coordination engine.	✓	No	✓	Moderate
MESSENGERS	Autonomous messages for flexible, mobile coordination.	✓	Yes	✓	Easy
Klaim	Manages mobility and distribution with multiple tuple spaces and remote primitives.	✓	Yes		Moderate
Limone	Agent-centered, non-blocking communication with timeouts in ad hoc networks.		Yes		Moderate
TuCSoN	Programmable tuple centers for mobile agents in Internet applications.	✓	Yes	✓	Moderate
Bach	Linda extension with tuple management, ideal for coordination in ad hoc networks.	✓	Yes	✓	Easy

Table 1.1: Summary of coordination languages listed in Section 1.3

Furthermore, Bach supports asynchronous communication, enhancing flexibility and responsiveness by allowing agents to operate independently without waiting for responses. This aligns with Bach's intuitive design philosophy, making it a valuable asset for teams aiming to improve collaboration in complex systems. Its ease of understanding, applicability to concurrent processes, and effective coordination capabilities position Bach as a highly usable and adaptable choice among coordination languages. Moreover, Bach can be considered one of the representatives of “data-driven” coordination languages and thus is very easy to integrate with other languages from the same family.

1.4 Socio-Technical Systems

1.4.1 Overview

Applications are rarely conceived in isolation. In contrast, most of them imply a form of interaction with users. However, in general, such interaction often results in structural reorganizations within companies. This has resulted in the consideration of so-called *socio-technical systems (STS)*. Several definitions of these systems have been provided in the literature. The most often cited one is however that proposed by Trist.

In [118], the authors define socio-technical systems as systems involving complex dependencies between human beings, machines and the environment in which they operate. One of the challenges posed by such systems comes from the fact that STS have to marry two different worlds, the social and technical worlds, driven by different concerns. In fact, in the social world, cognitive, psychological, and organizational aspects lead people to formulate high-level goals to be pursued and state of affairs to be achieved, these in general depending upon a current context.

In contrast, in the technical world, performance and reliability are two key concepts that are often expressed in terms of actions at the level of perceptions and measurements provided by sensors. There is thus a conceptual gap to be overcome. Obviously, to that end, the best solution is to somehow increase the abstraction level understandable by devices rather than forcing humans to speak the "language of devices". Another challenge posed by STS is that their design should pay special attention to interaction. Indeed, as people interact with each other through technology, supporting and governing interactions both from *social-to-technical, technical-to-social, social-to-social, and technical-to-technical* directions is of crucial importance. Following [81], we think that these two challenges may be faced by Linda-like languages and in particular the Bach language. Indeed, by clearly separating the interactional and computational aspects of software components, coordination models have proven to be well-suited to program the interaction of conventional distributed systems, and in particular to model service-oriented applications (see eg [26, 68, 116]). Their expressiveness [28, 43, 25] is such that it is to be expected that they also offer suitable means to handle other forms of socio/technical to socio/technical interactions mentioned above. Moreover, information produced by sensors may be materialized as tuples, traditionally used in data-driven coordination models, which in turn may be looked at as atomic formulae of logic programs whose rules may be used to perform inferences of high-level goals. This new way of looking at tuples opens the possibility of regarding pieces of information produced by sensors as the basic blocks to be manipulated by rules of logic programs to infer high-level goals expressed in the social world. Finally, as suggested in [65] suitably handling relations on multiple blackboards provides an elegant means to capture contexts.

As *STS* involves socio and technical dimensions, Mariani identifies in [79] two major dimensions in which to sum up research efforts in coordination devoted to STS: on one hand, social-to-technical efforts devoted to reflecting all the nuances of social activity and relationships in technical terms and, on the other hand, technical-to-social approaches used to exploit the full potential of technical platforms in order to enhance the effects of social interactions. We subsequently review these efforts according to these two dimensions and, in particular, sketch how coordination can bring solutions to problems encountered. The following two sections are largely inspired by [79].

1.4.2 Technical-to-Social Approaches

As defined in [79], *technical-to-social* coordination approaches are those approaches "concerned with how to exploit the technical components of the STS to (a) enhance the interaction capabilities of human users as well as to (b) exploit the activities, side effects , and outcomes of their collaborations to improve themselves". Two approaches are of particular interest: *observation-based coordination* and *self-organization based coordination.*

Observation-Based Coordination

As depicted in [79], *observation-based coordination* captures the idea of coordinating an ensemble of agents by enabling them to observe each other's actions or the traces that those actions leave in the environment. *Stigmergy* is a key concept in these lines. It denotes agents communicating and synchronizing their activities through the environment rather than by directly communicating. It is worth observing how data-driven coordination models nicely meet stigmergy. Indeed, the shared space on which they rely offers an easy means to create an environment on which the agent can communicate and synchronize. The term stigmergy has been originally introduced by Grassé [58] to define the coordination approach of a specific species of termites in collectively building their nest, Then, throughout the years, it has undergone many generalizations/specializations/extensions [96, 106, 89]. Here, we refer to a generic set of coordination mechanisms mediated by the environment. Let us examine some examples of stigmergic coordination.

LAbS. LAbS, the *Language with Attribute-based Stigmergies*, introduced in [88] is a good example of coordination language featuring stigmergy. This language has been developed for multiagent systems (MAS), in the aim of modelling several classes of systems that lend themselves to the intuitive design of local specifications and that can be used as the basis for automated analysis. According to the concept of *virtual stigmergy*, LAbS has a distributed and decentralized data structure, to model the communication between agents or the propagation of knowledge. LAbS combines the stigmergic interaction in [99, 106] and generalizes the communication based on the attributes introduced in [4]. It models the external environment as a shared data repository accessible by the different agents. In LAbS, agents can

only access their local copy of the stigmergy. This language offers a communication mechanism based on agent attributes instead of limiting message exchanges in the spatial neighbourhood. This helps to describe, and reason about different types of distributed systems and to introduce multiple stigmergies. For the evaluation of the quality and power of the LAbS language, the authors used some classical case studies, such as Boids (short for bird-oid objects) [103].

MoK. MoK, the *Molecules of Knowledge model*, introduced in [82, 80] is a way of designing computing platforms that support knowledge management in STS. Accordingly the software exploits user interactions to self-organize information in a continuous way. MoK is built around the integration of (a) a biochemical metaphor [122] to define how to perform computations, also (b) a BIC [35] that defines how to manage interactions. Mariani [80] defined the MoK system as a network of compartments, representing information repositories, where:

(i) the seeds (where information sources) continuously and spontaneously inject atoms (atomic information),
(ii) the atoms can then aggregate into molecules (composite information),
(iii) the molecules diffuse to other compartments, gain/lose relevance, etc.

Respecting the laws of coordination to dictate the evolution of the system, processes are implemented by MoK reactions [80], which take place within the compartments and are influenced by enzymes (the reification of the agents' actions) and traces (their side effects). Enzymes and traces are left in the compartments by the catalysts (the agents) during their activities. These reactions take advantage of decentralization and localization to promote self-organization: first, they rely only on local information in their compartment and can only affect neighbours. Second, they are programmed according to dynamic rate expressions inspired by natural chemical reactions, which are sensitive to contextual information affecting (eventually) their own outcomes.

Self-Organization Based Coordination

Having agents perform tasks independently while synchronizing through their environment naturally leads to self-organisation and even one step beyond to decentralisation. As noted in [79], this is actually a sort of holy grail for coordination models and languages. It would indeed be fantastic to conceive systems able to autonomously reconfigure themselves in the presence of changes on the sole basis of locally available information, without any global supervision. Here again, rule-based coordination [67] and multiple shared spaces [24] can be of particular interest. In fact, multiple spaces allow creating local environments. Moreover, by inserting or retrieving tuples dynamically on events, blackboard rules allow to achieve a form of autonomous behaviour.

It is worth observing that the decentralized approach of self-organization is used by systems of a natural type, in order to enable an emerging phenomenon

and to harmonize the local interaction effects logically. As an imitation, a number of pieces of work have highlighted the ways in which decentralization of investment decision-making authorities can be implemented, such as G.Michael and A.Campbell in [56] and J-P. Denis's doctoral thesis [46], which focused on the analysis of the coordination relationships between centers and branches inside multidimensional groups of an organization. This is the most studied fundamental perspective at the level of group management in a unit that does not deal with process strategy/tactics. Babaoglu and al., 2006 [7] have demonstrated with examples the importance of design model families for the *distributed computing* on the basis of these models. Also, the article [130] has shown that nature-inspired models of *coordination* can be well adapted to respond to the challenging needs of invasive environments.

The evolution of information during these years has created new challenges in the knowledge production process, facing the heterogeneity of the information structure and the increasing quantity. In these socio-technical environments [81] as an example, software systems are characterized by a huge amount of information to process and the user's behaviour deeply affects the behaviour of the system itself.

1.4.3 Social-to-Technical Approaches

Social-to-technical coordination approaches are defined in [79] as those approaches that let technical components of the STS facilitate social relationships, protocols, expectations, conventions, and norms. The key issues for these approaches are social protocols and argumentation.

Social Protocols

In [36], Chopra and Singh proposed a *social protocol* based on a software engineering paradigm called *Interaction Oriented Software Engineering (IOSE)*. This paradigm concentrates on social protocols (social relationships), characterizing the responsibility of the parties involved as they progress according to their interaction.

It adopts the representation of responsibility (such as openness and innovation that motivate social machines), with the help of computer media, to grasp the significance of the states and transitions of a social machine. The technical component of an STS, named "SE-machine" are implementation of social protocols, that facilitate the interaction between social relationships and the responsibility of the interacting parties during the communication. According to the article [36], the social protocol has three objectives:

(i) make social expectations explicit in an environment to follow individual objectives,
(ii) identify who is responsible for the interacting parties, for the wait and who is expected to wait (for whom for what wait),

(iii) release the principals for the implementation of SE-machines, impose only one compliance standard.

In order to achieve their goal and as shown in Figure 1.3, Chopra and Singh al in [36] use a technical mechanism based on the notion of commitment, which is also exploited by Baldonia et al in [9]. The latter introduces a relationship of expectations and responsibility between two interacting parties: the *creditor* expects the debtor to respect his commitment, while the *debtor* accepts to be responsible for it. The treatment of undertakings, in the sense of their control, coordination, and execution, is an efficient way to achieve, in terms of calculation, the social protocols that play a crucial role in STS. In this qualification of "Social-to-Technical", the authors of [9] and [10], propose an explicit representation in terms of a set of relations defined in a normative way between two or more parties, subject to social control through the monitoring of observable behaviour. In this theme [36], social relations are resources, at the typical agents' disposition without relying on a functional approach, who use them in their practical reasoning.

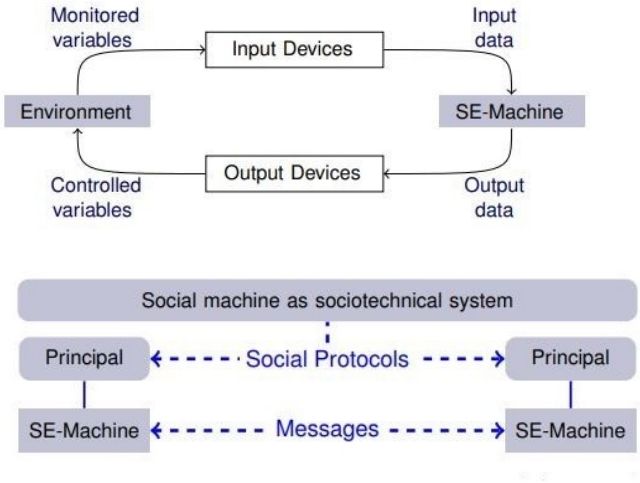


Figure 1.3: Social Protocols in Decentralized Social Machines via IOSE [36]

The IOSE, as shown in Figure 1.3, introduces social protocols as a way of specifying decentralized social machines. In doing so, it goes beyond the machine-oriented conception of traditional SE [36]. The first figure presents the current machine-oriented software engineering: given the STS requirements, design an SE machine to control the environment. The SE-machine is conceptually monolithic, although it can be implemented as a distributed system. The second illustration shows the IOSE model that gives the STS and designs a social protocol. Principals enact the protocol via their respective SE-machines, which communicate by messaging. A principal's SE-machine interfaces with the environment, exactly as depicted in the left figure, and helps a principal participate in a social protocol.

JaCaMo. JaCaMo [22] is a *multi-agent oriented programming* platform that provides programming constructs, corresponding to the abstractions used at the level of a multi-agent system, to express social coordination. This platform [22] describes a global approach that integrates the agent, environment, and organization levels, by providing a programming model that aims to efficiently and synergistically integrate the related abstraction dimensions. The agent programming in JaCaMo is supported by the Jason programming language [23] while the environmental engineering is made possible by the notion of artefact provided by the CArtAgO framework [107]. JaCaMo has extended for a new platform, JaCaMo+ [11] for MAS development. On the one hand, the first approach is focused on the programming of agents, environment and organizational aspects. It is exploited to build a customized coordination artefact that follows the normative state of the MAS, i.e., the collection of commitments. On the other hand, JaCaMo+ enriches these commitments to allow interacting agents to react to a change in the state of the commitments.

Argumentation

According to [79], computational argumentation exploits computational techniques to automatically analyse and construct arguments and their relationships [122], including generating explanations and justifications for decision-making [33]. Argumentation-based coordination approaches have significant confidence in STSs, above all in the "Social-to-technical" category, as they work to capture in computational terms the rules of engagement and progression of different types of human dialogues, all aimed at enabling people to cooperate to make decisions. Thus, here, the social protocol to which agents must conform is defined by the rules of the argumentation framework taken as a reference, and the set of norms and regulations derives from the rules deciding which arguments are admissible, better than others, and, ultimately, win the debacle.

In other words, if users can get justifications for decisions made, they are likely to increase their confidence in the system's capabilities. In this regard, it is worth noting that efforts to ensure trust and responsibility are an increasingly heated topic, well outside of coordination in STSs, but in many areas of AI (from big data to algorithms in general [85]) as evidenced by the recently approved "transparency initiative" by many organizations around the world. Taking advantage of the situatedness and argumentation, the "Speaking Objects" vision is emerging. The principle of Speaking Objects is to give users the right level of abstraction when interacting with technical systems.

Speaking Objects. According to [77], the concept of "speaking objects" is described as a general way of designing distributed systems, with a particular focus on the vision of the Internet of Things. The core idea [79] is that sensors and actuators no longer just provide readings of predefined parameters and respond to simple commands to change the state of the local environment. Instead, they are empowered to assert complexities about the state of the world and to pursue goals

assigned to users or explicitly set for the system itself. Essentially, this amounts to a shift from actions and perceptions to goals and propositions. Key factors behind this paradigm shift are increases in computing power that can be built into everyday objects, and advances in machine learning techniques In this context, coordination becomes the ability to debate the "state of affairs" and initiate a dialogue to jointly decide how to act to achieve the desired future state.

1.5 Closing Remarks

The elements presented in this chapter are crucial for understanding both existing coordination models and the socio-technical systems that influence our world today. We explored how data-based coordination languages effectively manage distributed applications through mechanisms like publication and retrieval, which offer benefits such as space and time uncoupling. The chapter also highlighted the need to integrate technical and social dimensions in system design. Coordination languages bridge the gap between social goals and technical actions, essential for managing complex socio-technical systems.

In the next chapter, we will introduce the Bach coordination language, which serves as the foundation for this thesis.

CHAPTER 2

BACH COORDINATION LANGUAGE

The rest of the thesis relies on the Bach coordination language, developed at the University of Namur. In this chapter, we introduce its key concepts: tokens, structured information terms, primitives, agents and operators. They are illustrated on a running example, featuring restaurants employing tablet ordering systems. Implementation issues are also tackled and an incarnation in Scala is proposed.

2.1 Opening

The Bach coordination language [65] is a Linda dialect developed by the *CoordiNam* research group of the University of Namur, for the efficient coordination of complex systems. This coordination language provides a systematic approach to describe the behaviour of entities and the operations that must be performed on them in a coordinated way.

To delve into the concepts and principles of the Bach coordination language with precision and using concrete examples, we initiate this chapter by elucidating our case study in Section 2.2: an intelligent restaurant that employs tablets for order placement. Subsequently, in Section 2.3, we provide an overview of the data, primitives, and agents inherent to Bach. Moving on to Section 2.4, we demonstrate the modelling of the Altar example using our base language. With focus on enhancing user-friendliness, Section 2.5 presents the implementation of Bach in Scala, and Section 2.6 its embedding in Scala. Finally Section 2.7 draws a conclusion.

2.2 Altar, a Tablet-Based Case Study

To illustrate the modelling with Bach, we shall use a tablet-based restaurant application. We shall subsequently name it Altar, as an anacronym for A wonderful TablEt-based Restaurant.

In the aim of being modern and providing a nice experience, the main idea of the Altar restaurant is to use the latest technology to order dishes, have them served and pay at the end of the meal. More precisely, as depicted in Figure 2.2, each table is equipped with an interactive electronic tablet. Clients are assumed to have created an account before entering the restaurant[1]. After having logged on the tablet, they can consult what is on the menu, place orders (possibly multiple times), track the status of the orders, and pay at the end of the meal. The cooks receive the order in the kitchen and prepare the dishes. Once done, they inform the waiters which serve the dishes at the ordering table. To simplify the modelling, we shall assume that a few Japanese dishes and drinks can be served: nigiri, maki, temaki and beer.

As we shall see, the Bach coordination language provides a solid framework for coordinating the actions of all the agents, including clients, kitchen staff, and waiters. It enables transparent communication and ensures that all parts of the restaurant work together efficiently.

2.3 Bach in a Snapshot

To describe Bach, one needs to explain how information placed on the shared space is represented and how it is handled by concurrent processes running around the shared space. This respectively leads to the two following subsections.

2.3.1 Data

Tokens are used as the fundamental elements to represent information. They consist of discrete units of information with no structure. In contrast, so-called **structured information terms** (or si-terms for short) take the form $f(t_1,...,t_n)$ where f is a functor and the t_i's are either tokens or si-terms. As usual, we shall consider tokens as particular structured information terms with no arguments. This allows us to define si-terms as follow:

Definition 1. ***Si-Terms*** *are defined as either flat tokens or structured information terms. Their syntax is shown in Figure 2.1.*

Example 1. *In the Altar example 2.2,* `manel` *is a token representing a client (in fact, the PhD student writing this thesis), and* `order(manel, maki)` *is a si-term expressing an order placed by manel to get maki.*

[1] A part that we shall not model.

IDENTIFIER

lid ::= any string of letters and digits starting with a lower case letter

SI-TERMS

f ::= *lid*	function name
si ::=	si-terms
lid	token
$f(si_1, \cdots, si_n)$	structured term

Figure 2.1: Syntax of Si-Terms

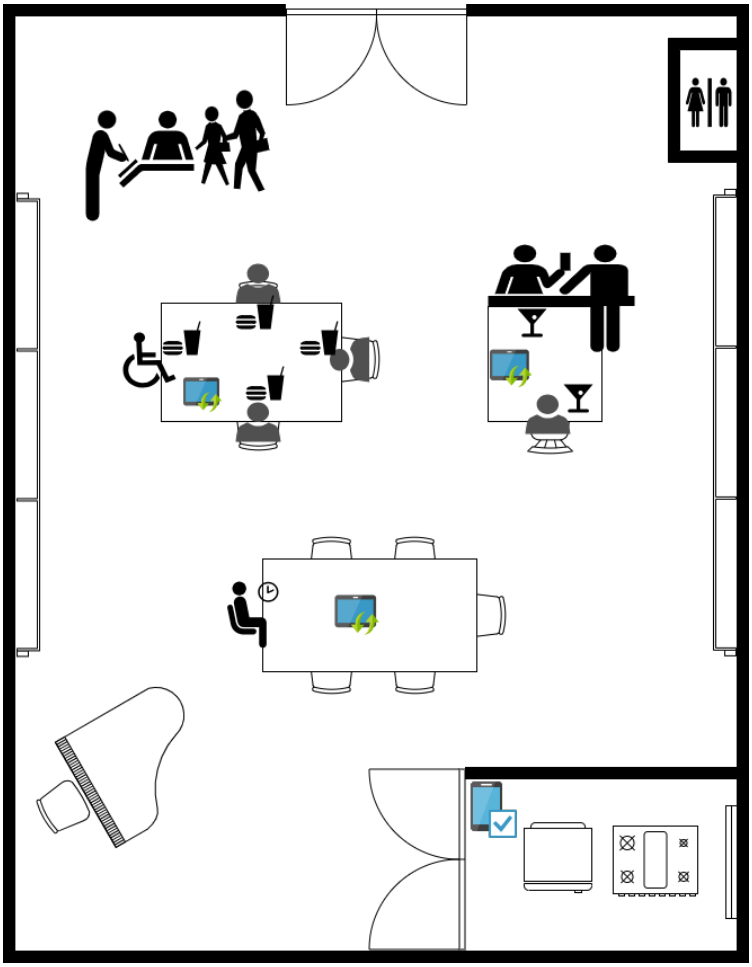


Figure 2.2: Example of “Altar Restaurant”

2.3.2 Statements

Primitives

Inspired by Linda [34], Bach relies on four key primitives tailored for shared space interaction. They are depicted in Figure 2.3. They consist of the primitives:

- "*tell(t)*" which places an occurrence of si-term "*t*" on the shared space,
- "*ask(t)*" which checks for the presence of si-term "*t*" on the shared space,
- "*get(t)*" which deletes an occurrence of si-term "*t*" on the shared space,
- "*nask(t)*" which dually checks for its absence.

It is important to note that the "*tell*" primitive always succeeds[2], while the last three primitives suspend as long as the presence/absence of the si-term "*t*" is not satisfied. Moreover, the shared space is seen as a multiset of si-terms, which naturally leaves room for multiple occurrences.

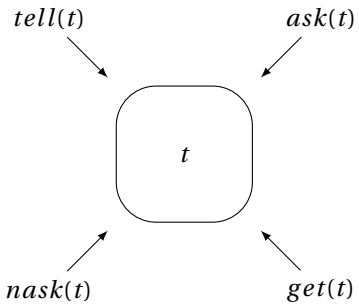


Figure 2.3: Primitives in Bach

Example 2. *In the Altar example,* `tell(order(manel, table_3, 5, maki))` *may be used to indicate that client* `manel` *orders maki at table 3 as a fifth order, while* `ask(status(manel, 5, served))` *may be used to check whether the fifth order from* `manel` *has been served.*

Agents

Primitives may be combined to construct more complex agents, using traditional composition operators from concurrency theory:

[2]Here and in the following we will say that a primitive "succeeds" or "fails" by reference to the logic programming framework – which has inspired Bach – to denote the fact that the computational transition associated with the primitive has been done or cannot be done, respectively. Intuitively a primitive succeeds when it can be computed. It suspends when it cannot computes but may do so later in a more global computation in which the primitive is involved.

- sequential composition noted ";",
- parallel composition noted "||",
- and non-deterministic choice noted "+".

Following concurrency theory, procedures may also be defined to describe agents. Their definition takes the following form

$$proc\_name(arg_1, \cdots, arg_n) = agent$$

where $proc\_name$ is the name of a procedure, the arg_i's are arguments consisting of si-terms and $agent$ is an agent, as just described. Following this introduction, procedure calls can be used as primitives in the composition of agents.

Definition 2. *Primitives, agents and procedure calls are associated to the syntax shown in Figure 2.4:*

Example 3. *Coming back to the Altar restaurant, the behaviour of a client identified by id can be depicted as follows:*

```
1 Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

This definition involves $Client$ as a procedure name, id as an argument and calls sequentially procedures $LogIn$, $EnjoyAltar$ and $LogOut$.

2.4 Modelling Altar in Bach

To illustrate the expressiveness of Bach, let us see how the Altar restaurant can be modelled. Three kinds of agents need to be tackled: the clients, the cooks and the waiters. Assuming that we have three clients (manel, alice and emma), one cook (john) and one waiter (tom), the behaviour of the restaurant can be modelled by the following parallel composition:

```
1 Client(manel) || Client(alice) || Client(emma) || Cook(john) || Waiter(tom)
```

All these five agents will interact via the shared space and not directly by sending messages between them. As a result, it is quite easy to introduce a new client, cook or waiter: one has just to introduce it in the parallel composition. Let us now code these three kinds of agents.

2.4.1 Modelling a Client

Clients, and more generally all the three agents, are modelled in a generic way by introducing an identifier as an argument. We have already described the behaviour of a client as doing a log in, enjoying his time at the Alter restaurant and performing a log out:

IDENTIFIERS

lid ::= any string of letters and digits starting with a lower case letter
uid ::= any string of letters and digits starting with an upper case letter

PRIMITIVES

cprim ::=	communication primitive
tell(si)	
ask(si)	
nask(si)	
get(si)	

PROCEDURES

$pn ::= uid$	procedure name
$pc ::= pn(si_1, \cdots, si_n)$	procedure call
$pharg ::= lid : sname$	arguments in procedure head
ph ::=	head of procedure declaration
$pn(pharg_1, \cdots, pharg_n)$	
$peq ::= ph = ag.$	procedure equation

AGENTS

ag ::=	agents
cprim	communication primitive
pc	procedure call
$ag_1; ag_2$	sequential composition
$ag_1 \| \| ag_2$	parallel composition
$ag_1 + ag_2$	non-deterministic choice

Figure 2.4: Syntax of primitives, agents and procedure calls

```
1 Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

Doing a log in or a log out is of no particular interest: this is respectively modelled through the telling and getting of the presence of the client, as follows:

```
1 LogIn(id) = tell(logged(id))
2 LogOut(id) = get(logged(id))
```

Describing the time spent by a client in the Altar restaurant is more interesting. It involves sitting at a table, placing orders, checking for the status of the orders and finally paying. Assuming four tables, numbered *t1* to *t4*, this can be coded as

follows, with procedure *SitAtTable* figuring a client sitting at a table and procedure *EnjoyAtTable* summarizing the remaining action.

```
EnjoyAltar(id)  =  SitAtTable(id,t1); EnjoyAtTable(id,t1)
                 + SitAtTable(id,t2); EnjoyAtTable(id,t2)
                 + SitAtTable(id,t3); EnjoyAtTable(id,t3)
                 + SitAtTable(id,t4); EnjoyAtTable(id,t4)
```

It is worth noting that the above choice repeating similar alternatives can be abbreviated in one line by means of the following general sum

sum(t in [t1, t2, t3, t4]): SitAtTable(id, t); EnjoyAtTable(id,t) (2.1)

Note also that this construct should be understood as a shorthand for the choice obtained by replacing *t* with each possible value in the group *(t1,t2,t3,t4)*, resulting in the following alternative definition of *EnjoyAltar*:

```
EnjoyAltar(id) = sum(t in [t1, t2, t3, t4]):
                     (SitAtTable(id, t) ; EnjoyAtTable(id, t))
```

Making an order is simulated by placing an order request on the shared space, namely of telling a si-term of the form $order(id,t,d)$ for some dish d. This will be taken by the cook who will indicate that the dish is ready by telling a $order\_prepared$ si-term. Similarly a waiter will indicate that the order has been served by taking a $order\_prepared$ si-term and telling a $order\_served$ si-term. Hence checking that the status of an order amount to checking the presence of one of the above si-terms. This results in the following code:

```
                MakeOrder(id,t)  =  tell(order(id,t,nigiri))
                                  + tell(order(id,t,maki))
                                  + tell(order(id,t,temaki))
                                  + tell(order(id,t,beer))
CheckOrder(id,t,d,in_progress)   =  ask(order(id,t,d))
   CheckOrder(id,t,d,prepared)   =  ask(order_prepared(id,t,d))
     CheckOrder(id,t,d,served)   =  ask(order_served(id,t,d))
```

Note that for simplicity of the reading, we have preferred to write the choice in *MakeOrder* explicitly instead of using the shorthand version

```
MakeOrder(id,t) = sum (d in [nigiri, maki, temaki, beer]):
                      tell(order(id,t,d))
```

Finally, counting the bill is left outside this simple exercise and is just coded by telling the si-term $orders\_paid(id,t)$:

```
PayOrders(id,t) = tell(orders_paid(id,t))
```

2.4.2 Modelling a cook

The behaviour of a cook consists of repeadly taking orders and of preparing them. Concretely, this amounts to getting a si-term $order(id,t,d)$, from some user identified by id, sitting at table t and asking for the dish d, and afterwards of telling a $order\_prepared(id,t,d)$ si-term. Roughly speaking, this can be coded as:

```
Cook(id_cook) = get(order(id,t,d)); tell(order_prepared(id,t,d))
```

However, although id_cook is an argument, which does not require quantification, the id, t, and t arguments should be quantified on all possible values. This can be achieved by using non-deterministic choices. Another way of proceeding is to provide generalized choices used before accounting for these quantifications. We shall use both subsequently by balancing the case of reading and the length of the resulting code.

The first quantification on the id of users is operated through the $CookForCustomer$ procedure, as follows

```
Cook(id_cook) =   CookForCustomer(id_cook,manel); Cook(id_cook)
                + CookForCustomer(id_cook,alice); Cook(id_cook)
                + CookForCustomer(id_cook,emma); Cook(id_cook)
```

Similarly, the $CookForCustomer$ procedure can be coded by the $CookForCustomerAtTable$ procedure by making explicit the tables.

```
CookForCustomer(id_cook,id_client) = sum(t in [t1, t2, t3, t4]):
                                       CookForCustomerAtTable(id_cook,id_client,t)
```

Finally, the $CookForCustomerAtTable$ procedure can be coded by making explicit the dishes, as follows:

```
CookForCustomerAtTable(id_cook,id_client,id_table) =
        sum(d in [nigiri, maki, temaki, beer]):
                    get(order(id_client,id_table,d));
                    tell(order_prepared(id_client,id_table,d))
```

2.4.3 Modelling a Waiter

A waiter has a behaviour similar to a cook. It waits for an $order\_prepared$ si-term and then puts and $order\_served$ si-term to indicate that the dish has been served. The following code results by analogy to what we wrote for a cook.

```
Waiter(id_waiter) =
      WaiterForCustomer(id_waiter,manel); Waiter(id_waiter)
          + WaiterForCustomer(id_waiter,alice); Waiter(id_waiter)
          + WaiterForCustomer(id_waiter,emma); Waiter(id_waiter)
```

```
5
6  WaiterForCustomer(id_waiter,id_client) =
7          sum(t in [t1, t2, t3, t4]):
8                  WaiterForCustomerAtTable(id_waiter,id_client,t))
9
10 WaiterForCustomerAtTable(id_waiter,id_client,id_table) =
11         sum(d in [nigiri, maki, temaki, beer]):
12                 get(order_prepared(id_client,id_table,d));
13                 tell(order_served(id_client,id_table,d))
```

2.5 Interpreting Bach using Scala

In order to allow the reader to experiment with the languages he has proposed, D. Darquennes has presented in his PhD thesis [43] interpreters for these languages as extensions of an interpreter he developed jointly with J.-M. Jacquet. They have chosen Scala as the implementation language in view of the combination of object and functional programming it offers. These interpreters will also be the basis of our work. To provide the reader with the background necessary to understand our thesis, we briefly outline in this section the core features of their implementation. Besides a parser which is left outside the scope of this section, they concern three main aspects: the representation of data, the representation of the shared space and the simulator making the computations.

2.5.1 Data Representation

The Bach version tackled by D. Darquennes and J.-M. Jacquet uses only si-terms, which are represented as strings. This can be easily extended to more complicated si-terms by using Scala case classes, composed of two arguments: the si-term functor and a list of arguments.

With respect to the agents, three kinds of case classes have been used:

- a case class `bacht_ast_empty_agent` to represent a terminated agent
- a case class `bacht_ast_primitive` to represent a primitive in the form of a pair composed of primitive type (tell, ask, nask, get) and of a token (being possibly generalized to a si-term)
- a case class `bacht_ast_agent` to represent a composed agent formed from an operator applied to two sub-agents `agenti` and `agentii`.

2.5.2 Shared Space

The shared space is implemented as a mutable map. Initially empty, it is enriched for each told token (resp. si-term) by an association of this token (resp. si-term) with a number representing the number of its occurrences on the shared space.

```
class Expr
case class bacht_ast_empty_agent() extends Expr
case class bacht_ast_primitive(primitive: String, token: String) extends Expr
case class bacht_ast_agent(op: String, agenti: Expr, agentii: Expr) extends Expr
```

Figure 2.5: Internal representation of agents

More precisely, the execution of a *tell* primitive, say *tell(t)* consists in checking whether t is already in the map. If it is then the number of occurrences associated with it is simply incremented by one. Otherwise a new association *(t,1)* is added to the map. Dually, the execution of *get(t)* consists in checking first whether t is in the map and then whether it is associated with a strictly positive integer. In this case, the execution of *get(t)* consists in decrementing by one the number of associated occurrences. In case one of these two conditions is not met then the *get* primitive cannot be executed. Note that with this simple strategy, a token may appear in the map but with 0 as a number of occurrences associated with it. Hence the implementation has not only to test whether the token appears in the map but also to test whether the associated number of occurrences is more than one. The *ask* primitive has a similar behaviour without removing an instance. Finally the *nask* primitive has an opposite behaviour, succeeding in case the *ask* primitive fails and failing in case it succeeds.

The code for the primitives is presented in Figure 2.6. It is worth noting that, in view of the similarity of Bach to concurrent logic programming, the shared space has been named as store, hence *BachTStore* as the name of the class.

2.5.3 Simulator

The simulator consists in repeatedly executing a computation step. This boils down to the definition of function `run_one`, which assumes an agent in internal form and which returns a pair composed of a boolean and an agent in internal form. The boolean aims at specifying whether the attempted computation step can indeed take place. In this case, the associated agent consists of the agent resulting from the computation step. Otherwise, failure is reported with the given agent as associated agent.

The function assumes a Bach store, as specified in the previous subsection. It is given as a parameter of the `BachTSimul` in which `run_one` is defined. The function is defined inductively on the structure of its argument, say `agent`. If it is a primitive, then the `run_one` function simply consists in executing the primitive on the store. This is technically achieved by the `exec_primitive` function, which actually calls the associated primitive function on the store.

If `agent` is a sequentially composed agent ag_i ; ag_{ii}, then the computation step proceeds by trying to execute the first subagent ag_i. Assume this succeeds and delivers ag' as resulting agent. Then the agent returned is ag' ; ag_{ii} in case ag' is

```
import scala.collection.mutable.Map

class BachTStore {

   var theStore = Map[String,Int]()

   def tell(token:String):Boolean = {
      if (theStore.contains(token))
        { theStore(token) = theStore(token) + 1 }
      else
        { theStore = theStore ++ Map(token -> 1) }
      true
   }


   def ask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { true }
             else { false }
      else false
   }


   def get(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1)
               { theStore(token) = theStore(token) - 1
                 true
               }
             else { false }
      else false
   }


   def nask(token:String):Boolean = {
      if (theStore.contains(token))
             if (theStore(token) >= 1) { false }
             else { true }
      else true
   }
```

Figure 2.6: Primitives and the shared space

not empty or more simply ag_{ii} in case ag' is empty. Of course, the whole computation fails in case ag_i cannot perform a transition step, namely in case `run_one` applied to ag_i fails. The code for these two first cases is presented in Figure 2.7.

The cases of the composed agent by a parallel or choice operator are more subtle. Indeed for both cases one should not always favour the first or second subagent. To avoid that behaviour, a random variable `branch_choice` is used, with assigned values 0 or 1. If 0 is given to the variable, then the computation starts with the first subagent. Otherwise, it starts with the second. In case of failure, the other one agent is evaluated and if both fails a failure is reported. In case of success for the parallel composition the resulting agent is determined in a similar way to what is done for the sequentially composed agent. For a composition by the choice operator the tried alternative is simply selected. The code for these two cases is reported in Figures 2.8 and 2.9.

Given the `run_one` function, the simulator mainly consists of a loop which is executed while the current agent is non empty and while failure does not occur. This is materialized in the `bacht_exec_all` function detailed in Figure 2.10.

```
1   def run_one(agent: Expr):(Boolean,Expr) = {
2
3      agent match {
4
5         case bacht_ast_primitive(prim,token) =>
6            {  if (exec_primitive(prim,token)) { (true,bacht_ast_empty_agent()) }
7               else { (false,agent) }
8            }
9
10        case bacht_ast_agent(";",ag_i,ag_ii) =>
11           {  run_one(ag_i) match
12                 { case (false,_) => (false,agent)
13                   case (true,bacht_ast_empty_agent()) => (true,ag_ii)
14                   case (true,ag_cont) => (true,bacht_ast_agent(";",ag_cont,ag_ii)
                          )
15                 }
16           }
17
18        ...
19
20     }
21  }
22
23
24  def exec_primitive(prim:String,token:String):Boolean = {
25        prim match
26           { case "tell" => bb.tell(token)
27             case "ask"  => bb.ask(token)
28             case "get"  => bb.get(token)
29             case "nask" => bb.nask(token)
30           }
31     }
32
33  }
```

Figure 2.7: Bach simulator: primitive and sequential composition

2.6 Embodying Bach in Scala

2.6.1 Some Notes on Scala

Following the work of P. Matiello and A. de Melo [83] to implement the π-calculus as a domain specific language hosted in Scala, it is possible to turn the implementation of Bach proposed by D. Darquennes and J.-M. Jacquet into a domain specific language hosted in Scala. To that end, two main features of Scala are worth stressing.

Functions and Objects

Functions may be coded by defining objects with an apply function. For instance, if we define:

```
val bach_random_choice = new Random()

def run_one(agent: Expr):(Boolean,Expr) = {

   agent match {

      ...

      case bacht_ast_agent("||",ag_i,ag_ii) =>
         {  var branch_choice = bach_random_choice.nextInt(2)
            if (branch_choice == 0)
              { run_one( ag_i ) match
                   { case (false,_) =>
                       { run_one( ag_ii ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                   }
               }
            else
              { run_one( ag_ii ) match
                   { case (false,_) =>
                       { run_one( ag_i ) match
                           { case (false,_)
                               => (false,agent)
                             case (true,bacht_ast_empty_agent())
                               => (true,ag_ii)
                             case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_cont,ag_ii))
                           }
                       }
                     case (true,bacht_ast_empty_agent())
                               => (true,ag_i)
                     case (true,ag_cont)
                               => (true,bacht_ast_agent("||",ag_i,ag_cont))
                   }
                }

            }
```

Figure 2.8: Bach simulator: parallel composition

```
object tell  { def apply(siterm: SI_Term)  = TellAgent(siterm)  }
object Agent { def apply(agent: BSC_Agent) = CalledAgent(agent) }
```

then the evaluation of:

```
val P = Agent { tell(f(1,2)) }
```

consists first in evaluating $tell$ on the si-term $f(1,2)$, which results in the structure $TellAgent(f(1,2))$, and then in evaluating the function $Agent$ on this value, which

```
1   val bach_random_choice = new Random()
2
3   def run_one(agent: Expr):(Boolean,Expr) = {
4
5      agent match {
6
7         ...
8
9         case bacht_ast_agent("+",ag_i,ag_ii) =>
10           {  var branch_choice = bach_random_choice.nextInt(2)
11              if (branch_choice == 0)
12                { run_one( ag_i ) match
13                     { case (false,_) =>
14                          { run_one( ag_ii ) match
15                               { case (false,_) => (false,agent)
16                                 case (true,bacht_ast_empty_agent())
17                                    => (true,bacht_ast_empty_agent())
18                                 case (true,ag_cont)
19                                    => (true,ag_cont)
20                               }
21                          }
22                       case (true,bacht_ast_empty_agent())
23                         => (true,bacht_ast_empty_agent())
24                       case (true,ag_cont)
25                         => (true,ag_cont)
26                     }
27                }
28              else
29                { run_one( ag_ii ) match
30                     { case (false,_) =>
31                          { run_one( ag_i ) match
32                               { case (false,_)
33                                    => (false,agent)
34                                 case (true,bacht_ast_empty_agent())
35                                    => (true,bacht_ast_empty_agent())
36                                 case (true,ag_cont)
37                                    => (true,ag_cont)
38                               }
39                          }
40                       case (true,bacht_ast_empty_agent())
41                         => (true,bacht_ast_empty_agent())
42                       case (true,ag_cont)
43                         => (true,ag_cont)
44                     }
45                }
46           }
```

Figure 2.9: BachT simulator: non-deterministic choice

results in the structure $CalledAgent(TellAgent(f(1,2)))$. It is that result which is assigned to P.

Strictness and Lazyness

Scala is a strict language that eagerly evaluates expressions. However there are cases in which it is desirable to postpone the evaluation of expressions, for instance to handle recursive definitions of agents. To that end, Scala proposes two basic mechanisms: call-by-name of arguments of functions and so-called thunks. To understand these two concepts, let us consider the following function:

```
def bacht_exec_all(agent: Expr):Boolean = {

    var failure = false
    var c_agent = agent
    while ( c_agent != bacht_ast_empty_agent() && !failure ) {
       failure = run_one(c_agent) match
            { case (false,_)          => true
              case (true,new_agent)  =>
                 { c_agent = new_agent
                   false
                 }
            }
        bb.print_store
        println("\n")
    }

    if (c_agent == bacht_ast_empty_agent()) {
       println("Success\n")
       true }
    else {
       println("Failure\n")
       false }

}
```

Figure 2.10: Bach simulator: main loop

```
def doubleFirst(x: Int, y: Int) = x + x
```

It returns the double of its first argument, regardless of the value of the second one (which is not used). Suppose we want to evaluate $doubleFirst(3+4,10+20)$. Using an eager strategy, Scala evaluates the two arguments and then computes the double of the first one. As a result $10+20$ is computed although the result is not used.

Let us slightly modify the definition of the $doubleFirst$ function, as follows:

```
def doubleFirstLazy(x: Int, y: => Int) = x + x
```

The first argument is passed using the call-by-value strategy. As for the $doubleFirst$ function, it is evaluated whenever the function is called.

In contrast, the second argument is passed using the call-by-name strategy. Accordingly, it is evaluated when needed and thus in our example not evaluated at all. Such a strategy is particularly useful to code *if-then-else* expressions for which one part only is evaluated according to the evaluation of the condition:

```
def myIf[A](cond: Boolean, onTrue: => A, onFalse: => A):
    A = { if (cond) onTrue else onFalse  }
```

For instance, using the following definition of $myDiv$, the evaluation of $myDiv(0)$ leads only to evaluate 1.

```
def myDiv(x: Float): Float = { (myIf[Float](x != 0, 1/x, 1))(x) }
```

However one step further needs to be made to handle recursive expressions that we want to evaluate step by step. In that case, so-called thunks are used. They amount to consider functions requiring no arguments, as in the following definition:

```
1 def myIf[A](cond: Boolean, onTrue: () => A, onFalse: () => A):
2       A = { if (cond) onTrue() else onFalse() }
```

Note that the arguments $onTrue$ and $onFalse$ are now functions that take no arguments and return expressions, rather than just being expressions themselves. Note also that according to the syntax of Scala $onTrue()$ can also be rewritten as $onTrue\ apply$.

To conclude this point, it is possible to delay the evaluation of val-declared expression by using the $lazy$ keyword, such as in

```
1 lazy val recursiveExpression = ... recursiveExpression ...
```

Programming Interface

To embody Bach in Scala, two main issues must be tackled: on the one hand, how data is declared, and, on the other hand, how agents are declared.

Data. As regards data, the trait SI_Term is defined to capture si-terms. Concrete si-terms are then defined as case classes of this trait. For instance in order to manipulate $f(1,2)$ in one of the primitives (tell, ask, ...) the following declaration has to be made:

```
1 case class f(x:Int, y: Int) extends SI_Term
```

Similarly, si-terms can be declared as in

```
1 case class a() extends SI_Term
```

However that leads to use parentheses everywhere as in $tell(a())$. To avoid that a $Token$ class has been defined as a case class of SI_Term. It takes as argument a string so that token a can be declared as

```
1 val a = Token(''a'')
```

Accordingly, a may now be used without parentheses, as in $tell(a)$.

Agents. The main idea for programming agents is to employ constructs of the form

```
1 val P = Agent { ( tell(f(1,2)) + tell(g(3)) ) || (tell(a) + tell(b)) }
```

which encapsulate a Bach agent inside Scala definitions. The *Agent* object is the main ingredient to do so. It is defined as an object with an apply method as follows

```
object Agent { def apply(agent: BSC_Agent) = CalledAgent(() => agent) }
```

It thus consists of a function mapping a *BSC_Agent* into the Scala structure `CalledAgent` taking a thunk, which consists of a function taking no argument and returning an agent. As we saw above, this is needed to treat in a lazy way recursively defined agent.

The *BS_Agent* type is in fact a trait equipped with the methods needed to parse Bach composed agents. Technically, it is defined as

```
trait BSC_Agent { this: BSC_Agent =>

  def *(other: => BSC_Agent) = ConcatenationAgent( () => this, other _)
  def ||(other: => BSC_Agent) = ParallelAgent( () => this, other _)
  def +(other: => BSC_Agent) = ChoiceAgent( () => this, other _) }
```

As ; is a reserved symbol in Scala, sequential composition is in fact rewritten with the $*$ symbol. The definition of the composition symbol $*$, || and + employs Scala facility to postfix operations. Using the above definitions, a construct of the form $tell(t) + tell(u)$ is interpreted as the call of method + to $tell(t)$ with argument $tell(u)$.

It is worth observing that the composition operators take agent arguments with call-by-name and deliver structures using thunks, namely functions without arguments to agents.

2.6.2 Implementation of the Domain Specific Language

The implementation of the domain specific language is based on the same ingredients as the implementation reported in Section 2.5: (i) how data is handled, (ii) what internal structures are needed and (iii) how computations are simulated. In fact, data is handled in the store exactly as in the interpreter of Section 2.5. Computations are also simulated in the same way. Internal structures need just to be slightly adapted.

Primitives are dealt with individually and not by means of a general `ast_primitive`. In a way similar to Agent, a *tell* primitive is interpreted as a function that return a *TellAgent* case class extending the trait BSC_Agent. This is achieved technically by the following code:

```
object tell { def apply(siterm: SI_Term) = TellAgent(siterm) }
    case class TellAgent(si:SI_Term) extends BSC_Agent {
       override def bsc_toString: String = {
         "tell(" + si.bsc_toString + ")" } }
```

Ask, nask and get primitives are of course handled similarly.

Moreover the generic `ast_agent` of Section 2.5 is also decoupled in different structures to cope with sequential composition, parallel composition and non-deterministic choice.

```
case class ConcatenationAgent(left: () => BSC_Agent, right: () =>
            BSC_Agent) extends BSC_Agent

case class ParallelAgent(left: () => BSC_Agent, right: () =>
            BSC_Agent) extends BSC_Agent

case class ChoiceAgent(left: () => BSC_Agent, right: () =>
            BSC_Agent) extends BSC_Agent
```

However, besides this rather syntactic rewriting, they are used in the same way as `ast_agent` for the Bach interpreter. The complete implementation is provided as a package available at [14]. It is subsequently referred to as BSCALA for Bach in Scala.

2.6.3 Back to the Altar Restaurant

To conclude this Section, let us see how the modelling of the Altar restaurant of Section 2.4, written in Bach can be recoded in BSCALA.

Top Goal

The top goal

```
Client(manel) || Client(alice) || Client(emma) || Cook(john) || Waiter(tom)
```

is rewritten first by declaring the si-terms *manel*, *alice*, ..., then by naming the clients and putting them in parallel to form the *Restaurant* agent, which is finally evaluated by calling an instance of the *BSC_Runner* interpreter. This is achieved by the following code.

```
val manel = Token("Manel")
val alice = Token("Alice")
val emma  = Token("Emma")
val john  = Token("John")
val tom   = Token("Tom")

val Manel = Agent { Client(manel) }
val Alice = Agent { Client(alice) }
val Emma  = Agent { Client(emma) }
val John  = Agent { Cook(john) }
val Tom   = Agent { Waiter(tom) }

```

```
13 val Restaurant = Agent { Manel || Alice || Emma || John || Tom }
14
15 val bsc_executor = new BSC_Runner
16 bsc_executor.execute(Restaurant)
```

Modelling a Client

Clients are modelled by mimicking the Bach description. For instance, the following Bach code

```
1 Client(id) = LogIn(id); EnjoyAltar(id); LogOut(id)
```

is rewritten in BSCALA as

```
1 def Client(id:SI_Term) = Agent {LogIn(id) * EnjoyAltar(id) * LogOut(id)}
```

Similarly, the recursive procedure *HandleOrders* coded in Bach as

```
1 HandleOrders(id,t) =    MakeOrder(id,t); HandleOrders(id,t)
2                       + CheckOrder(id,t,prepared); HandleOrders(id,t)
3                       + CheckOrder(id,t,served); HandleOrders(id,t)
4                       + CheckOrder(id,t,in_progress); HandleOrders(id,t)
```

is rewritten in BSCALA as follows

```
1 def HandleOrders(id:SI_Term,t:SI_Term): BSC_Agent = {
2    Agent {  ( MakeOrder(id,t) * HandleOrders(id,t) )
3           + ( CheckOrder(id,t,prepared) * HandleOrders(id,t))
4           + ( CheckOrder(id,t,served) * HandleOrders(id,t) )
5           + ( CheckOrder(id,t,in_progress) * HandleOrders(id,t))
6          }}
```

Modelling Cooks and Waiters

Modelling a cook or a waiter is done in a similar way. For instance, the following Bach declarations

```
1 Cook(id_cook)     = CookForCustomer(id_cook,manel); Cook(id_cook)
2                   + CookForCustomer(id_cook,alice); Cook(id_cook)
3                   + CookForCustomer(id_cook,emma); Cook(id_cook)
4
5 Waiter(id_waiter)= WaiterForCustomer(id_waiter,manel); Waiter(id_waiter)
6                   + WaiterForCustomer(id_waiter,alice); Waiter(id_waiter)
7                   + WaiterForCustomer(id_waiter,emma); Waiter(id_waiter)
```

are coded as follows

```
def Cook(id:SI_Term): BSC_Agent =
    Agent {      ( CookForCustomer(id,manel) * Cook(id) )
               + ( CookForCustomer(id,alice) * Cook(id) )
               + ( CookForCustomer(id,emma)  * Cook(id) ) }

def Waiter(id_waiter:SI_Term): BSC_Agent =
    Agent {      ( WaiterForCustomer(id_waiter,manel) * Waiter(id_waiter))
               + ( WaiterForCustomer(id_waiter,alice) * Waiter(id_waiter))
               + ( WaiterForCustomer(id_waiter,emma)  * Cook(id_waiter)) }
```

The whole code is available in Appendix A.

2.7 Closing Remarks

In summary, we introduced the coordination language Bach laying down the foundation for our research by elucidating key concepts like si-terms, primitives, and agents. We illustrated these notions through a tangible example that influenced the trajectory of our thesis research.

The next chapter explores the first major extension of the Bach language: the Anim-Bach coordination model. We will introduce this model in detail, along with a new workbench called Scan, designed to animate Anim-Bach. This extension represents an important step forward in our exploration of Bach's coordination features, paving the way for exciting new directions in our continuing research.

Part II

Animating Programs

CHAPTER 3

ANIMATING PROGRAMS WITH SCAN AND ANIM-BACH

This chapter addresses the need for a workbench designed to reason about programs written in the Linda-like language, Bach, which includes inter-agent communication primitives. While various coordination models have been proposed, emphasis has not been placed on the practical construction of programs in these languages and on ensuring that programs model systems accurately.

Building on previous work on Bach, we present our workbench with three main objectives: to enable users to understand step-by-step execution of programs, to connect agents to animations to model system evolution, and to facilitate property verification through model checking and trace generation. The workbench Scan *is designed to be user-friendly, in particular, by maintaining a direct relationship between user input and internal representation. The rush-hour puzzle, is used as a practical example to illustrate the concepts. In this follow-up, we present the tool functionalities, the coordination language and temporal logic used, and briefly cover the tool's implementation.*

3.1 Opening

The idea of separating the interactional and computational aspects of software components has been advocated by Gelernter and Carriero in their work [54]. They introduced the Linda model [34], which consists of inter-agent communication primitives that can be added to various programming languages. As explained before, these primitives include operations for process creation, and data manipulation (addition, deletion, and presence/absence testing) in a shared data-

space. Since then, several coordination models, also known as coordination languages, have been proposed. However, while many research efforts have focused on proposing new languages, semantics, and implementations, few articles have addressed the practical concerns of building programs in coordination languages and ensuring that programs accurately represent the intended models.

In this chapter, we address these problems by building upon Bach to present an extension embodying animations features together with a workbench to reason on programs. The main goal of this workbench is threefold:

(i) To facilitate the understanding of Bach instructions by enabling step-by-step execution and by visualizing modifications on the shared space. This helps users comprehend how suspended processes can be released.
(ii) To enhance the modelling of real systems in Bach by connecting agents to animations that represent the evolution of the system. This enables users to better understand how agents interact in the modelled system.
(iii) To enable property checking through model checking of temporal logic formulas and the production of traces that serve as evidence for the validation of these formulas.

The development of the workbench is guided by two key properties:

(i) Simplicity of deployment and usage. The workbench is designed as a standalone executable file that can be launched through a simple command line. It incorporates a straightforward process algebra that focuses on core coordination and animation features, relieving users from dealing with additional complexities commonly found in commercial systems.
(ii) Maintaining a direct relationship between the user's input and the internal representation of the program. This property allows users to have a better understanding of the actual computations performed by the workbench and facilitates the generation of meaningful traces.

Given these objectives and properties, the workbench is dedicated to creating a practical and user-friendly environment that facilitates the process of reasoning about programs written in the coordination language Bach and its extended version. The rest of this chapter is organized as follows. In Section 3.2, we will illustrate these concepts with a case study on "Rush Hour". In Section 3.3, we explain the coordination language and temporal logic used in the tool. Section 3.4 covers the features of our workbench and gives a complete view of the tool. Next, we present our implementation of Scan. Before concluding this chapter, we also offer a brief comparison with related work.

3.2 Rush Hour Puzzle: A Case Study of a Popular Game Worldwide

In order to illustrate the concepts of our coordination language Anim-Bach, we will use a well-known board game called Rush Hour puzzle. This game [63], shown in

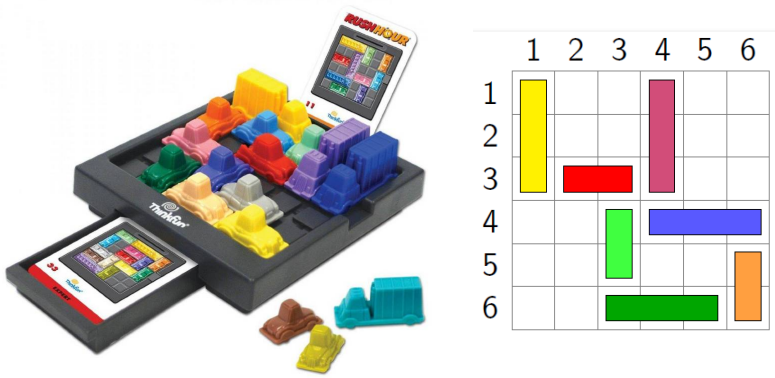


Figure 3.1: Rush hour puzzle's game

Figure 3.1, consists of moving cars and trucks on a 6×6 grid, according to their direction, such that the red car can exit. It can be formulated as a coordination problem by considering cars and trucks as autonomous agents that have to coordinate on the basis of free places.

3.3 Anim-Bach, Coordination Model

3.3.1 Data Definition

As illustrated in Chapter 2, the Bach language [43, 65] utilizes four basic operations to manipulate information: $tell$ for adding information to a shared space, ask for checking if the information is present, $nask$ for checking if the information is absent, and get for checking the presence and removing one occurrence. In the simplest version, called *BachT*, the information consists of atomic si-terms, and the shared space, known as the store, is a multiset of such si-terms.

However, while this basic setup is theoretically sufficient for coding many applications, it can be too elementary in practice. Therefore, more structured pieces of information are introduced they take the form $f(a_1, \cdots, a_n)$, where f is a functor and $a_1, \ldots, a_n$ are si-terms. They use user-defined sets. These sets are defined by associating an identifier with an enumeration of elements, as exemplified in:

```
1    eset Cols = { 1, 2, 3, 4, 5, 6}.
2         Rows = { 1, 2, 3, 4, 5, 6}.
```

Definition 3. *Sets are defined by the keyword eset followed by a list of set declarations of the form:*

$$set\_identifier = set\_enumeration.$$

where set_identifier is a string of letters and digits, starting with an upper case letter,

and where set_enumeration is a comma-separated list of strings composed of letters and digits, surrounded by the { and } brackets.

Example 4. *In order to illustrate those concepts, let us consider the rush hour puzzle game [63], shown in Figure 3.1. Two sets are required: one to specify the columns and the rows (ranging from 1 to 6) and one to specify the colors of the vehicles.*

```
1    eset RCInt  =  { 1, 2, 3, 4, 5, 6 }.
2         Colors =  { yellow, green, blue, purple, red, orange }.
```

In this example, free places of the game are represented by the si-terms *free(i,j)* with i a row and j a column. The extension of Bach we shall define, which is subsequently called Anim-Bach, additionally uses mapping equations as rewriting rules, from left to right, with the aim of progressively reducing an expression into a set element.

Example 5. *As an example, in the Rush Hour problem, assuming that trucks take three cells and are identified by the upper and left-most cell they occupy, the operation* `down_truck` *determines the cell to be taken by a truck moving down:*

```
1   map down_truck : RCInt -> RCInt.
2   eqn down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.
```

Example 6. *Besides down_truck, our modelling of the Rush Hour problem requires specifying a similar mapping down_car for cars (taking two cells instead of three). It is also convenient to specify two additional mappings, right_truck and right_car corresponding to horizontal movements. Finally, pred and succ mappings are defined to reflect the predecessor and successor functions.*

```
1   eqn down_car(1) = 3.    down_car(2) = 4.    down_car(3) = 5.
2       down_car(4) = 6.    down_truck(1) = 4.  down_truck(2) = 5.
3       down_truck(3) = 6.
4
5       right_car(1) = 3.   right_car(2) = 4.   right_car(3) = 5.
6       right_car(4) = 6.   right_truck(1) = 4. right_truck(2) = 5.
7       right_truck(3) = 6.
8
9       pred(2) = 1. pred(3) = 2. pred(4) = 3. pred(5) = 4. pred(6) = 5.
10      succ(1) = 2. succ(2) = 3. succ(3) = 4. succ(4) = 5. succ(5) = 6.
```

It is worth noting from this example that mappings may be partially defined. The responsibility is placed on the programmer to use them only when they are defined. Map rewritings are declared using the `eqn` keyword along with equations of the form $s = t$, where s and t are si-terms. These mapping equations establish a rewriting relation that transforms si-terms into other si-terms, denoted by $\rightsquigarrow$.

IDENTIFIERS

lid ::= any string of letters and digits starting with a lower case letter
uid ::= any string of letters and digits starting with an upper case letter
id ::= any string of letters and digits starting with a letter

SI-TERMS

$f ::= lid$	function name
$si ::=$	si-terms
$\quad lid$	$\quad$ token
$\quad f(si_1, \cdots, si_n)$	$\quad$ structured term

SETS

$sname ::= id$	set name
$seq ::= sname = \{si(,si)*\}.$	set equation
$sdefs ::=$ set $seq(seq)*$	set definitions
$setprod ::= sname(\times sname)*$	set product

MAPS

$mname ::= lid$	map name
$msign ::=$	map signature
$\quad$ map $mname : setprod \rightarrow sname.$	
$meq ::=$	map equation
$\quad mname(si_1, \cdots, si_n) = si.$	
$meqs ::=$	equations definition
$\quad$ eqn $meq(meq)*$	
$mdecl ::=$	map declaration
$\quad msign\, meqs$	

Figure 3.2: Syntax of data

To summarize this section, we assume that a series of sets, mappings, and a set of structured pieces of information, denoted as $\mathscr{I}$, have been defined. With the help of the mapping definitions, we also assume the existence of a rewriting relation $\rightsquigarrow$ that transforms any mapping expression into a set element. The precise syntax is shown in Figure 3.2. With these components in place, we can now proceed with the definition of agents in Anim-Bach.

3.3.2 Primitives

Primitives of Bach Basics

In Anim-Bach, the primitives available are `tell`, `ask`, `nask`, and `get`, which operate on elements from $\mathscr{I}$. These primitives are similar to Linda's primitives but adapted to a constraint-like context. As a reminder, in the context of a store represented as a multiset of si-terms, the execution of the `tell(t)` primitive adds an occurrence of t to the store. The `ask(t)` and `get(t)` primitives check if t is present in the store, with the latter removing one occurrence. Conversely, the `nask(t)` primitive tests whether t is absent from the store.

Figure 3.3 specifies the transition rules for theses primitives. They use as configurations pairs of instructions, here consisting of the primitives under consideration, coupled with the contents of the store. The symbol E represents a terminated computation. Rule (T) expresses that a `tell` primitive can be executed in any store σ, and that its action has the effect of adding the considered si-term t to the store. Rule (A) states that an `ask` primitive can be executed in any store σ containing the si-term t, while leaving the store σ unchanged. Rule (G) operates similarly but with the difference of retrieving an occurrence of its associated si-term t. Finally, Rule (N) establishes that a `nask` primitive can be executed in any store σ not containing the si-term t, leaving store σ unchanged after its execution.

$$\textbf{(T)} \qquad \frac{t \rightsquigarrow u}{\langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$\textbf{(A)} \qquad \frac{t \rightsquigarrow u}{\langle\, ask(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$\textbf{(G)} \qquad \frac{t \rightsquigarrow u}{\langle\, get(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$\textbf{(N)} \qquad \frac{t \rightsquigarrow u, u \notin \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 3.3: Transition rules for the primitives in Anim-Bach

Note that, in these rules, before processing, a structured piece of information t is rewritten as u using the rewriting relation $\rightsquigarrow$.

Graphical Primitives

$$(\textbf{Gr}) \qquad \frac{p \in GPrim}{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 3.4: Transition rule for the graphical primitive

Anim-Bach also includes a set $GPrim$ of graphical primitives. They consist of

- a `place_at` primitive to place images,
- a `move_to` primitive to move images from their current position to another,
- a `hide` primitive to hide images and
- at `show` primitive to make images appear.

The operational semantics of these primitives is described by Rule (Gr) of Figure 3.4. Essentially this transition rule states that any graphical primitive succeed on any store. As a summary, Figure 3.5 provides a listing of the various primitives of Anim-Bach.

PRIMITIVES

cprim ::=	communication primitive
tell(si)	
ask(si)	
nask(si)	
get(si)	
aprim ::=	animation primitive
draw(*si*)	
move_to(si_1, si_2, si_3)	
place_at(si_1, si_2, si_3)	
hide(*si*)	
show(*si*)	

Figure 3.5: Syntax of primitives

3.3.3 Agents

By combining primitives using composition operators from concurrency theory, such as sequential composition, parallel composition, and non-deterministic choice, more complex agents can be created. Additionally, Anim-Bach introduces conditional statements of the form $c \rightarrow s_1 \diamond s_2$, which execute s_1 if the condition c evaluates to true, or s_2 otherwise. Conditions of type c can be constructed using classical logical operators: & for conjunction (AND), | for disjunction (OR), and !

for negation (NOT). Elementary conditions are formed by comparing set elements or mappings using equality (=) or inequality ($\neq$, $<$, $\leq$, $>$, $\geq$) operators. With these concepts in mind, the statements in the Anim-Bach language, referred to as agents, are generated by the following grammar:

$$A \quad ::= \quad Prim \quad | \quad Proc \quad | \quad A\,;A \quad | \quad A \,||\, A \quad | \quad A+A \quad | \quad C \rightarrow A \diamond A$$

in which $Prim$ denotes a primitive, $Proc$ a procedure call, and C a condition.

Procedures are defined by using the `proc` keyword, similar to how mappings are defined. A procedure name is associated with an agent. It is assumed that the defining agents are guarded, meaning that any procedure call is preceded by the execution of a primitive or can be transformed into such a form. For instance, the behaviour of a vertical truck can be defined as follows:

```
1  proc VerticalTruck(r: Rows, c: Cols) =
2     ( (r>1 & r<5) -> ( get(free(pred(r),c)); tell(free(succ(r),c);
3                        VerticalTruck(pred(r),c) )
4   +
5     ( (r<4)       -> ( get(free(down_truck(r),c)); tell(free(r,c));
6                        VerticalTruck(succ(r),c)) ).
```

To understand this code, it is essential to mention that the position of trucks is determined by the upper-left cell it occupies. Accordingly, the "VerticalTruck" procedure's parameters explicitly include the row and column numbers of this cell. When it comes to a vertical truck, it has the choice to move either one cell up or one cell down, which is symbolized by the "+" operator. The first option involves moving the truck one cell up, but for this to be valid, the row denoted as r must be greater than one (to avoid reaching the top row) and strictly less than five (to prevent exceeding the fifth row, as shown in Figure 3.1). Assuming these conditions are met ($r > 1 \& r < 5$), moving the truck up is a three-step process. Firstly, we must ensure that the cell above is unoccupied, which is checked using the si-term $free(pred(r), c)$) via the "get(free(pred(r),c))" primitive. Note that $pred(r)$ is effectively represented as $r-1$. Secondly, the cell vacated by moving the truck up must be marked as free. This is achieved by informing the corresponding $free$ si-term in the store through the "tell(free(succ(succ(r)),c))" operation. Here, $succ(r)$ is equivalent to $r+1$ due to a mapping. Finally, the truck procedure is called recursively with $pred(r)$ and c as the new coordinates for the upper-left cell it occupies.

The behaviour for the alternative movement, where the truck moves down by one cell, is analogous. Since r is assumed to belong to the set $RCInt = \{1, \cdots, 6\}$ and we don't perform a $pred$ operation, there's no need to verify that r is greater than or equal to 1. However, to move the cell downward, it's crucial to check that r is strictly less than 4 to prevent moving beyond the fourth row.

Example 7. *Animations can be added between the get and tell equations in order to better present the movement of the truck. This leads us to the following code:*

```
1  proc VerticalTruck(r: RCInt, c: RCInt) =
2    ( (r>1 & r<5) -> ( get(free(pred(r),c));
3                       tell(free(succ(succ(r)),c));
4                       VerticalTruck(pred(r),c))  )
5  +
6    ( (r>=1 & r<4)-> ( get(free(down_truck(r),c));
7                       tell(free(r,c));
8                       VerticalTruck(succ(r),c)) ).
```

The transition rules in Figure 3.6, define the operational semantics of complex statements. Configurations in this context consist of agents, representing the current state of running agents, and a multi-set of si-terms, representing the current state of the store. To indicate the termination of an agent's computation, a special symbol E is used as before. IT is considered as a fully computed agent. For the sake of uniformity, we refer to E as an agent. Intuitively, agents in the form of $(E;A)$, $(E||A)$, and $(A||E)$ are always rewritten as A.

$$\textbf{(S)} \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle A\,;\,B \mid \sigma\rangle \longrightarrow \langle A'\,;\,B \mid \sigma'\rangle}$$

$$\textbf{(P)} \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A \,||\, B \mid \sigma\rangle \longrightarrow \langle A' \,||\, B \mid \sigma'\rangle \\ \langle B \,||\, A \mid \sigma\rangle \longrightarrow \langle B \,||\, A' \mid \sigma'\rangle\end{array}}$$

$$\textbf{(C)} \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle A + B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle B + A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$\textbf{(Co)} \quad \frac{\models C, \langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\begin{array}{c}\langle C \rightarrow A \diamond B \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle \\ \langle !C \rightarrow B \diamond A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle\end{array}}$$

$$\textbf{(Pc)} \quad \frac{P(\overline{x}) = A, \langle A[\overline{x}/\overline{u} \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\langle P(\overline{u}) \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}$$

Figure 3.6: Transition rules for the operators in Anim-Bach

The rules presented in Figure 3.6 follow the classical concurrency setting:

- Rule (S) states that a sequentially composed agent can perform a computation step if its first subagent can do so.
- Rule (P) defines the parallel operator in an interleaving manner.

- Rule (C) requires the non-deterministic choice to be made based on the first step of its components.
- Rule (Co) specifies that the conditional instruction $C \rightarrow A \diamond B$ behaves as A if the condition C can be evaluated to true and as B otherwise.
- Rule (Pc), makes procedure call $P(\overline{u})$ behaves as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

To conclude, Figure 3.7 provides the syntax of complex agents.

PROCEDURES

$pn ::= uid$	procedure name
$pc ::= pn(si_1, \cdots, si_n)$	procedure call
$pharg ::= lid : sname$	arguments in procedure head
$ph ::=$	head of procedure declaration
$\quad pn(pharg_1, \cdots, pharg_n)$	
$peq ::= ph = ag.$	procedure equation
$pdecls ::=$ proc $peq\,(peq)*$	procedure declarations

CONDITIONS

$compop ::=$	
$\quad = \mid ! = \mid < \mid <= \mid > \mid >=$	comparison operators
$elmc ::= si_1\ compop\ si_2$	elementary conditions
$boolop ::= \& \mid$ '|' $\mid !$	boolean operators
$c ::=$	conditions
$\quad elmc$	
$\quad c_1\ boolop\ c_2$	

AGENTS

$ag ::=$	agents
$\quad cprim$	communication primitive
$\quad aprim$	animation primitive
$\quad pc$	procedure call
$\quad ag_1 ; ag_2$	sequential composition
$\quad ag_1 \mid\mid ag_2$	parallel composition
$\quad ag_1 + ag_2$	non-deterministic choice
$\quad c \rightarrow ag_1 \diamond ag_2$	conditional agent

Figure 3.7: Syntax of agents

3.3.4 Animations

Two issues need to be addressed in order to create animations: the first is to describe the scene to be represented, while the second uses primitives for image placement, visibility control, and movement. To define a scene, we specify the size of the background image and a collection of images to be included.

Example 8. *To get the picture of Figure 3.8, we can define the scene as follows:*

```
scene (640,640)
    background  = loadImage(Images/the_background_img.png).
    GreenTruck  = loadImage(Images/gtruck.jpg).
    Yellowtruck = loadImage(Images/ytruck.jpg).
```

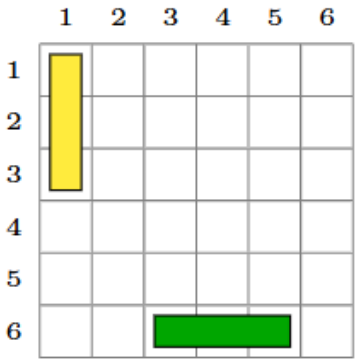


Figure 3.8: Animation of Rush hour puzzle's game

The images are manipulated by means of the graphical primitives already announced but left partially specified. Their coordinates are expressed in pixels with respect to the canvas, with (0,0) being the upper-left corner of the canvas. They consist of

- `place_at(i, x, y)`: to place image identified by i at the coordinates (x, y)
- `move_to(i, x, y)`: to move image identified by i from its current position to the new coordinates (x, y)
- `hide(i)`: to hide image identified by i
- `show(i)` : to make appear image identified by i

As already mentioned, primitives are added to the `tell, get, nask,` and `ask` primitives in the definition of Anim-Bach. As announced in Section 3.3.2, it is worth observing that the map constructs (introduced earlier) allow the declaration of coordinates in a symbolic manner, making it easy to specify the position of images.

Example 9. *By using* (x, y) *to denote respectively the column and the row occupied by the leftmost uppermost cell occupied by a vehicle, one may translate coordinates expressed logically in rows and columns of the grid in pixel positions as follows:*

```
eqn x_vehicle(1) = 55.  x_vehicle(2) = 130. x_vehicle(3) = 210.
    x_vehicle(4) = 285. x_vehicle(5) = 360. x_vehicle(6) = 435.

    y_vehicle(1) = 80.  y_vehicle(2) = 155. y_vehicle(3) = 230.
    y_vehicle(4) = 310. y_vehicle(5) = 385. y_vehicle(6) = 460.
```

Example 10. *Using such declarations, one may then place the yellow truck by executing:* `place_at(yellow_truck,rhScene,x_vehicle(1),y_vehicle(1)).` *More generally, placing a truck at a place can be coded as follows:*

```
proc PlaceTruck(r: RCInt, c: RCInt, p: Colors) =
(p = yellow -> place_at(yellow_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = green -> place_at(green_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = blue -> place_at(blue_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = purple ->place_at(purple_truck,rhScene,x_vehicle(c),y_vehicle(r))).
```

Example 11. *Moving cars and trucks can be coded similarly, as illustrated below.*

```
proc MoveTruck(r:Rows, c: Cols, p: Colors) =
(p = yellow -> move_to(yellow_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = green -> move_to(green_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = blue -> move_to(blue_truck,rhScene,x_vehicle(c),y_vehicle(r)))
 +
(p = purple -> move_to(purple_truck,rhScene,x_vehicle(c),y_vehicle(r))).
```

The interested reader will find the complete coding of the rush-hour problem in appendix B.

3.3.5 Logic Formulae

With the Anim-Bach language developed so far, the programmer can hope to model a system and, as we shall see in a few pages with the Scan workbench, can hope to incrementally execute it step-by-step or illustrate its execution through animations. However this is not enough to solve puzzles like the rush hour problem. Indeed with what is described until now, one can in the best case follow cars and trucks moving in an uncontrolled manner and dream that magically the red car will exit.

More generally, one may want to be able to state formulae and prove that they hold. This is exactly what model checking aims at. In this section, we will first provide a simple introduction to model checking and then specify which formulae can be written with Anim-Bach.

An Introduction to Model Checking

As defined in [8] by Baier and Katoen, model checking is an automatic verification technique that systematically explores the states of a system to determine whether a given specification holds. Practically this involves two tasks: on the one hand, the creation of a model that represents the system behaviour and, on the other hand, the specification of properties, typically expressed in logic. Model checking then aims at establishing that the model meets the logic formulae.

In the simple context in which we place our work, model checking can be seen as characterized by several key features:

- **Finite-State Systems:** The technique is particularly effective for finite-state systems, where the number of possible states is manageable. Note that techniques such as state space reduction are employed to handle larger systems.
- **Automatic Verification:** Model checking provides an automated way to verify system properties without requiring manual inspection. This is achieved through algorithms that systematically explore the state space of the model. Very often both the model and the logic formula are transformed in finite state automata and the verification is performed in that setting. In case the verification fails a counter-example is produced. Otherwise a successful message is returned to the user.
- **State Space Exploration:** Model checking involves exploring the state space of the model, which may include exhaustive search methods. Techniques such as breadth-first search or depth-first search are commonly utilized to traverse the state space.
- **Temporal Logic Specifications:** Properties of the system are expressed in temporal logic, which allows reasoning about the timing of events. Typically, Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are used since they provide rich languages to specify temporal properties. Both are formalisms that allow to express properties about states that evolve over time. Hence the qualification “temporal”. Of the two, LTL is the simplest to use and check but, as one may infer, it is also the less expressive.

Linear Temporal Logic

Aiminig at a proof-of-concept, we shall place our work under the simplest LTL framework. This logic extends propositional logic by introducing temporal operators that allow reasoning about how propositions evolve in a linear sequence of time points or states. LTL enables the expression of statements such as:

- "**Globally**"(**G**): This specifies that a property is always true in the future across all states. Formally, $G\varphi$ means φ holds in the current state and every subsequent state.
- "**Finally**" (**F**): This indicates that a property will eventually become true at some point in the future. Formally $F\varphi$ means that φ holds at the current or subsequent state.
- "**Next**" (**X**): This ensures that a property is true in the immediate next state. As a result $X\varphi$ evaluates φ only for the next time step.
- "**Until**" (**U**): This allows for the expression of properties where one property holds until another becomes true. For instance, $\varphi U \psi$ means φ must hold in all states up to the point where ψ becomes true. Note that the formula also implies that ψ holds at some point.

It is worth observing that the last two operators are more fundamental as the first two operators can be expressed with negation and the until operator:

$$\begin{aligned} F\varphi &\equiv true U \varphi \\ G\varphi &\equiv \neg F \neg \varphi \end{aligned}$$

LTL Formulae in Anim-Bach

As Anim-Bach already allows to model systems, to be able to use model checking, it remains to specify the propositional formulae we shall employ and then the temporal operators we shall use.

Propositional Formulae. In our coordination context, a key point is to determine which si-terms occur on the store. The following definition aims at capturing this idea.

Definition 4. *Given a structured piece of information t, we define #t as the number of occurrences of t in the store.*

This notation serves as a fundamental building block for constructing basic propositional formulae, allowing for a clear representation of system state and behaviour. These formulae utilize equalities or inequalities that combine algebraic expressions involving integers and the count of occurrences of structured pieces of information.

Definition 5. *We define as basic propositional formulae, equalities, or inequalities combining algebraic expressions involving integers and a number of occurrences of structured pieces of information.*

Example 12. *One example of a basic formula is #free(1; 1) = 1, which indicates that the cell at coordinates (1; 1) is free.*

EXPRESSIONS ON SI-TERMS

int ::= positive integer	
arop ::= $+ \mid -$	arithmetic operators
cardsi ::= *#si*	number of occurences of *si*
exp ::=	expressions on si-terms
cardsi	
int	
exp_1 *arop* exp_2	
bf ::=	elementary basic formula
exp compop exp	
pf ::=	propositional formula
bf	
bf_1 *boolop* bf_2	
tf ::=	temporal formulae
pf	
Next *tf*	
pf Until *tf*	
Reach *pf*	

Figure 3.9: Syntax of logical formulae

Definition 6. *Propositional state formulas are constructed by combining these basic formulae using classical propositional connectors, thus adhering to the principles outlined in [8].*

As specific cases, *true* and *false* represent formulae that are always *true* and *false*, respectively. These formulae are actually shorthand notations for $p \vee \neg p$ and $p \wedge \neg p$, where p is a basic propositional formula (*PF*).

Temporal Formulae. Building upon the next and until operators, the temporal formulae are defined as follows.

Definition 7. *The temporal logic fragment utilized with* Anim-Bach *is defined by the grammar:*

$$TF \quad ::= \quad PF \quad | \quad Next\ TF \quad | \quad PF\ Until\ TF$$

where PF represents a propositional formula.

Example 13. *As an illustration, consider a scenario where the red car leaving the grid is indicated by placing out on the store. A solution to the rush hour puzzle can be obtained by verifying the formula:*

$$true\ Until\ (\#out = 1)$$

As such, formulae are so frequent to use, we shall subsequently note them as

$$Reach(\#out = 1)$$

In other terms, we use *Reach* to denote the F operator mentioned above.

As a summary of this section on temporal logic, Figure 3.9 provides a recap of the syntax of the logical formulae.

3.4 Scan, Coordination Workbench

In line with Linda's approach, the Bach language utilizes a shared space to facilitate process coordination. This shared space enables the separation of time and space, which is a fundamental characteristic of data-based coordination languages [95]. Taking inspiration from the concept of a blackboard system [49], where a group of specialists collaboratively updates knowledge on a blackboard based on a given problem, Scan revolves around an interactive blackboard. Illustrated in Figure 3.10, the interface initially presents the current content of the shared space, allowing direct interaction through buttons such as `tell`, `get`, and `clear`. Additionally, the platform offers the ability to create four types of processes.

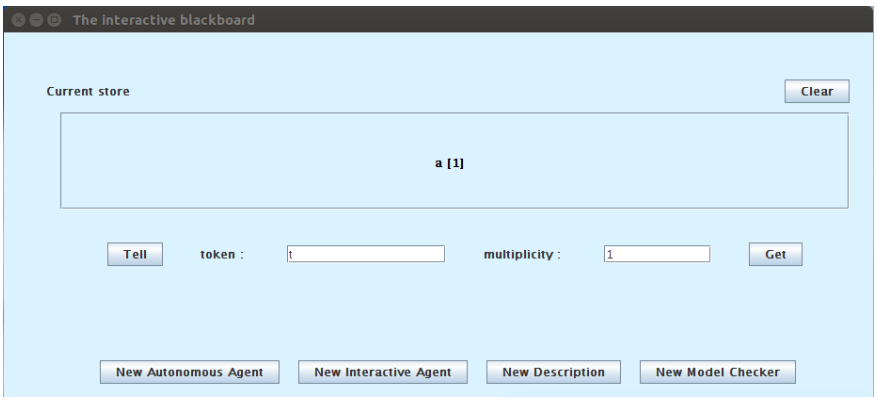
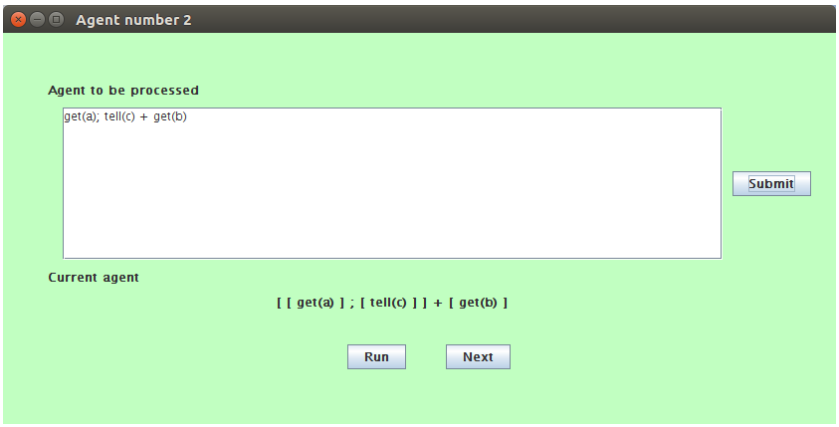


Figure 3.10: Interactive blackboard window

The first two processes, namely the "Autonomous Agent" and the "Interactive Agent", enable the user to input instructions in Anim-Bachand execute them. In Figure 3.11 (a), windows of the first type allow step-by-step computation by letting the user choose which primitives to execute. Conversely, as shown in Figure 3.11 (b), windows of the second type execute computations either in one run when the "run" button is activated or step-by-step when the "next" button is selected. In both cases, the Scan workbench determines (randomly) the primitives to be executed. These windows execute their operations in parallel, hence the term "agent" to indicate entities capable of concurrent activities.

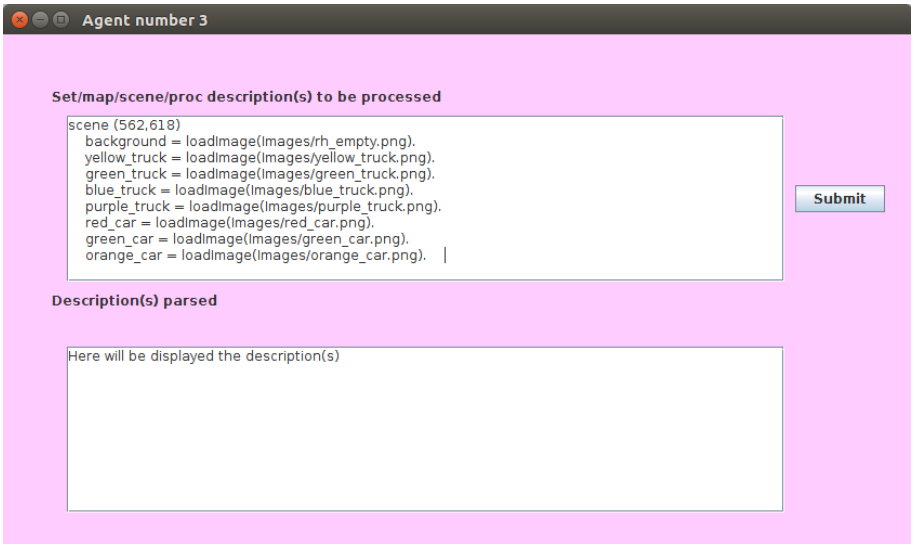
(a) Interactive agent window



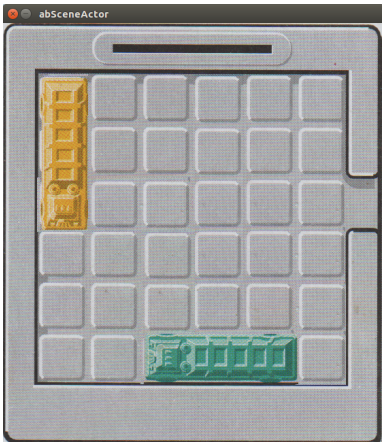
(b) Autonomous agent window

Figure 3.11: Interacting with the blackboard

While the interactive and autonomous agents are useful for debugging concurrent executions at a low level around the shared space, they provide limited insights into whether the code accurately reflects the programmer's intended model. Moreover, they offer excessive details regarding the main execution steps leading to a solution for the given problem. To address these limitations, Scan incorporates animations through a third type of process initiated by the "new description" button (refer to Figure 3.10). In Figure 3.12 (a), such animations are created by describing a scene using a set of pictures manipulated by primitives that insert them into the scene, control their visibility, and orchestrate their movements. These primitives allow the high-level drawing and animation of pictures, as demonstrated in Figure 3.12 (b). Notably, as these primitives can be embedded within instructions of autonomous agents, the concurrent execution of these agents provides dynamic simulations of the problem being addressed.



(a) Description window



(b) Scene window

Figure 3.12: Animation

While the graphical simulation of systems is appealing, it does not always offer a solution to the problem at hand, as demonstrated by the rush hour problem. Therefore, the Scan workbench introduces a fourth type of process through the "new model checker" button in Figure 3.10, which generates windows as depicted in Figure 3.13. These windows allow the verification of formulas written in the subset explained in Section 3.3.5, the identification of execution traces that satisfy the formulas, and the replay of these traces, including the primitives that generate animations.

Despite its simplicity, we believe that the Scan workbench fulfills the threefold objective stated in the introduction:

- By providing a view of the shared space and incorporating interactive and autonomous agents, users gain a better understanding of the execution of Bach programs.
- The animation features provide a high-level perspective on the actual computation process and an intuitive understanding of the problem modelling.
- The model checker capabilities enable the verification of properties and, in conjunction with animation features, allow graphical replay of executions as visual proofs.

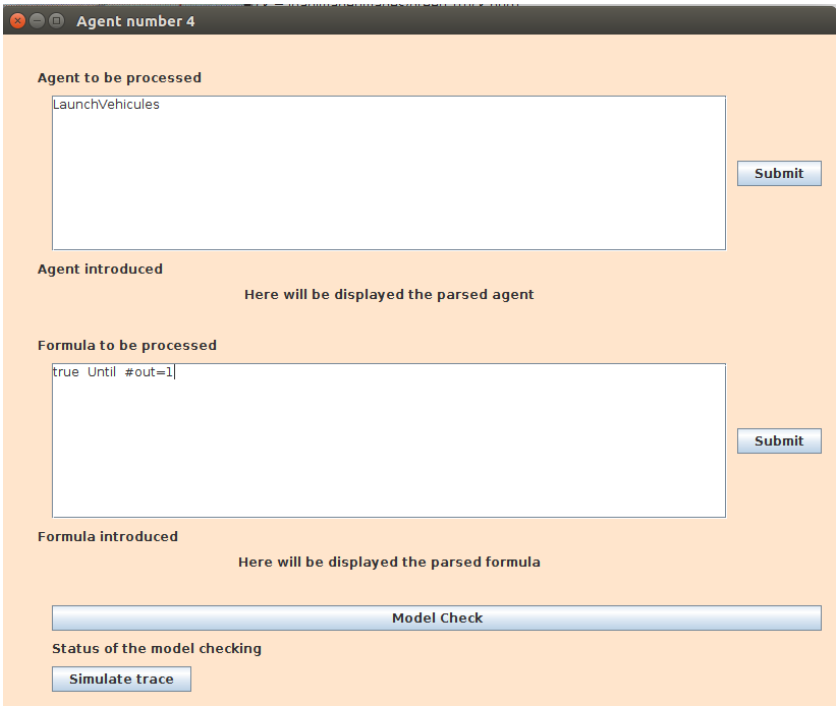


Figure 3.13: Verification Window

3.5 Implementation of Scan

The implementation of the Scan workbench leverages the Scala programming language [90] and is built on top of the Processing library [101].

As already mentioned in Chapter 2, Scala is a programming language that combines the strengths of object-oriented and functional paradigms and is equipped with a robust static type system. It compiles Scala source code into Java bytecode,

which facilitates seamless integration with Java libraries. Additionally, Scala offers powerful parsing capabilities. These characteristics make Scala well-suited for interpreting the Anim-Bach language, which, as demonstrated in the Chapter 2 in the Section 2.5, can be effectively described using recursive definitions.

Processing, on the other hand, is a graphical library primarily designed for teaching programming in a visual context, particularly to artists. While it is commonly used with a dedicated integrated development environment (IDE), Processing can also be employed as a Java library, which is the approach adopted by Scan. Central to Processing is the `draw` method, which is called multiple times per second (typically 60 times) and allows for the creation of animations by modifying various parameters such as image coordinates. The subsequent subsections will provide readers with an understanding of the key components and elements of our implementation.

3.5.1 Internal Representation of Data

Scala case classes provide an elegant mechanism for representing data in an internal manner while maintaining a strong connection to the textual representation in Anim-Bach. For example, following the outline already mentioned in the Chapter 2 in the Section 2.5.1, an abstract class called `AB_AG` is introduced to represent agents in Anim-Bach.

Case classes are then defined to represent specific types of agents, such as `AB_AST_Empty_Agent()` for the empty agent, `AB_AST_Primitive(primitive: String, stinfo: AB_SI_ELM)` for a primitive, and `AB_AST_Agent(op: String, agi: AB_AG, agii: AB_AG)` for a composed agent using the operator `op`, such as || for parallel composition, with two subagents `agi` and `agii`. Similar structures are utilized to represent code sets, structured pieces of information, map equations, and temporal logic formulae.

This approach ensures a close correspondence between the internal representation and the textual description in Anim-Bach. Consequently, unlike tools like mCRL2 [38], it becomes relatively straightforward to provide the user with informative messages that are directly linked to their written code.

3.5.2 Parsing Anim-Bach Constructs

As detailed in chapter 33 of [90], Scala provides tools for parsing languages. The key components for parsing are threefold:

- Firstly, parsers are treated as functions that consume input and produce a parse result. These parsers can be sequenced using the "~" operator. For example, consider the expression:

```
1     "tell(" ~ stInfo ~ ")"
```

In this case, Scala attempts to parse the string "`tell(`", followed by the result of the "`stInfo`" parser (which parses a structured piece of information), and finally the string "`)`". The result of this evaluation is an instance of the "`~`" class, which can be seen as a pair or, in this context, a triple of values.

- Secondly, Scala allows the application of functions to the result of a parser using the "`^^`" construction. For instance:

```
("[a-z][0-9a-zA-Z_]*").r ^^ {_.toString}
```

applies the toString function to the values that can be examined by the regular expression parser, transforming it into a string.

- thirdly, values can be examined through a case statement, which allows performing a matching, as illustrated as follows :

```
"tell("~stInfo~")" ^^ {
    case _ ~ sti ~ _  => AB_AST_Primitive("tell",sti) }
```

There a string composed of the string "tell(" followed by the result of the function `stInfo` followed by the string ")" is given as the corresponding threefold sequence of strings to be matched to the expression "`_ ~ sti ~ _`". It is worth noting that `sti` is actually a variable that is matched with the corresponding piece of structured information in case the matching is successful. The underscores denote different anonymous variables, which are respectively used to match the strings "tell(" and ")". In case the matching is successful, the value after the arrow "`=>`" is given as a result of the parsing. In the above example, a new case class is returned for the primitive tell with "`sti`" as a structured piece of information. It is worth noting that for expressively purposes, Scala permits to avoid explicitly writing the new statement (which would have been written in Java for instance).

Writing the parser follows from the above description by specifying the priorities of the operators. We subsequently take as an example the parsing of agents, other structures are being processed similarly.

Our choice of agents has been quite classical: we stipulate that the sequential composition binds more than the parallel composition, which itself binds more than the non-deterministic choice operator. As a result, assuming that functions `compositionPara` and `compositionSeq` are in charge of parsing agents composed in a parallel and sequential manner, the former can be defined from the latter as follows:

```
def compositionPara : Parser[AB_AG] =
    compositionSeq~rep(opPara~compositionPara) ^^ {
        case ag ~ List() => ag
        case agi ~ List(op~agii)  => AB_AST_Agent(op,agi,agii) }
```

The first case of the matching `ag ~ List()` instantiates `ag` to the result of a sequential-like agent which is followed by an empty list, namely which is followed

by an empty repetition of the parallel operator followed by a parallel-like agent. In the second case of the matching, `agi` represents the parsing of the sequential-like agent (similarly `ag` does for the first case) and `agii` represents the parsing of the repetition of the parallel operator followed by a parallel-like agent.

3.5.3 Store

Following the description in Chapter 2, the store in Scala is implemented as a mutable map. Initially, the store is empty, and for each told structured piece of information, it is updated by associating it with a number representing the occurrences of that information in the store. The implementation of primitives directly follows this idea. For example, when executing the tell primitive, denoted as `tell(t)`, the program checks if `t` is already present in the map. If it is, the number of occurrences associated with `t` is incremented by one.

On the other hand, if `t` is not in the map, a new association `(t,1)` is added. Similarly, the execution of `get(t)` involves checking if `t` exists in the map. If it does, the number of occurrences is decremented by one. If either of these conditions is not satisfied, the get primitive cannot be executed. The declaration of sets, map equations, and procedure definitions are also stored using maps or lists to preserve their information.

3.5.4 Simulator

The simulator functions by iteratively performing transition steps. In our implementation, this is accomplished through the `run_one` function, which accepts a parsed agent as input and returns a pair containing a boolean value and a parsed agent. The boolean value indicates whether a transition step has occurred. If a transition step takes place, the associated agent in the pair represents the resulting agent after the transition. However, if no transition step occurs, the simulator reports a failure, and the associated agent remains the same as the input agent.

The `run_one` function is recursively defined based on the structure of the argument, denoted as `ag`. When `ag` is a primitive agent, the function executes the corresponding primitive operation on the store. In the case of a sequentially composed agent, denoted as $ag_i; ag_{ii}$, the transition step begins by attempting to execute the first subagent, ag_i. If this execution succeeds and yields ag' as the resulting agent, the function returns $ag'; ag_{ii}$ if ag' is not empty. If ag' is empty, the function simply returns ag_{ii}. Importantly, if the transition step fails when applied to ag_i, i.e., if `run_one` applied to ag_i fails, the entire computation also fails.

The situation becomes more intricate when dealing with an agent composed of a parallel or choice operator. In both cases, it is not always appropriate to favour either the first or second subagent consistently. To avoid such bias, we introduce a boolean variable randomly assigned the value of 0 or 1. Based on this value, we initiate the evaluation of either the first or second subagent. If one evaluation fails, we proceed to evaluate the other subagent. If both evaluations fail, a failure is reported.

In the case of a successful evaluation for a parallel composition, we determine the resulting agent similarly as we did for sequentially composed agents. For a choice composition, the alternative that was successfully evaluated is simply selected.

The computation of a procedure call and a conditional statement follows the expected behaviour. With the one-step transition function implemented, the simulator primarily consists of a loop that continues executing as long as the current agent is non-empty and no failures occur.

3.5.5 Scene

The scene and its animation are implemented by using Processing as the underlying framework. When declaring a scene, specific declarations are introduced within the Processing's `setup` method. To create the illusion of image movement, an update is performed in the `draw` method of Processing, utilizing linear interpolation between the initial and final coordinates. Placing images, as well as hiding or showing them, is accomplished by modifying the relevant variables and attributes associated with the images.

3.5.6 Temporal Formulae

Scan temporal formulae are verified by means of a home-made program inspired by the techniques proposed in [104]. For more general temporal formulae, it essentially uses a limited depth-first search algorithm based on the simulator described in Section 3.5.4 with recursive reasoning on the temporal formulae. More concretely, the key function `check_lts` takes as arguments an integer I, a temporal formula F, an agent A, and a trace T. The first argument is the length of the remaining search allowed. The second and third arguments are the temporal formulas to be checked against the agent. The path consists of the trace prefix already computed. The function returns a boolean, stating whether the formula has been checked, together with a path, describing the last path explored. It provides an execution witness of the truth of the formula in case the returned boolean is true.

The `check_lts` function is coded by using a recursive reasoning on the formulae:

- if F is a propositional formula, then the current contents of the store should verify it. If this is the case, then `true` is returned together with the path P. Otherwise, `false` is returned together with P.
- if F is of the form *Next* TF and if I is strictly positive, then `check_lts` is successively called on the list of next possible agents returned by `run_one` with $I-1$ as integer, the agent produced by `run_one` as agent, TF as formula and P augmented with a reference to the computation step as path. In case one of these calls succeed, namely returns `true` with the associated path, then this result is returned. Otherwise or in case $I = 0$, then `false` is returned together with the path P.

- the case where *F* is of the form *PF Until TF* is treated similarly. Either *TF* holds on the current store, in which case `true` is returned together with *P*, or *PF* holds in the current store and there is one successor agent (explored as for the above case) for which *PF Until TF* holds. In this latter case, `true` is returned together with the discovered path. In case none of the two situations holds, then `false` is returned with the path *P*.

It is worth noting that the algorithm is not complete. If it returns `true` then the considered formula has been established and the returned path provides a witness execution that can be replayed. Otherwise, because of the limited depth-first search, `false` may be returned wrongly as the formula could have been proven by using a more exhaustive search. Nevertheless, such a simple algorithm is in practice already useful to establish formulae. Note also that, in case of success, the algorithm is sound because of the limited form of the temporal formulae considered in Scan which, in particular, does not involve negation in the definition of the *TF* formulae. This reflects that temporal logic formula is used here in an extensional way, where formulae are interpreted over concrete execution paths rather than over all possible behaviours.

Being simpler, *Reach* formulae allow for a more efficient algorithm. Indeed, they basically required to check that configurations enjoy the property they express. As a result, Reach formulae are checked by a breadth-first search algorithm that generates from a configuration the configurations obtained after one step and that, for each of them, tests whether the formula is met. Moreover, the contents of the store and the state of agents are memorized at each node of the graph in order to perform loop checking.

3.6 Related work

While numerous works in the field of coordination have introduced new languages, new semantics, and new implementations, only a few have addressed the practical aspects of building programs in coordination languages. A few notable exceptions, including the Extensible Coordination Tools [69], ReoLive [39], and TAPAs [32]:

The Extensible Coordination Tools (ECT) has been developed for the control-based coordination language Reo, a language quite different from Anim-Bach. ECT consists of a set of plug-ins for the Eclipse platform that provide graphical editing facilities of Reo connectors, the animation of these connectors as well as model checking based on constraint automata or a translation to the process algebra mCRL2 [38]. Although it is certainly less elaborated, our work differs in several respects. First, it deals with tuple spaces instead of connectors. Second, it allows grasping the modelling of real-life systems by connecting agents of Bach to animations at the application level. Consequently, although one may animate connectors in ECT, one cannot animate the modelling of the rush hour problem for instance, as we did with Anim-Bach. Finally, in contrast to our work, model checking in ECT does not preserve a one-to-one link with textual representations, in particular when mCRL2 is used. ReoLive is also dedicated to Reo. It proposes similar tools, but by

means of a set of web-based tools using ScalaJS. As a consequence, the above comparison with ECT also applies to ReoLive. TAPAs [32] is a tool developed essentially for CCSP with a plug-in for an extension of the Klaim coordination language. It allows to graphically specify systems and to verify their equivalence by means of bisimulations based equivalences (strong, weak, and branching) or decorated trace equivalences (weak and strong variants of trace completed trace, divergence sensitive trace, must, testing). It also allows model checking systems by using formulae of the μ-calculus. The two main differences of our work with TAPAs are, on the one hand, our concern for tuple-based coordination languages, and, on the other hand, the facilities offered by Anim-Bach for animations.

Declarative invariant assertions are proposed in [70] to detect inconsistencies in models expressed in the Peer model, a coordination model based on shared tuple spaces, messages and Petri nets. In addition to the fact that the Peer Model is quite different from Anim-Bach, our work differs in two main respects. On the one hand, assertions in [70] are verified at runtime, whereas our temporal formulae are checked statically. On the other hand, in contrast to our work, no animation facilities are provided. Although it includes facilities to view the evolution of the shared space, TUCSON [93] does not provide facilities to animate computations nor to model-check them.

Finally, a Linda workbench is presented in [48] with the goal of providing a simple tool that allows users to experiment with a Linda-inspired language. It is integrated with Netbeans and uses the JavaSpaces language, an extension of Java supporting Linda primitives. It is hence named JavaSpaces Netbeans. This workbench provides a tuple browser and a distributed debugger, including record facilities to replay a sequence of tuple space operations. Although our work provides facilities to explore and modify the tuple space, we do not provide debugging facilities. In contrast, however, we provide animation facilities as well as formulae verification facilities which are not included in JavaSpaces Netbeans.

3.7 Closing Remarks

In this chapter, we presented a workbench for reasoning about programs written in Linda-type coordination languages at three levels: (i) executing instructions step-by-step or automatically, while showing their impact on the shared space, (ii) illustrating calculations with animations and (iii) verifying properties using temporal formulas.

In the next chapter, we will present an extension of this workbench, to improve it in many aspects (modelling, animation, etc.), bearing in mind that the Scan version has been designed to be as simple as possible while incorporating key ideas.

CHAPTER 4

IMPROVING SCAN INTO ANEMONE AND MULTI-BACH

This chapter extends the developments of Chapter 3. While the overarching research objectives remain consistent, noteworthy enhancements have been made. At the process algebra level, our modifications include the introduction of generalized choices, primitives for handling processes as active data, manipulation of blackboard rules, and a revised description of scenes that now allows for multiple scenes, incorporating widgets as the primary elements.

The workbench has been refined to accommodate these new features and provide additional facilities. Notably, the interactive blackboard now integrates a filter button, enabling users to selectively choose elements within the shared space. Two novel windows, parallel to the interactive and autonomous agent windows, are introduced to facilitate the creation of new processes induced by their active data view. Furthermore, a new window type is proposed for rule management. The resulting workbench is referred to as Anemone*, and it can be accessed at the website [13], together with examples, videos, tutorials, and documentation.*

4.1 Opening

In this chapter, we present an extension of Chapter 3, the very one we presented at the Coordination'19 conference [63], with the aim of tackling the complex challenges of coordination and modelling in Linda-like languages.

This advanced language is called Multi-Bach. It is a coordination language enriched with essential primitives for system modelling and animation. Our research objectives remain the same, but we have made several revisions and introduced

new features. At the level of process algebra, we have integrated generalized choices and primitives to manipulate processes as active data. We will present these in a new coordination model in Section 4.2. In addition, we have restructured the way scenes are described, enabling the creation of multiple scenes and the use of multiple agents. Moreover, we have enhanced this description system, allowing us to incorporate widgets as key scene elements. This is explored in Section 4.3. Section 4.4 discusses how our implementation is extended and Section 4.5 compares our work to related work. Finally, Section 4.6 presents our conclusion.

4.2 Multi-Bach, Overview of the Coordination Model

4.2.1 Data Definition and Primitives

As done in Chapter 3, Multi-Bach assumes the existence of defined sets, mappings, variables, and si-terms together with the presence of a rewriting relation $\rightsquigarrow$ that transforms any mapping expression into a set element. Primitives include the *tell, ask, get* and *nask* primitives explained in Chapter 3. As we shall the latter, they will be enriched by other new primitives.

4.2.2 Agents

At the agent level, we continue to use the traditional composition operators of concurrency theory, as described in Section 3.3.3. The primitives of Multi-Bach comprise the *tell, ask, nask* and *get* primitives. They can be composed to form more complex agents by: sequential composition noted ";", parallel composition noted "||" and non-deterministic choice noted "+". As in Anim-Bach, Multi-Bach introduces conditional statements. They are structured as $c \rightarrow a_1 \diamond a_2$, in which c represents a condition, and a_1 and a_2 are agents. Conditions are derived from basic ones using classical operators like (&), or (|), and negation (!). Elementary conditions arise by associating set elements or mappings with them through equalities (=) or inequalities (! =, <, <=, >, >=). Procedures also offer a simple way to group together sets of instructions. In Multi-Bach one creates them using the `proc` keyword to link an agent with a procedure name. This is similar to regular concurrency theory, where it's assumed that the agents being defined are guarded. This means that any procedure call comes after the execution of a basic action or can be transformed into that form.

The Multi-Bach model, [62] introduces an additional mechanism: generalized choices. They take the form $\Sigma_{e \in S} A_e$, where A_e is an agent that depends on a variable e taken from the set S. This is usually achieved by introducing a term involving e. The execution follows the general pattern of classical choices. All the variations of A_e obtained by allowing e to span all values in S are presented as potential options. The alternative capable of executing a computation step is chosen to carry out that step within the generalized choice.

The statements in the Multi-Bach language, referred to as agents informally, are composed of statements denoted as A. These statements are produced according to the grammar outlined below, where *Prim* signifies a primitive action, *Proc* represents a procedure call, C stands for a condition, e signifies a variable, and S denotes a set:

$$A \quad ::= \quad Prim \quad | \quad Proc \quad | \quad A\,;A \quad | \quad A \,||\, A \quad | \quad A + A \quad | \quad C \rightarrow A \diamond A \quad | \quad \Sigma_{e \in S} A_e$$

The operational semantics of the primitives remains the one given in Figure 4.1 in the previous chapter. The operational semantics of complex agents extends the one given in Figure 3.6. As before, configurations consist of agents representing the current state of the agent and a multiset of si-terms representing the current state of the store. As before also, to express the termination of an agent's computation uniformly, we consider E as a special terminating agent. For the sake of clarity, we rewrite agents in the form of $(E;A)$, $(E \,||\, A)$, and $(A \,||\, E)$ simply as A to maintain consistency with our intuitive understanding.

$$\textbf{(T)} \qquad \frac{t \rightsquigarrow u}{\langle\, tell(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$\textbf{(A)} \qquad \frac{t \rightsquigarrow u}{\langle\, ask(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \cup \{u\} \,\rangle}$$

$$\textbf{(G)} \qquad \frac{t \rightsquigarrow u}{\langle\, get(t) \mid \sigma \cup \{u\} \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

$$\textbf{(N)} \qquad \frac{t \rightsquigarrow u, u \not\in \sigma}{\langle\, nask(t) \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \sigma \,\rangle}$$

Figure 4.1: Transition rules for the primitives in Multi-Bach

Rules (S), (P) and (C) provide the usual semantics for sequential, parallel and choice compositions, already exposed in the previous chapter. Rule (Gc) generalizes rule (C) for multiple alternatives obtained by variable e ranging over set S. As expected, rule (Co) specifies that the conditional instruction $C \rightarrow A \diamond B$ behaves as A if condition C can be evaluated to true and as B otherwise. Note that the notation $\models C$ is used to denote the fact that C evaluates to true. Rule (Pc) makes procedure call $P(\overline{u})$ behave as the agent A defining procedure P with the formal arguments $\overline{x}$ replaced by the actual ones $\overline{u}$.

4.2.3 Processes as Active Data

Linda introduces an interesting concept where computation threads are treated as active data. This data is placed into a shared space and can be thought of as something to be executed.

In our system, defining procedures helps achieve this. When we make a procedure call, it's like starting a new task that runs alongside the current one. Because a procedure call looks like a piece of text and is similar to some other data we use, it becomes a new kind of data. As a consequence, repeatedly invoking the same procedure call results in the creation of multiple threads with identical names, much like repeatedly telling the same si-term leads to multiple occurrences in the store. Drawing an analogy, inquiring about or retrieving a procedure call involves checking for the presence of a corresponding thread, and in the case of retrieval, removing one of the threads associated with that call. Furthermore, applying a *"nask"* primitive to a procedure name entails verifying the absence of a thread corresponding to the procedure call. It's important to note that terminated processes are considered to be in a special stopped state but are not removed from the multiset of procedure calls.

As a result, the Multi-Bach process algebra introduces four new operations: *tellp(Proc)*, *askp(Proc)*, *naskp(Proc)* and *getp(Proc)*. These operations serve the purpose of creating a computational thread responsible for executing the specified procedure call *Proc*, checking for its presence or absence, and terminating a thread executing *Proc*. In the context of the Rush Hour problem, their computation follows the above explanation. For instance, if the red truck is defined by:

```
1    proc RedTruck = VerticalTruck(1,1).
```

then the execution of `tellp(RedTruck)` creates a new thread in charge of executing the agent `RedTruck`. Moreover, performing `getp(RedTruck)` amounts to killing this thread, whatever point of execution it has reached. Consequently, repeating a procedure call multiple times results in the creation of several threads with the same name, akin to multiple occurrences of the same si-term in the store. Following this analogy, inquiring or retrieving a procedure call involves checking for the presence of a corresponding thread, and in the case of retrieval, removing one of the threads associated with the call. Moreover, applying a `naskp` primitive to a procedure name involves verifying the absence of a thread corresponding to the procedure call. This approach allows treating windows created by the `New Autonomous Agent` and `new Interactive Agent` buttons as special cases of processes being told, with `Agent1`, `Agent2`, etc., as process call identifiers.

The formal description of the newly introduced `tellp(Proc)`, `askp(Proc)`, `naskp(Proc)`, and `getp(Proc)` primitives is obtained through a new transition relation denoted by the symbol $\hookrightarrow$. This relation operates on configurations of the form $\langle \mathscr{P} | \sigma \rangle$, where $\mathscr{P}$ is a multiset of pairs of the form $pc : A$, representing procedure call and agent pairs, and σ is a multiset of si-terms. The intuition is that $\mathscr{P}$ formalizes parallel-running threads, each identified by a

$$(\mathbf{O}) \quad \frac{\langle A \mid \sigma\rangle \longrightarrow \langle A' \mid \sigma'\rangle}{\prec \mathscr{P} \cup \{pc : A\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{pc : A'\} \mid \sigma' \succ}$$

$$(\mathbf{T_p}) \quad \prec \mathscr{P} \cup \{pc : tellp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{pc : E\} \cup \{P : P\} \mid \sigma \succ$$

$$(\mathbf{A^o_p}) \quad \prec \mathscr{P} \cup \{pc : askp(P)\} \cup \{P : A\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{pc : E\} \cup \{P : A\} \mid \sigma \succ$$

$$(\mathbf{A^s_p}) \quad \prec \mathscr{P} \cup \{P : askp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{P : askp(P)\} \mid \sigma \succ$$

$$(\mathbf{G^o_p}) \quad \prec \mathscr{P} \cup \{pc : getp(P)\} \cup \{P : A\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{pc : E\} \mid \sigma \succ$$

$$(\mathbf{G^s_p}) \quad \prec \mathscr{P} \cup \{Proc : getp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \mid \sigma \succ$$

$$(\mathbf{N_p}) \quad \frac{\mathscr{P} \cup \{pc : naskp(P)\} \neq \mathscr{P}' \cup \{P : A\} \text{ for any } \mathscr{P}' \text{ and } A}{\prec \mathscr{P} \cup \{pc : naskp(P)\} \mid \sigma \succ \hookrightarrow \prec \mathscr{P} \cup \{pc : E\} \mid \sigma \succ}$$

Figure 4.2: Transition rules for active data

reference pc and a current execution state A, while the store σ is shared among all threads for reading and writing.

The transition rules for the primitive agents are defined by rules (O) to (N_p) in Figure 4.2. Rule (O) relates the rules (T) to (N) of Figure 4.1, defining the transitions of the tell, ask, get and nask, with the new transition relation. It states that any computation step of $\langle A \mid \sigma\rangle$ can be lifted to the new configuration $\prec \mathscr{P} \cup \{pc : A\} \mid \sigma \succ$. Rule (T_p) states that $tellp(P)$ generates a new thread, referenced to by P with as current computation state, the agent P to be executed. Moreover, the process pc on which $tellp(P)$ is computed moves to the ending state E. It is here worth noting that union is to be understood in a multiset fashion. Rules (A^o_p) and (A^s_p) proceed in the same way but by checking that the asked process P belongs to the multiset of processes. Rule (A^O_p) treats the case of a process different from the process computing the $askp$ primitive while rule (A^s_p) allows the process to check itself. Rules (G^o_p) and (G^s_p) are similar with the process checked for presence being removed. Finally rule (N_p) specifies that $naskp(P)$ succeeds only when the multiset of procedure calls does not contain a process referenced to by P. It is worth noting that, for ease of reading, indices have been used in the rules of Figure 4.2 in a systematic manner as follows: the subscript p refers to processes (being told, asked, ...), the superscript s, for self, refers to transitions due to actions relating to the process under consideration, while the superscript o, for other, refers to transitions due to actions of other processes.

To completely describe the transitions, one should also lift the rules of Figure 3.6 for the operators to the new configurations. This can be done in a similar way as

rules of Figure 4.1 were lifted in rule (O). For instance, rule (S) vcan be rewritten as follows:

$$\frac{< \mathscr{P} \cup \{pc : A\} \mid \sigma > \hookrightarrow < \mathscr{P} \cup \{pc : A'\} \mid \sigma' >}{< \mathscr{P} \cup \{pc : A; B\} \mid \sigma > \hookrightarrow < \mathscr{P} \cup \{pc : A'; B\} \mid \sigma' >}$$

4.2.4 Rules

When we illustrate computations, it's important to capture not only the events directly produced by these computations, but also events indirectly triggered by them. For example, in the Rush Hour problem, if the red car leaves the grid and places the *'out'* si-term on the store, we may want to infer that the puzzle is solved which we could indicate by the *'position'* attribute of the red car widget changing from *'in_grid'* to *'out_grid'*. To accomplish this, as described in references [67] and [43], Multi-Bach uses rules of the form:

$$+\mathbf{t}, -\mathbf{u} \longrightarrow +\mathbf{v}, -\mathbf{w}$$

This rule asserts that the presence of object t and the absence of object u implies the presence of object v and the absence of object w. In a broader context, these rules can involve multiple positively and negatively marked objects on both sides. They indicate that any combination of objects from the left side results in adding positively marked objects on the right side and removing one occurrence of negatively marked objects on the right side. For example, in the rule mentioned above, if we add a new occurrence of t or remove u, it leads to adding v and removing one occurrence of w.

These rules are applied systematically whenever there is a change on the store. It's important to note that, following the analogy between passive si-terms and active processes, objects in these rules can be either si-terms or processes. To implement these rules in Multi-Bach, a two-step approach is followed:

- *Declaration:* Rules are declared by assigning a name to a rule in the specified format.
- *Activation:* Rules are activated by using the `tellr` command with the rule name as an argument. The presence or absence of rules can be checked using the `askr` and `naskr` primitives, respectively. To remove a rule, the `getr(rn)` command is used.

Example 14. *As an example, consider the following rule declaration:*

```
1 rule move_car =
2       for r in Rows, c in Cols, p in Colors:
3           +moveCar(p,r,c)  -->  +MoveCar(p,r,c), -moveCar(p,r,c).
```

In this rule, whenever a si-term like `moveCar(r,c,p)` is found in the store, where `p` represents a color, `r` represents a row number, and `c` represents a column number, it triggers the action `MoveCar(r,c,p)`. To prevent this rule from being applied multiple times, the si-term `moveCar(p,r,c)` is also removed. To activate the rule `move_car`, one executes `tellr(move_car)`, typically at the start of the computation. In our modelling of the Rush Hour problem, we use another rule, defined as follows:

Definition 8. *As an example, consider the following rule end of puzzle:*

```
1 rule end_puzzle = +out --> +puzzle_solved, +RedCarOut, -out.
```

with procedure `RedCarOut` *defined as:*

```
1 proc RedCarOut = att(position,red_car,rhScene,out_grid).
```

A rule is declared by using the keyword `rule` together with an equation of the form:

$$rule\_name = rule\_description.$$

with the following conventions:

- The *rule name* is a string of letters and digits, starting with a lower-case letter.
- The *rule description* is a construct of the form

$$quant : l\_m\_objects \longrightarrow l\_m\_objects$$

  where `quant` quantifies variables (represented as strings of letter and digits, starting with a lower case letter) over sets and where `l_m_objects` lists si-terms or procedure calls, positively or negatively marked.

To formally capture rules, we expand blackboards by including multisets of rule names. This leads us to new configurations of the form

$$\lhd \mathscr{P} \mid \sigma \mid R \rhd$$

where $\mathscr{P}$ is a multiset of procedure calls associated with agents, σ is a multiset of si-terms, and R is a multiset of rule names. Extending the rules from Figure 4.2 to accommodate these new configurations is straightforward. The execution of the primitives $tellr$, $askr$, $getr$, and $naskr$ can be described as expected by the corresponding updates of the R component. We denote the resulting transition relation as $\twoheadrightarrow$. However, the execution of rules requires attention and is denoted by an auxiliary relation $\mapsto$. Let's establish a few definitions first:

Definition 9. *Define the set SRP of rule primitives as the set of elements generated by the following grammar:*

$$RP \quad ::= \quad +s \quad | \quad -s \quad | \quad +P \quad | \quad -P$$

where s denotes a si-term, possibly containing variables, and P denotes a procedure call, possibly containing variables. Define the set SeqRP as the set of non-empty sequences of rule primitives. Given such a sequence Seq, we respectively denote Pterm(Seq) and Nterm(Seq) as the multisets of positively marked si-terms $+s$ and negatively marked si-terms $-s$ of Seq. Similarly, Pproc(Seq) and Nproc(Seq) are respectively the multisets of positively marked procedure calls $+P$ and negatively marked procedure calls $-P$ of Seq. Define the set Srule as the set of constructions of the form

$$for \quad v_1 \in S_1, ..., v_n \in S_n : A \rightarrow B$$

where $A, B \in SeqRP$, $S_1, \ldots, S_n$ are sets, and $v_1, \ldots, v_n$ are all the variables occurring in A and B. The sequence A is called the precondition of the rule, and the sequence B is called its postcondition.

Define the set Snrule of named rules as the set of constructs in the form $n = r$, where n is an identifier, composed of a sequence of lowercase letters, and r is a rule from Srule. It is assumed throughout that the identifier n for a named rule uniquely identifies the rule. Consequently, given an identifier n, denote Inst(n) as the set encompassing all instances $Ai \rightarrow Bi$. These instances are derived by replacing the variables $v_1, \ldots, v_n$ in the above rule with all values from $S_1, \ldots, S_n$.

Several auxiliary notations are needed to formulate the transition rules.

Definition 10. *For a multiset $\mathscr{P}$ consisting of pairs pc:A representing procedure calls and agents, we use $PC(\mathscr{P})$ to denote the multiset of procedure calls pc present in $\mathscr{P}$. Additionally, given a multiset MPC of procedure calls, we represent $\mathscr{P} \diagdown MPC$ as the multiset obtained by removing a pair pc:A for each pc found in MPC. Conversely, $\mathscr{P} \sqcup MPC$ denotes the multiset obtained by adding a pair of the form $pc : pc$ for every occurrence of pc in MPC.*

Definition 11. *Given two multisets σ and τ of closed si-terms, we denote $\sigma \ominus \tau$ as the multiset obtained from σ by removing an occurrence for each si-term in τ.*

Now, let's discuss the transition induced by the execution of a rule, formalized by rule (R) in Figure 4.3. Considering a rule from R identified by n, we examine an instance of that rule, denoted as $(A \rightarrow B)$, that can be applied. For this application, the positively marked si-terms and processes of the precondition A must be present in the current configuration, while the negatively marked si-terms and processes are absent. Formally, regarding si-terms, this involves checking that $Pterm(A) \subseteq \sigma$ for the positively marked si-terms and $Nterm(A) \cap \sigma = \emptyset$ for the negatively marked si-terms. The handling of procedure calls is similar, using $PC(\mathscr{P})$ instead of σ.

The application of such an instance $A \rightarrow B$ updates the current store σ by adding the positively marked si-terms of the postcondition B and removing an occurrence of any negatively marked si-terms. Similarly, the current multiset of procedure calls is updated by adding the positively marked procedure calls of B and removing an occurrence of any negatively marked procedure call of B. This is formalized in rule (R) by adding $Pterm(B)$ to σ and using the $\ominus$-difference with

$Nterm(B)$. Similarly, $\mathscr{P}$ is updated by adding new procedure calls for any element of $Pproc(B)$ and removing the procedure calls of $Nproc(B)$.

$$(\mathbf{R}) \quad \frac{\begin{array}{c} n \in R, (A \to B) \in Inst(n), Pterm(A) \subseteq \sigma, Pproc(A) \subseteq PC(\mathscr{P}), \\ Nterm(A) \cap \sigma = \emptyset, Nproc(A) \cap PC(\mathscr{P}) = \emptyset \end{array}}{\begin{array}{l} \lhd \mathscr{P} \mid \sigma \mid R \rhd \longmapsto \quad \lhd (\mathscr{P} \sqcup PC(Pproc(B))) \diagdown Nproc(B) \mid \\ \qquad\qquad\qquad\qquad (\sigma \cup Pterm(B)) \ominus Nterm(B) \mid R \rhd \end{array}}$$

Figure 4.3: Transition for the application of a rule

To conclude the formalization of rules, we establish a connection between the $\twoheadrightarrow$ and $\mapsto$ transition relations. This connection is defined through rules (W_1) and (W_2) in Figure 4.4. Additionally, we introduce a new transition relation $\rightarrowtail$, which serves as the final one. Rule (W_1) asserts that a $\rightarrowtail$ transition step is permissible whenever a $\mapsto$ transition step can take place. Conversely, rule (W_2) specifies that a $\twoheadrightarrow$ transition step is only possible when a $\mapsto$ transition step is not feasible. This signifies that rules are applied to their fullest extent before any other transition can occur.

$$(\mathbf{W_1}) \quad \frac{\lhd \mathscr{P} \mid \sigma \mid R \rhd \mapsto \lhd \mathscr{P}' \mid \sigma' \mid R' \rhd}{\lhd \mathscr{P} \mid \sigma \mid R \rhd \rightarrowtail \lhd \mathscr{P}' \mid \sigma' \mid R' \rhd}$$

$$(\mathbf{W_2}) \quad \frac{\begin{array}{c} \lhd \mathscr{P} \mid \sigma \mid R \rhd \twoheadrightarrow \lhd \mathscr{P}' \mid \sigma' \mid R' \rhd, \\ \lhd \mathscr{P} \mid \sigma \mid R \rhd \not\mapsto \lhd \mathscr{P}'' \mid \sigma'' \mid R'' \rhd \end{array}}{\lhd \mathscr{P} \mid \sigma \mid R \rhd \rightarrowtail \lhd \mathscr{P}' \mid \sigma' \mid R' \rhd}$$

Figure 4.4: Rules for the final transition relation

4.2.5 Animations

Animations are created through a two-part process in Anemone. Firstly, by describing the scene that needs to be portrayed, and secondly, by utilizing specific commands to position, move, make widgets appear or disappear, and modify their characteristics. Within the framework of Anemone, it's possible to define multiple scenes.

Describing a scene involves specifying various elements such as the canvas's size, the background image to be used, a sequence of images, and an array of widgets. A widget, in turn, encompasses attributes' definitions, their visual representation, and the initial values for these attributes. In addition to the user-defined attributes, five predefined attributes are available:

- The current coordinates (`wdX` and `wdY`) of the widget.

- The rotation angle (`wdA`) of the widget.
- The scale factor (`wdF`) of the widget.
- The visibility (`wdV`) of the widget.
- The layer (`wdL`) of the widget.

This allows for the flexible and detailed creation of animations within the Anemone framework.

Example 15. *The provided example defines a scene named `rhScene`, with dimensions 640 by 640 pixels and three layers: `top`, `middle`, and `bottom`. It utilizes images for the background, a green car, and a red car. A widget `car` is defined with a user-defined attribute `color`, displayed conditionally as a green or red car. Initial values for predefined attributes are specified: initial positions at (40,60), layer as `top`, and color as `red`. Predefined attributes default to `0` or `1` if not explicitly defined.*

Note that file names are relative to the Anemone *execution path, and canvas size and coordinates are in pixels, with (0,0) as the upper-left corner.*

```
1  scene rhScene =
2  {  size    = (640,640).
3     layers = { top, middle, bottom }.
4
5     background = loadImage(Images/the_background_img.png).
6     green_car  = loadImage(Images/green_car.jpg).
7     red_car    = loadImage(Images/red_car.jpg).
8     widget car = { attributes = { color in idColors.  }
9                    display    = { color = green -> green_car.
10                                  color = red -> red_car.    }
11                   init       = { wdX = 40.   wdY = 60.
12                                  wdL = top.   color = red.  }}
13 }
```

Scene and widget manipulation involves two primitives:

- `draw_scene(s)`: Draw the scene identified by *s*.
- `att(x,w,s,v)`: Set the value *v* to the attribute *x* of the widget *w* on scene *s*.

Additionally, there are primitives that provide a more declarative approach for common operations on predefined attributes:

- `place_at(w,s,x,y)`: Place the widget *w* on scene *s* at coordinates (x, y).
- `move_to(w,s,x,y)`: Move the widget *w* on scene *s* to new coordinates (x, y).
- `hide(w,s)`: Hide the widget *w* on scene *s*.
- `show(w,s)`: Make the widget *w* appear on scene *s*.
- `layer(w,s,l)`: Place the widget *w* on scene *s* on layer *l*.

These primitives seamlessly integrate with the existing `tell`, `ask`, `get`, and `nask` primitives without necessitating modifications to the existing transition relations. A formal treatment of these new primitives can be achieved through a sequence of widget updates, but for simplicity such details are omitted here. A notable feature is the symbolic representation of coordinates using map constructs, simplifying the specification of image positions.

4.3 Anemone, Overview of the Coordination Workbench

In line with the principles of Linda, the `MultiBach`, as discussed in Section 4.2, utilizes a shared space for coordinating processes. This shared space plays a crucial role in achieving the separation of time and space within processes, a fundamental concept in data-based coordination languages [95].

Consequently, following the `Scan` workbench, `Anemone` [62] is structured around an interactive blackboard, as illustrated in Figure 4.5. This blackboard initially displays the current contents of the shared space and allows for direct interaction through the *tell, get,* and *clear* buttons. Two new buttons, *filter* and *unfilter*, have been introduced to respectively filter and remove filters from the content in the shared space. Moreover, it offers the ability to create five different types of processes.

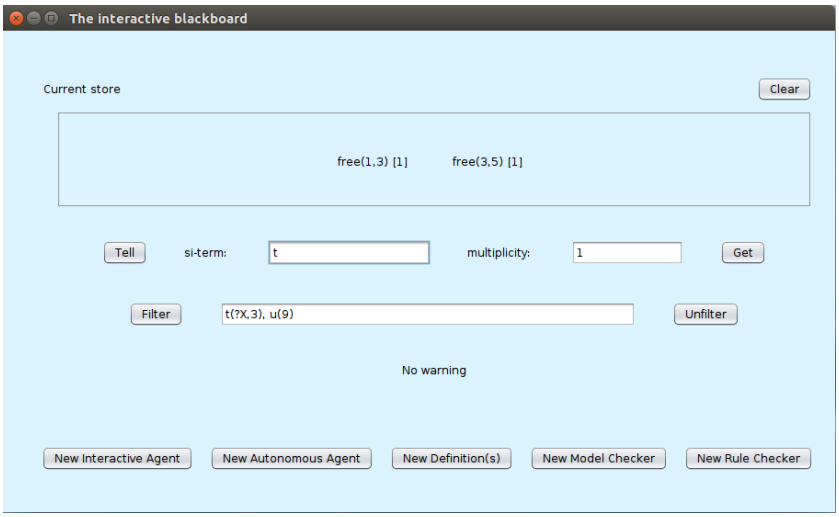


Figure 4.5: Interactive blackboard window

In this new version, we have kept a feature called the `Interactive Agent`, along with the `Autonomous Agent`. These processes allow users to give instructions to Multi-Bach and execute them. As shown in Figure 4.6, the first type of windows allows step-by-step computations, giving users the choice of which primitives to execute. Similarly to the `Scan` workbench, windows of the second type conduct

computations either in a single run or in a step-by-step fashion, with the `Anemone` workbench determining the primitives to execute. It's important to highlight that these window executions occur in parallel, hence the term *agent* used to signify entities capable of concurrent activities. In both scenarios, a new option, with the `abbrev` button, offers a convenient way to condense complex instructions by concealing the non-executable components. As mentioned earlier in Chapter 3, the capabilities provided by the interactive and autonomous agents are particularly useful for debugging concurrent executions that operate directly on the shared space, including scenarios where processes may deadlock while awaiting unavailable data. However, these agents lack the level of abstraction required for describing and reasoning about models effectively. This leads to three types of windows:

(i) a window to declare sets, procedures, scenes, and rules. It is launched by means of the `New definition(s)` button,
(ii) a window to reason on the execution of rules. It is launched by the `New rule checker` button,
(iii) and a window to model check properties. It is launched by the `New model checker` button.

As regards user interactions, many of Scan's features are covered by the Anemone workbench. Two features require additional treatments: processes perceived as active data and rules.

The interactive and autonomous agent windows can be seen by users as representing processes, as their function involves computing agents, including procedure calls. Therefore, in the workbench, executing the `tellp(P)` primitive means creating a window similar to the interactive or autonomous agent windows. Consequently, executing the `getp(P)` primitive involves not only removing an occurrence of the process P from the multiset of procedure calls $\mathscr{P}$ but also removing the window associated with P. We assume that the interactivity or autonomy property is inherited from the agent window on which the `tellp` primitive is executed.

Moving on to rules, they necessitate a new type of window. To provide users with a clear view of the current calculation state, the Anemone workbench displays the current instances of rules applicable in the current application context. Users are then responsible for selecting the rule to apply.

4.4 Implementation of Anemone

The implementation of Anemone follows the same implementation lines as Scan, reusing its core principles while expanding functionality. Like Scan, it is developed in Scala using the Processing library, which enables a robust integration of object-oriented and functional paradigms. Scala's powerful parsing capabilities facilitate interpreting Multi-Bach language, while Processing is used for graphical output and animation through its `draw` method.

(a) Interactive agent window

(b) Autonomous agent window

Figure 4.6: Interacting with the blackboard

The internal representation relies on Scala maps and lists for storing data such as processes and definitions, maintaining a one-to-one mapping between Multi-Bach instructions and their abstract representations. This ensures user interfaces and feedback remain closely tied to user input, a feature less intuitive in other tools like mCRL2.

Anemone computations simulate Multi-Bach execution through repeated transition steps and randomization for non-deterministic behaviours, such as choice and parallel compositions. Temporal logic verification is achieved with a custom model-checker that employs depth-first search and recursive reasoning but faces challenges with state-space explosion.

The Anemone workbench is accessible as a downloadable `jar` file through the project's GitHub repository at [13]. The complete source code is also available in

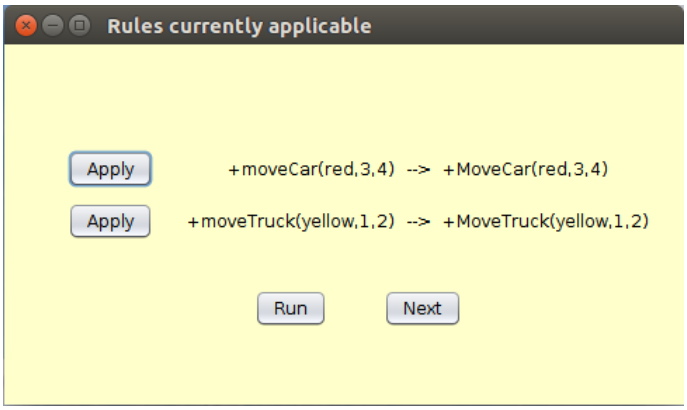


Figure 4.7: Window for rules

the same repository. The implementation primarily uses Scala and the Processing library. To launch the workbench, save the Anemone jar file in a directory (e.g., `myDir`). Then, open a terminal and execute the following command:

```
1        java -jar myDir/anemone.jar
```

4.5 Related work

In the coordination community, much effort has gone into proposing new languages, semantics, and implementations. However, few articles address the practical construction of programs in coordination languages and the verification that program descriptions align with intended models. Notable exceptions include the Extensible Coordination Tools [69], ReoLive [39], and TAPAs [32].

Extensible Coordination Tools (ECT) is designed for the control-based coordination language Reo, distinct from Multi-Bach. ECT tools, as Eclipse plug-ins, offer graphical editing of Reo connectors, animation, and model checking using constraint automata or translation to the mCRL2 process algebra [38]. In contrast, our work focuses on tuple spaces instead of connectors and allows connecting Multi-Bach agents to animations at the application level, providing a unique capability not found in ECT. ReoLive dedicated to Reo like ECT, provides similar tools through web-based tools using ScalaJS. The comparison with ECT is applicable to ReoLive as well. TAPAs primarily targets CCSP with a plug-in for an extended Klaim coordination language. It allows graphical system specification and verification through bisimulations and decorated trace equivalences, along with model checking using μ-calculus formulae. Our work differs by focusing on tuple-based coordination languages and offering animation facilities. However, our model-checking capabilities are less advanced than those of TAPAs.

Klaim a Linda-like coordination language for distributed systems, has its X-Klaim variant utilizing modern IDE facilities [18]. While Anemone lacks similar semantic analysis facilities, it compensates with animations, not provided by X-Klaim. Declarative invariant assertions in [70] aims to detect inconsistencies in models expressed in the Peer model, a coordination model. Unlike Multi-Bach, assertions are runtime-verified, and the work lacks animation facilities. TUCSON [93], despite featuring shared space evolution viewing, lacks animation and model-checking facilities. JavaSpaces Netbeans [48] integrates a Linda-inspired language with Netbeans, offering tuple browsing and distributed debugging. Unlike our work, it lacks debugging facilities but provides record facilities for tuple space operations replay.

The work in [45] uses a process algebra for modelling systems, introducing a language based on virtual stigmergy. Unlike Multi-Bach, it lacks a programming environment and animation facilities. NetLogo, Jason, JaCaMo, and Repast focus on modelling and simulating multi-agent systems but lack a real concern for coordination. They emphasize system modelling and simulation, whereas Multi-Bach aims to build confidence in models through animations. Several works, like [71, 72, 86, 112, 121, 125, 60], provide animations for understanding real-life models in various domains, such as formal specifications, goal-oriented requirements, and reactive systems. Our work, while focusing on a coordination language, shares the fundamental idea of illustrating models.

This effort expands on the Scan workbench [63] by improving features and incorporating elements such as generalized choices, primitives for managing processes as active data, and refined scene descriptions. Notable additions include an interactive blackboard with a filter button, windows for generating new processes as passive data, and a novel window type for managing rules.

4.6 Closing Remarks

In summary, this chapter introduces a testing environment for analysing a Linda-like coordination language on three levels: (i) executing instructions step-by-step or automatically, showcasing their impact on the shared space, (ii) visually representing computations through animations, and (iii) validating property models using temporal formulas.

As written above, Multi-Bach and Anemone extends the work reported in Chapter 3 on Anim-Bach and Scan. In the next chapter, we will introduce a guarded list construct within our model Multi-Bach and demonstrate how it enhances performance when dealing with state-space explosion.

Part III

Optimizing Programs

CHAPTER 5

EXPLORING THE GUARDED LIST CONCEPT: PERFORMANCE AND CORRECTNESS

As well-known in concurrency theory, verifying logic formula is recognized to raise performance issues due to the state space explosion problem. The temporal logic formulae of Anemone are no exception to this general rule. In this chapter, we propose a guarded list construct as a solution to address this problem. We demonstrate that the guarded list construct increases performance of data-based coordination languages. Furthermore, we introduce a notion of refinement to introduce the guarded list construct in a correctness-preserving manner.

5.1 Opening

In Chapter 3, we have introduced a workbench Scan to reason on programs written in Bach, a Linda-like dialect developed at the University of Namur. It has been refined in Chapter 4 to cope with relations, processes and multiple scenes. The resulting workbench has been named Anemone. In both cases, one of our goals was to allow the user to check properties by temporal logic formulae analysis, and by producing traces that can be replayed as evidence of the establishment of the formulae. However, as well-known in formal verification, this goal raises performance issues related to the state space explosion. In particular, letting animation-related primitives interleave in many ways duplicates research paths during temporal logic verification, with considerable performance problems in checking that formulae are established. To address this problem, we introduce in this chapter a guarded list construct, and establish that it yields an increase in performance in verifying logic formula of Multi-Bach.

To that end, the rest of this chapter is organized as follows. Section 5.2 explains the creation of the guarded list and introduces the refinement relation. The efficiency improvements in temporal logic verification are established in Section 5.3. Finally, Section 5.4 compares our work with related studies, and Section 5.5 concludes the chapter, outlining potential future work.

5.2 A Guarded List Construct

5.2.1 General Context

In this section, we will describe a new extension to our coordination model MultiBach, the "**Guarded list**". We define a guarded list of primitives to be a structure of the form $[p \rightarrow p_1, \cdots, p_n]$ where p, p_1, ..., p_n represent primitives, and the list p_1, ..., p_n may be empty. In cases where the list is empty, we will simply write $[p]$ for simplicity.

A guarded list of primitives, basically, is a collection of primitives that must include at least one. The term "*guarded*" is derived from the practice of representing these lists with an arrow, and it signifies that if the first primitive can be executed successfully, all subsequent primitives are immediately executed without the possibility of roll-back in case of a failure. It is the responsibility of programmers to ensure that if the first primitive can be successfully evaluated, the remaining primitives can also be executed without issues. It is important to note that this condition holds for tell primitives and graphical primitives, as they consistently succeed regardless of the current store content. Additionally, we will later outline criteria for introducing guarded lists while maintaining correctness. It is important to note that guarded lists are atomic constructs, which sets them apart from conditional statements.

To explain the execution of $[p \rightarrow p_1, \ldots, p_n]$, the process unfolds as follows:

(i) Initially, the store is locked, and the execution of p is tested.
(ii) If it fails, no changes are made to the store, and it is then released.
(iii) However, if the test on p succeeds, not only is p executed, but also subsequently, $p_1, \ldots, p_n$ are executed in sequence. In case one of the p_i's primitives does not succeed then the whole computation fails on that point.

After this sequence, in all the cases, the store is released. In contrast, the execution of a conditional statement $c \rightarrow s_1 \diamond s_2$ involves checking the condition c, which does not necessitate locking the store because conditions are based on comparing siterms rather than their presence or absence in the store. If the condition c evaluates to true, then s_1 is executed, meaning that one step of s_1 is performed if possible. If c is determined to be false, then one step of s_2 is attempted.

The operational semantics of guarded lists is determined by the rules (GL1) to (GL3) presented in Figure 5.1. They necessitate the following definition.

$$\textbf{(GL1)} \qquad \frac{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle}{\langle\, [p] \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle}$$

$$\textbf{(GL2)} \qquad \frac{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle,\ \langle\, seq(L) \mid \tau \,\rangle \longrightarrow^{*} \langle\, E \mid \phi \,\rangle}{\langle\, [p \rightarrow L] \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \phi \,\rangle}$$

$$\textbf{(GL3)} \qquad \frac{\langle\, p \mid \sigma \,\rangle \longrightarrow \langle\, E \mid \tau \,\rangle,\ \langle\, seq(L) \mid \tau \,\rangle \longrightarrow^{*} \langle\, A \mid \phi \,\rangle,\ A \neq E}{\langle\, [p \rightarrow L] \mid \sigma \,\rangle \longrightarrow \langle\, F \mid \phi \,\rangle}$$

Figure 5.1: Transition rules for guarded lists [16, 15]

Definition 12. *Let $L = [p_1, \cdots, p_n]$ be a list of primitives. We denote by $seq(L)$ the sequential composition $p_1; \cdots; p_n$.*

In essence, with respect to the guarded list $[p \rightarrow L]$, the $seq(L)$ construct captures the sequential nature of the execution of the primitives guarded by the primitive p.

This notation being given, rules of Figure 5.1 are explained as follows. Rule (GL1) states that a guarded list $[p]$ composed of one primitive succeeds whenever the primitive p succeeds, with an equal update of the store. Rule (GL2) establishes that a guarded list in the form of $[p \rightarrow L]$ can perform a computation step, transitioning from the store σ to ϕ, if the primitive p can take a step, changing the store from σ to τ, and if the list of primitives L can all be further executed, modifying τ to ϕ. Finally, rule (GL3) treats the case of failure in computing the list L. This happens when primitive p can be successfully executed from store σ yielding τ and when the list L can be partly executed but not to the end. In that case a failure is produced and is denoted as F. It is worth noting here that to force the computation to fail as a whole, we assume subsequently that any composition of the form $F; A$, $F||A$, $A||F$ is rewritten as F, for any agent F.

5.2.2 A Theory of Refinement

Incorporating guarded lists as atomic constructs reduces the potential for interleaving between parallel processes. This reduction in interleaving aligns with our objective of enhancing the efficiency of the formal verification phase. However, when viewed from a programming perspective, it is imperative to ensure that computations are maintained in some manner. This is precisely why we introduce histories and their contractions.

General Considerations

First, let's introduce the concepts of histories and their contractions:

Definition 13. *We define the set of computational histories, referred to as "histories" and denoted as Shist, as the set of two components: $Sstore^{\omega} \cup Sstore^{*}.\{\delta^{+}, \delta^{-}\}$, where:*

(i) Sstore represents the set of stores, which are finite multisets of final si-terms.
(ii) the symbols $*$ and ω are used to indicate finite and infinite repetitions, respectively.
(iii) the symbols δ^{+} and δ^{-} are introduced as ending marks. With δ^{+} denoting successful computations, while δ^{-} denotes failing computations.

In summary, the set of histories Shist encompasses sequences of stores that can be either finite or infinite, with the addition of ending markers δ^{+} and δ^{-} to signify the outcomes of these computations.

Example 16. *To illustrate computational histories, let us consider the following procedures whose code is inspired from the* `VerticalTruck` *procedure given in Chapter 3:*

```
1 proc P = tell(a); Q.
2       Q = get(a); move_to(...); att(...); tell(b); tell(a).
```

Note that the parameters of `move_to` *and* `att` *being of no interest for our illustrative purposes, we have replaced them by dots.*

A computational history for P is obtained by executing the different primitives from the empty store: we thus obtain successively

$\{\}$	*as a starting store*
$\{a\}$	*after executing* `tell(a)`
$\{\}$	*after executing* `get(a)`
$\{\}$	*after executing* `move_to(...)`
$\{\}$	*after executing* `att(...)`
$\{b\}$	*after executing* `tell(b)`
$\{a,b\}$	*after executing* `tell(a)`

As the computation ends successfully, a computational history for P is

$$\{\}.\{a\}.\{\}.\{\}.\{\}.\{b\}.\{a,b\}.\delta^{+}$$

Consider now `R` *defined by*

```
1 proc R = tell(a); Q; nask(b).
```

Basically `R` *is* `P` *followed by* `nask(b)`*. As* `b` *is present on the store resulting from the computation of* `P`*, the execution of* `nask(b)` *ends in a failure state, which leads us to the following computational history:*

$$\{\}.\{a\}.\{\}.\{\}.\{\}.\{b\}.\{a,b\}.\delta^{-}$$

To get an infinite computational history, consider the following variant `Qinf` *of* `Q`*:*

```
1 proc Pinf = tell(a); Qinf.
2      Qinf = get(a); move_to(...); att(...); tell(b); tell(a); Qinf.
```

Obviously a computation of `Pinf` *starts by telling* `a` *and then repeats steps 2 to 7 of the computational history of* `P` *by adding an occurrence of* `b` *after each repetition. This leads to the following infinite computational history:*

$\{\}.\{a\}.$

$\{\}.\{\}.\{\}.\{b\}.\{a,b\}.$

$\{b\}.\{b\}.\{b\}.\{b,b\}.\{a,b,b\}.$

$\{b,b\}.\{b,b\}.\{b,b\}.\{b,b,b\}.\{a,b,b,b\}.$

$\ldots$

Definition 14. *A history denoted as h_c is considered a contraction of another history denoted as h if it can be derived from the latter by eliminating a finite number of its elements (which could be none at all), while preserving the terminating marks δ^+ and δ^-. This relationship is denoted as $h_c \preceq h$.*

Example 17. *Coming back to the previous example,*

$$h_c = \{\}.\{a\}.\{a,b\}.\delta^-$$

is a contraction of

$$h = \{\}.\{a\}.\{\}.\{\}.\{\}.\{b\}.\{a,b\}.\delta^-$$

since h_c results from h by eliminating the sequence

$$\{\}.\{\}.\{\}.\{b\}$$

Note that, if we take $\overline{\sigma_0}$ and $\overline{\sigma_1}$ as the empty sequence and if we write

$$\overline{\sigma_2} = \{\}.\{\}.\{\}.\{b\}$$

h can be expressed from h_c as

$$h = \overline{\sigma_0}.\{\}.\overline{\sigma_1}.\{a\}.\overline{\sigma_2}.\{a,b\}.\delta^+$$

Definition 15. *Given a contraction $h_c = \sigma_0.\cdots.\sigma_n.\delta$ (resp. $h_c = \sigma_0.\cdots.\sigma_n.\cdots$) of an history h, there are thus sequences of stores, $\overline{\sigma_0}$, ..., $\overline{\sigma_n}$ such that $h = \overline{\sigma_0}.\sigma_0.\cdots.\overline{\sigma_n}.\sigma_n.\delta$ (resp. $h = \overline{\sigma_0}.\sigma_0.\cdots.\overline{\sigma_n}.\sigma_n.\cdots$). For any logic formula F, the history h_c is said to be F-preserving iff, for any i and for any store τ of $\overline{\sigma_i}$, one has $\tau \models F$ iff $\sigma_i \models F$. This is subsequently denoted as $h_c \ll_F h$.*

Example 18. *The previous example has already illustrated how an history can be obtained from a contraction. Going back to it it is easy to see that for $F = (\#b \geq 1)$, one has $\sigma_i \models F$ iff for any store τ of $\overline{\sigma_i}$, one has $\tau \models F$. In fact, this amounts to say that executing*

```
1 get(a); move_to(...); att(...); tell(b)
```

in one big step or step by step does not change the fact that the number of b *is greater than one or not.*

Contractions and F-preserving contractions can be lifted in an obvious way to sets of histories.

Definition 16. *A set S_c of histories is a contraction (resp. a F-preserving contraction) of a set S of histories if any history of S_c is the contraction (resp. a F-preserving contraction) of a history of S. By lifting notations on histories, this is subsequently denoted by $S_c \preceq S$ (resp. $S_c \ll_F S$).*

Then, let's define the history-based operational semantics as the one delivering all the computational histories. To make it general, we shall define it on any contents of the initial store.

Definition 17. *Define the language $\mathscr{L}_g$ as the* Multi-Bach *language with the guard list construct.*

Definition 18. *Define the* ***operational semantics*** $\mathscr{O}_h : \mathscr{L}_g \to \mathscr{P}(Shist)$ *as the following function. For any agent A and any store τ*

$$
\begin{aligned}
&\mathscr{O}_h(A)(\tau) = \\
&\quad \{\sigma_0.\cdots.\sigma_n.\delta^+ : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \sigma_n\rangle, \sigma_0 = \tau, n \geq 0\} \\
&\quad \cup \{\sigma_0.\cdots.\sigma_n.\delta^- : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle A_n \mid \sigma_n\rangle \not\longrightarrow, \sigma_0 = \tau, A_n \neq E, n \geq 0\} \\
&\quad \cup \{\sigma_0.\cdots.\sigma_n.\cdots : \langle A \mid \sigma_0\rangle \longrightarrow \cdots \longrightarrow \langle A_n \mid \sigma_n\rangle \longrightarrow \cdots, \sigma_0 = \tau, \forall n \geq 0 : A_n \neq E\}
\end{aligned}
$$

Now, we are in a position to define the refinement of agents.

Definition 19. *Agent A is said to refine agent B iff $\mathscr{O}_h(A)(\tau) \preceq \mathscr{O}_h(B)(\tau)$, for any store τ.*

The following proposition directly follows from the definitions provided above. Its purpose is to establish contractions and F-preserving properties based on a syntactic characterization. Part (i) of the proposition expresses that when p_1, ..., p_n are tell primitives of graphical primitives, namely primitives that always succeed, then computing p, p_1, ..., p_n atomically or step by step does not matter since p_1, ..., p_n always succeed. As a result, the same properties are reached after both executions. However properties met at the end of the whole computation may not be met in middle points, after the execution of some p_i's. Part (ii) of the proposition addresses that point and states that properties do hold in middle points in case the tell primitives in the p_i's involve si-terms that are of no concern for the property under consideration.

Proposition 1.

(i) If $p_1, \cdots, p_n$ are tell primitives or graphical primitives then for any primitive p, the guarded list $GL = [p \rightarrow p_1, \cdots, p_n]$ refines the sequential composition $SC = p; p_1; \cdots; p_n$. As a result, any reachable property proved on the stores generated by the execution of GL from a given store τ is also established on the stores generated by the execution of SC from τ.

(ii) Assuming additionally that the arguments of the tell primitives of $p_1, \cdots, p_n$ are distinct from the si-terms appearing in the reachable formulae F, then GL is also a F-preserving contraction of SC. It results that F is established on the stores resulting from the execution of SC from any store τ iff it is established on the stores resulting from the execution of GL from τ.

Introducing Guarded Lists

As previously explained, the rush hour puzzle can be framed as a coordination problem by treating cars and trucks as autonomous agents that must coordinate based on available spaces. This leads us to the generic procedure `VerticalTruck` from Section 3.3.3. The problem is considered solved when the out si-term is placed on the store, which ensures that the property $\#out = 1$ can be achieved. Now, let's focus on the following snippet from the first alternative provided by `VerticalTruck`:

```
1 get(free(pred(r),c));
2 move_to(truck_img(c),pred(r),c);
3 tell(free(succ(succ(r)),c))
```

As easily checked, the hypotheses of Proposition 1 are verified so that we can replace the above code snippet by the following one:

```
1 [ get(free(pred(r),c)) ->  move_to(truck_img(c),pred(r),c),
2                            tell(free(succ(succ(r)),c)) ]
```

Indeed, this code is an F-preserving contraction for the formulae $F = (\#out = 1)$.

The same applies for the second alternative, which allows us to refine the procedure `VerticalTruck` as follows:

```
1  proc VerticalTruck(r: RCInt, c: RCInt, p: Colors) =
2      ( (r>1 & r<5) ->  ( [ get(free(pred(r),c)) ->
3                            move_to(truck_img(p),pred(r),c),
4                            tell(free(succ(succ(r)),c)) ];
5                            VerticalTruck(pred(r),c,p) ))
6      +
7      ( (r<4)       -> ( [ get(free(cbelow_truck(r),c)) ->
8                           move_to(truck_img(p),succ(r),c),
9                           tell(free(r,c)) ];
10                          VerticalTruck(succ(r),c,p) )).
```

By applying this transformation, one gains the computation of two stores out of four per vehicle, which leads to the expectation of a performance gain of 2^n when n represents the number of vehicles in parallel. This will be verified in Section 5.3.

5.2.3 Transforming Programs

Although simple, the refinement from the previous subsection enables further transformations of the program, resulting in an additional improvement during the formal verification phase. The key point is that after refinement, the `VerticalTruck` procedure takes the following form:

```
P = (c1 -> a1; P) + (c2 -> a2; P)
```

where with `c1` and `c2` are conditions and where `a1` and `a2` are atomic actions. Consider another one `Q` defined similarly as

```
Q = (d1 -> b1; Q) + (d2 -> b2; Q)
```

and consider the parallel composition

```
Syst = P || Q
```

One has

```
Syst =  P || Q
     =  (c1 -> a1; (P||Q)) + (c2 -> a2; (P||Q))
       + (d1 -> b1; (P||Q)) + (d2 -> b2; (P||Q))
     =  (c1 -> a1; Syst) + (c2 -> a2; Syst)
       + (d1 -> b1; Syst) + (d2 -> b2; Syst)
```

To be more specific, consider an instance of rush-hour composed of one vertical purple truck, located in coordinates (2,1), and the red car, located at coordinates (3,2). The solution to this problem may be given by attempting to reach formulae ($\#out = 1$) for the following agent:

```
VPurpleTruck(2,1) || RedCar(3,2)
```

or, rephrased in more generic terms[1]

```
VerticalTruck(r,c,purple) || HorizontalCar(r,c,red)
```

To be even more general consider

```
Syst(tr,tc,tco,rr,rc,rco) =
    VerticalTruck(tr,tc,tco) || HorizontalCar(rr,rc,rco)
```

[1] The red car is a horizontal car.

For the ease of writing, let us use the following abbreviations:

```
A1 = [ get(free(pred(tr),tc)) -> move_to(truck_img(tco),pred(tr),tc),
                                 tell( free(succ(succ(tr)),tc )) ]
A2 = [ get(free(cbelow_truck(tr),tc)) ->
                                 move_to(truck_img(tco),succ(tr),tc),
                                 tell(free(tr,tc)) ]
```

The (refined) `VerticalTruck` can then be coded as

```
VerticalTruck(tr,tc,tco) =
      ( (tr>1 & tr<5) -> ( A1; VerticalTruck(pred(tr),tc,tco) ) )
    + ( (tr<4)        -> ( A2; VerticalTruck(succ(tr),tc,tco) ) )
```

Similarly, by defining

```
B1 = [ get(free(rr,pred(rc))) ->  move_to(car_img(rco),rr,pred(rc)),
                                  tell(free(rr,succ(rc))) ]
B2 = [ get(free(rr,succ(rc))) ->  move_to(car_img(rco),rr,succ(rc)),
                                  tell(free(rr,rc)) ]
B3 = [ tell(out)              ->  move_to(car_img(rco),rr,succ(rc)) ]
```

the refined `HorizontalCar` procedure can be coded as

```
HorizontalCar(rr,rc,rco) =
      ( (rc>1 & rc<6)            -> ( B1; HorizontalCar(rr,pred(rc),rco)))
    + ( (rc<5)                   -> ( B2; HorizontalCar(rr,succ(rc),rco)))
    + ( (rr=3 & rc=5 & rco=red) -> ( B3; HorizontalCar(rr,succ(rc),rco)))
```

By slightly generalizing the transformation in the beginning of this section, one gets

```
Syst(tr,tc,tco,rr,rc,rco) =

  VerticalTruck(tr,tc,tco) || HorizontalCar(rr,rc,rco) =

  ( (tr>1 & tr<5) ->
    (A1; (VerticalTruck(pred(tr),tc,tco) || HorizontalCar(rr,rc,rco)))

+ ( (tr<4) ->
    (A2; (VerticalTruck(succ(tr),tc,tco) || HorizontalCar(rr,rc,rco)))

+ ( (rc>1 & rc<6) ->
    (B1; (VerticalTruck(tr,tc,tco) || HorizontalCar(rr,pred(rc),rco)))

+ ( (rc<5) ->
    (B2; (VerticalTruck(tr,tc,tco) || HorizontalCar(rr,succ(rc),rco)))

+ ( (rr=3 & rc=5 & rco=red) ->
    (B3; (VerticalTruck(tr,tc,tco) || HorizontalCar(rr,succ(rc),rco))))
```

As we are only interested by the red car going out, we can even skip the last parallel composition and rewrite:

```
Syst(tr,tc,tco,rr,rc,rco) =

  VerticalTruck(tr,tc,tco) || HorizontalCar(rr,rc,rco) =

  ( (tr>1 & tr<5) ->
    ( A1; ( VerticalTruck(pred(tr),tc,tco) || HorizontalCar(rr,rc,rco)))

+ ( (tr<4) ->
    ( A2; ( VerticalTruck(succ(tr),tc,tco) || HorizontalCar(rr,rc,rco)))

+ ( (rc>1 & rc<6) ->
    ( B1; ( VerticalTruck(tr,tc,tco) || HorizontalCar(rr,pred(rc),rco)))

+ ( (rc<5) ->
    ( B2; ( VerticalTruck(tr,tc,tco) || HorizontalCar(rr,succ(rc),rco)))

+ ( (rr=3 & rc=5 & rco=red) -> ( B3 ) )
```

and, by the definition of `Syst`,

```
Syst(tr,tc,tco,rr,rc,rco) =

  ( (tr>1 & tr<5) -> ( A1; Syst(pred(tr),tc,tco,rr,rc,rco) ) )

+ ( (tr<4)        -> ( A2; Syst(succ(tr),tc,tco,rr,rc,rco) ) )

+ ( (rc>1 & rc<6) -> ( B1; Syst(tr,tc,tco,rr,pred(rc),rco) ) )

+ ( (rc<5)        -> ( B2; Syst(tr,tc,tco,rr,succ(rc),rco) ) )

+ ( (rr=3 & rc=5 & rco=red) -> ( B3 ) )
```

In terms of the parallel composition of the initial agent, the performance gain arises from the fact that the search is directly represented by a recursive procedure, where the arguments correspond to a state in the search space to be explored.

Note that the above code is equivalent to what would be obtained by applying the state-space approach commonly found in classical AI textbooks (e.g., chapter 3 of [108]). In this approach, the rush hour problem would be defined in terms of states representing the various positions of the vehicles, with transitions between them based on vehicle movements. Interestingly, we have arrived at the same code by applying program transformations to a version where vehicles compete for free cells.

5.3 Performance

Now, let's illustrate the efficiency gains achieved during temporal logic verification with the utilization of the guarded list construct. For this purpose, we will compare the performance of the *Scan* and *Anemone* breadth-first search model checker on various instances of the rush hour puzzle. These instances will be coded both without the guarded list construct and with the guarded list construct.

As explained in earlier sections, the rush hour puzzle can be represented as a coordination problem, where cars and trucks are treated as autonomous agents that must coordinate based on available spaces. The complete code for this is accessible in Appendix C.The code for cars and trucks adheres to a pattern similar to the code presented in Section 5.2.2. We shall subsequently compare the performance of the *Scan* and *Anemone* breath-first search model checker on various examples of the rush hour puzzle coded in three ways:

- with procedures written *without the guarded list construct*,
- with procedures written *with the guarded list construct*, as they result from the refinement of Subsection 5.2.2,
- with the `Syst` procedures written *with the guarded list construct and global search*, as they result from the transformations exposed in Subsection 5.2.3.

These three kinds of procedures will subsequently be referred to by the labels "Without GL", "With GL" and "GL with Syst".

Case	**Nb's cars/trucks**	**Game**
1	2	VPurpleTruck(2,1), HRedCar(3,2)
2	3	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1)
3	4	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1)
4	5	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4)
5	6	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4), HGreenTruck(6,3)
6	7	VPurpleTruck(2,1), HRedCar(3,2), HGreenCar(1,1), VOrangeCar(5,1), VBlueTruck(2,4), HGreenTruck(6,3), VYellowTruck(1,6)

Table 5.1: Test cases [17]

We base our comparison on the examples of Table 5.1. They are inspired by cards of the real game and are taken by progressively adding vehicles. The last column in Table 5.1 gives a brief description of the considered game. The V and H prefixes refer to a vehicle put vertically or horizontally, while the coordinates are those of the rows (counted from top to bottom) and columns (counted from left to right). Table 5.2 reports the data obtained on an x64-bit Lenovo laptop, running Windows 10 with 16GB of memory. The first column refers to the test case described in Table 5.1. The second, third and fifth columns show the time in milliseconds taken to run the model checker on the considered cases. The fourth and sixth columns present the gain obtained between the models, respectively, on the one hand, between the "Without GL" and "With GL" procedures, and, on the other hand, between the "With GL" and "GL with Syst" procedures.

As we can see from this table, the introduction of guarded lists leads to a real gain of performance as so does the global transformation of Subsection 5.2.3. For cases 1, 2 and 4, the gain in the third column is close to the expected gain of 2^n announced at the end of Section 5.2.2, with n the number of vehicles. Indeed, we get a gain of 2.7, 6 and 13.9, for an expected gain of 4, 8 and 16, respectively. It is worth observing that a deviation with respect to the theoretical gain is to be expected since we stop the computations once we have found a solution and thus only compute the necessary states (and not all of them). This also explains why case 3 is an exception to the increase of time with the addition of vehicles. In fact, because of their positions, the purple truck and the green car cannot move much whereas the red car is completely unconstrained. In these conditions, the breath-first search algorithm of the Anemone model checker discovers a solution very rapidly and thus computes less states. It is also worth observing that from case 4, the gain is higher than the one delivered by formula 2^n. Again this is due to the fact that the GL variants find solutions in a quite fast way. For case 6, we were not able to find a solution after computing 10 hours and generating more than 200 000 states. We thus stopped the computations at this point. Interestingly a result is found after 7 562 ms (namely about 7 seconds) with the guarded list construct. Finally it is worth observing that the "GL with Syst" transformation allows a further increase of performance and a much more reasonable computation time for the more complicated cases.

Case	Without GL	With GL	Gain	GL with Syst	Gain
1	175	63	2.7	63	1
2	815	136	6	102	1.3
3	285	90	3.2	72	1.2
4	3 401	245	13.9	118	2
5	1 258 577	801	1 571	174	4.6
6	≥ 10 h	7 562	/	654	11.6

Table 5.2: Performance results wrt execution time (in ms) [17]

Case	Without GL	With GL	Gain	GL with Syst	Gain
1	208	30	7	20	1.5
2	832	86	9.7	52	1.6
3	270	42	6.4	25	1.7
4	1 822	157	11.6	56	2.8
5	28 812	449	64.2	80	5.6
6	≥ 200000	2 534	/	280	9

Table 5.3: Performance results wrt number of states generated [17]

Table 5.3 provides another look at the computations by comparing the number of states computed during the executions. It fully meets the conclusions drawn for Table 5.2.

5.4 Related Work

As far as we know, the idea of forcing statements to be executed without interruption is not new, nor has it been exploited by coordination languages. In [47] Dijkstra has introduced guarded commands, which are statements of the form of $G \rightarrow S$ that atomically executes statement S provided the condition G is evaluated to true. They are mostly combined in repetitive constructs of the form

do $G_0 \rightarrow S_0$
□ $G_1 \rightarrow S_1$
$\cdots$
□ $G_n \rightarrow S_n$
od

which repetitively selects one of the executable guarded commands until none of them are executable. A non-deterministic choice is operated in the selection of the guarded commands in case several of them can be executed. Later, Abrial has used guarded commands in the Event-B method [1]. Such a construct is also at the core of the guarded Horn clause framework proposed by Ueda in [119] to introduce parallelism in logic programming. There Horn clauses are rewritten in the following form

$$H \leftarrow G_1, \cdots, G_m | B_1, \cdots, B_n$$

with H, G_1, …, G_m, B_1, …, B_n being atoms. The classical SLD-resolution used to reduce an atom is modified as follows. Assume A is the atom to be reduced. All the clauses whose head H is unifiable with A have their guard $G_1, \cdots, G_m$ evaluated. The first one that succeeds determines the clause that is used, and the other is simply discarded. To avoid mismatching instantiations of variables, the evaluation of any G_i is suspended if it can only succeed by binding variables. Finally,

several pieces of work have tried to incorporate transactions and atomic constructs in "classical" process algebras, like CCS. For instance, A2CCS [57] proposes to refine complex actions into sequences of elementary ones by modelling atomic behaviours at two levels, with so-called high-level actions being decomposed into atomic sequences of low-level actions. To enforce isolation, atomic sequences are required to go into a special invisible state during all their execution. In fact, sequences of elementary actions are executed sequentially, without interleaving with other actions, as though in a critical Section. RCCS [40] is another process algebra incorporating distributed backtracking to handle transactions inside CCS. The main idea is that, in RCCS, each process has access to a log of its synchronization history and may always wind back to a previous state. A similar idea of log is used in AtCCS [2]. There, during the evaluation of an atomic block, actions are recorded in a private log and have no effects outside the scope of the transaction until it is committed. An explicit termination action "end" is used to signal that a transaction is finished and should be committed. States are used in addition to model the evaluation of expressions and can be viewed as tuples put or retrieved from shared spaces in coordination languages. When a transaction has reached commitment and if the local state meets the global one, then all actions present in the log are performed at the same time and the transaction is closed. Otherwise the transaction is aborted.

Our guarded list constructs [16] shares similarities with these pieces of work. A major difference is however that we restrict the guard to a single primitive to be evaluated. This eases the implementation since, once the primitive has been successfully evaluated, the remaining primitives can be executed in a row without using distributed backtracking as in RCCS, private spaces as in AtCCS for speculative computations and checks for compatibility between local and global environments. Intricate suspensions inherent in guarded Horn clauses are also avoided. Nevertheless, under this restriction, the combination with the non-deterministic choice operator allows for achieving computations similar to the repetitive statements of guarded commands. In relation to these studies, our contribution lies in emphasizing model checking and presenting a refinement strategy that facilitates the transformation of programs through the incorporation of the guarded list construct.

Limiting the state explosion problem in model checking by limiting interleaving is similar in spirit to the partial-order reduction introduced in [55, 84, 97, 120]. Realizing that n independent parallel transitions result in $n!$ different orderings and 2^n different states, the idea is to select a representative composed of $n+1$ states. Indeed, as the transitions are independent, properties need only to be verified on a possible ordering. This technique has been employed in many research efforts for model-checking asynchronous systems. However, these efforts aim at designing more efficient algorithms on optimized automata. The approach taken here is different. We do not change our algorithm for model checking, but rather introduce a new construct as well as considerations on refinements to transform programs into more efficient programs.

5.5 Closing Remarks

With the goal of enhancing the performance of formulae verification in the Scan [63] and Anemone [62] workbenchs, this chapter introduced a new construct called the guarded list. This construction improved efficiency in the verification phase for Linda-like languages. Additionally, to facilitate the safe introduction of the guarded list into programs, we proposed a notion of refinement and identified situations in which a sequence of primitives can be safely replaced by a guarded list of primitives.

In the next chapter, we shall turn to expressiveness issues and establish that introducing guarded lists also results in an increase of expressiveness.

CHAPTER 6

GUARDED LIST CONCEPT: EXPRESSIVENESS EVALUATION

This chapter explores how guarded lists enhance the expressiveness of the Multi-Bach *coordination model. More precisely using modular embedding proposed by De Boer and Palamidessi, we shall relate the $\mathscr{L}_g$ family of languages embodying guarded lists with the $\mathscr{L}_r$ family of languages that do not contain guarded Lists. We shall establish that the $\mathscr{L}_r$ languages can be embedded in their $\mathscr{L}_g$ counterpart but not vice-versa. Moreover we shall prove that $\mathscr{L}_g$ languages obey to the expressiveness hierarchy of traditional coordination languages. These results illustrate that guarded lists not only bring an increase of performance to verify formulae but also an increase of expressiveness.*

6.1 Opening

In the previous chapter, we introduced the concept of guarded lists, emphasizing their value in improving performance for formulae verification by mitigating the state-space explosion. However, the benefits of guarded lists extend beyond efficiency. This chapter delves into another crucial advantage: the increased expressiveness they bring to the Multi-Bach coordination model.

Expressiveness in programming languages often reflects the ability to represent constructs from one language within another without requiring significant reorganization of the program, presented in Section 6.2. For sequential languages, this typically involves preserving termination. For concurrent languages, expressiveness must also account for system behaviours, such as different computational evolutions and deadlocks.

In this chapter, we explore how guarded lists enhance the expressive power of the Multi-Bach language family, in Section 6.3. By leveraging the concept of modular embedding, we compare the expressiveness of languages with and without guarded lists. We show that languages with guarded lists can embed their counterparts without them, demonstrating their increased versatility. Conversely, we highlight constructs enabled by guarded lists that cannot be replicated by languages lacking this feature. The discussion will include operational semantics for concurrent languages, embedding hierarchies, and detailed comparisons between the $\mathscr{L}_r$ and $\mathscr{L}_g$ families of languages, in Section 6.4. Next, in Section 6.5, we summarize the results to clarify the fundamental advantages guarded lists bring to Multi-Bach's language ecosystem. Finally, Section 6.6 concludes this chapter.

6.2 Expressiveness

The introduction of guarded lists has been motivated by a gain of efficiency during verification. It is interesting to observe that the guarded list construct also brings an increase of expressiveness. This is evidenced in this section by using the notion of modular embedding introduced in [21]. As pointed out there, from a computational point of view, all "reasonable" sequential programming languages are equivalent, as they express the same class of functions. Still, it is common practice to speak about the "power" of a language on the basis of the expressibility or non-expressibility of programming constructs. In general, a sequential language L is considered to be more expressive than another sequential language L' if the constructs of L' can be translated in L without requiring a "global reorganization of the program" [50], that is, in a compositional way. Of course, the translation must preserve the meaning, at least in the weak sense of preserving termination.

When considering concurrent languages, the notion of termination must be reconsidered, as each possible computation represents a possible different evolution of a system of interacting processes. Moreover, deadlock represents an additional case of termination. Following previous work [29, 28, 43, 74, 75, 76], we shall consequently rely on the following operational semantics $\mathscr{O}_f$ provided by Definition 20, which focuses on the final store of finite computations together with the termination mark.

Definition 20. *Define the* ***operational semantics*** $\mathscr{O}_f : \mathscr{L}_g \to \mathscr{P}(Sstore \times \{\delta^+, \delta^-\})$ *as the following function: for any agent* $A \in \mathscr{L}_g$

$$\begin{aligned} \mathscr{O}_f(A) \quad = \quad & \{(\sigma, \delta^+) : \langle A \mid \emptyset \rangle \to^* \langle E \mid \sigma \rangle\} \\ & \cup \\ & \{(\sigma, \delta^-) : \langle A \mid \emptyset \rangle \to^* \langle B \mid \sigma \rangle \not\to, B \neq E\} \end{aligned}$$

Note that (as it is immediately verified), for any agent A, the semantics $\mathscr{O}_f(A)$ is obtained by considering the final stores of the finite histories of $\mathscr{O}_h(A)(\emptyset)$.

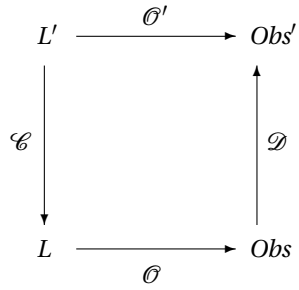


Figure 6.1: Basic embedding

The basic definition of embedding, given by Shapiro [113] is the following. Consider two languages L and L'. Moreover assume we are given the semantics mappings $\mathcal{O} : L \to Obs$ and $\mathcal{O}' : L' \to Obs'$, where Obs and Obs' are some suitable domains. Then L can **embed** L' if there exists a mapping $\mathscr{C}$ (**coder**) from the statements of L' to the statements of L, and a mapping $\mathscr{D}$ (**decoder**) from Obs to Obs', such that the diagram of Figure 6.1 commutes, namely such that for every statement $A \in L'$: $\mathscr{D}(\mathcal{O}(\mathscr{C}(A))) = \mathcal{O}'(A)$.

The basic notion of embedding is too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [21] to add three constraints on the coder $\mathscr{C}$ and on the decoder $\mathscr{D}$ in order to obtain a notion of **modular** embedding usable for concurrent languages:

(i) $\mathscr{D}$ should be defined in an element-wise way with respect to $\mathcal{O}$:

$$\forall X \in Obs\colon \mathscr{D}(X) = \{\mathscr{D}_{el}(x) \mid x \in X\} \tag{P_1}$$

for some appropriate mapping $\mathscr{D}_{el}$;

(ii) the coder $\mathscr{C}$ should be defined in a compositional way with respect to the sequential, parallel and choice operators[1]:

$$\begin{aligned} \mathscr{C}(A\,;\,B) &= \mathscr{C}(A)\,;\,\mathscr{C}(B) \\ \mathscr{C}(A \,||\, B) &= \mathscr{C}(A) \,||\, \mathscr{C}(B) \\ \mathscr{C}(A \,+\, B) &= \mathscr{C}(A) \,+\, \mathscr{C}(B) \end{aligned} \tag{P_2}$$

(iii) the embedding should preserve the behavior of the original processes with respect to deadlock, failure and success (**termination invariance**):

$$\forall X \in Obs, \forall x \in X\colon\ tm'(\mathscr{D}_{el}(x)) = tm(x) \tag{P_3}$$

where tm and tm' extract the information on termination from the observables of L and L', respectively.

[1] Actually, this is not required for the sequential operator in [21] since it does not occur in that work.

An embedding is then called **modular** if it satisfies properties P_1, P_2, and P_3.

The existence of a modular embedding from L' into L is denoted as $L' \leq L$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $L' \subseteq L$ then $L' \leq L$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting $\mathscr{C}$ and $\mathscr{D}$ equal to the identity function.

This background being provided, let us now compare the Anim-Bach language with guarded lists with the Anim-Bach language without guarded lists. As introduced before, the former is denoted by $\mathscr{L}_g$. The latter will be denoted by $\mathscr{L}_r$. Following [28], we shall also test three sublanguages composed (i) of the ask, tell primitives, (ii) of the ask, tell, get primitives and (iii) of the ask, tell, get, nask primitives. These sublanguages will be denoted by specifying the primitives between parentheses, as in $\mathscr{L}_g(ask, tell)$. Moreover, to focus on the core features, we shall discard conditional statements and procedures, which are essentially introduced for the ease of coding applications.

The article [29] has established that

$$\mathscr{L}_r(ask, tell) < \mathscr{L}_r(get, tell) = \mathscr{L}_r(ask, get, tell) < \mathscr{L}_r(ask, nask, get, tell).$$

It is to be expected, but to be verified, that similar relations hold for guarded lists. The intuition behind these relations is first that $\mathscr{L}_r(ask, tell)$ has a monotonic behavior : once a piece of information has been told, it cannot be removed later in the computations. This property obviously does not hold for languages involving the get primitive. Hence $\mathscr{L}_r(ask, tell)$ appears to be the weakest language. However, for any si-term t, the $ask(t)$ primitive can be coded by a $get(t)$ followed by $tell(t)$. Strictly speaking there is a difference since, in contrast to $ask(t)$ which does modify the store, get followed by $tell$ does modify it. However one needs the $nask$ primitive to evince that property. As a result, in absence of $nask$, $\mathscr{L}_r(get, tell)$ and $\mathscr{L}_r(ask, get, tell)$ have the same expressive power but both of them are less expressive than the full language $\mathscr{L}_r(ask, nask, get, tell)$.

Moreover, by language inclusion, it is obvious that the Multi-Bach sublanguages with guarded lists embed their counterparts without guarded lists. We announced before that the converse does not hold. Intuitively, the key ingredient for these properties is that, thanks to the guarded list construct which provide atomic computations, languages of the $\mathscr{L}_g$ family have the possibility to atomically check the presence or absence of two or more pieces of information. This does not hold for $\mathscr{L}_r$ family as there is no atomic construct. One may however argue that using a special token, say s for semaphore, one may simulate atomic behaviors by first putting t on the store and then encapsulating between $get(s)$ and $tell(s)$ primitives critical code to be executed in isolation. However, this tends to forget the compositionality requirement imposed to the coder (see property P_2 above) which leads to mess up the execution, for instance when agents are executed in parallel.

All these explanations rely on intuition but call for formal proofs. They are provided in the following two sections. A final section sums up the results and put them in perspective.

6.3 Embedding Hierarchy in the $\mathscr{L}_g$ family

Let us start the comparison of languages of the $\mathscr{L}_g$ family of languages by comparing $\mathscr{L}_g(ask, tell)$ and $\mathscr{L}_g(get, tell)$. Intuitively, all the computations of $\mathscr{L}_g(ask, tell)$ are monotonic in the sense that once told an si-term cannot be removed. This is not the case for $\mathscr{L}_g(get, tell)$ which can remove told si-terms by employing the get primitive. This result is formally established as follows.

Proposition 2. *One has $\mathscr{L}_g(ask, tell) < \mathscr{L}_g(get, tell)$*

Proof. On the one hand, by coding each $ask(t)$ primitive as $get(t)$ followed by $tell(t)$, one immediately establish that $\mathscr{L}_g(ask, tell) \leq \mathscr{L}_g(get, tell)$. Let us now establish that the converse does not hold. Assume by contradiction that $\mathscr{L}_g(get, tell) \leq \mathscr{L}_g(ask, tell)$ hold and thus that there exists a coder from $\mathscr{L}_g(get, tell)$ to $\mathscr{L}_g(ask, tell)$. Let us consider $[tell(a)]\ ;\ [get(a)]$. Since $\mathscr{C}$ is compositional and since $\mathscr{O}_f([tell(a)]\ ;\ [get(a)]) = \{(\emptyset, \delta^+)\}$, the termination mark of any element of $\mathscr{O}_f(\mathscr{C}([tell(a)])\ ;\ \mathscr{C}([get(a)]))$ is successful. As $\mathscr{C}([get(a)])$ is composed of ask and tell primitives only and since ask and tell primitives do not destroy elements, it follows that any element of $\mathscr{O}_f(\mathscr{C}([tell(a)])\ ;\ \mathscr{C}([get(a)])\ ;\ \mathscr{C}([get(a)]))$ has a successful termination mark. However, $\mathscr{O}_f([tell(a)]\ ;\ [get(a)]\ ;\ [get(a)]) = \{(\emptyset, \delta^-)\}$ which contradicts property P_3. □

In the presence of the tell primitive, the get primitive does bring expressiveness with respect to the ask primitive. As a result, it is expected that it is redundant when the language contains both the tell and get primitives. This is indeed the case, as established by the following proposition.

Proposition 3. *One has $\mathscr{L}_g(get, tell) = \mathscr{L}_g(ask, get, tell)$.*

Proof. Indeed, $\mathscr{L}_g(get, tell) \leq \mathscr{L}_g(ask, get, tell)$ holds by language inclusion. Moreover, translating each get and tell primitive to itself and each $ask(t)$ primitive in $get(t)$ followed by $tell(t)$, establishes that $\mathscr{L}_g(ask, get, tell) \leq \mathscr{L}_g(get, tell)$. □

In contrast, it worth observing that nask strictly enhances the expressivity when combined with tell and get primitives.

Proposition 4. *One has $\mathscr{L}_g(get, tell) < \mathscr{L}_g(nask, get, tell)$.*

Proof. Indeed, by language inclusion, one has $\mathscr{L}_g(get, tell) \leq \mathscr{L}_g(nask, get, tell)$. Moreover, it is possible to establish by contradiction that $\mathscr{L}_g(get, tell) \leq \mathscr{L}_g(nask, get, tell)$ does not hold. Indeed otherwise it is possible to prove that any computation of $\mathscr{C}([tell(t)])\ ;\ \mathscr{C}([nask(t)])$ starting in the empty store is successful, which contradicts by P_3 the fact that $\mathscr{O}_f([tell(t)]\ ;\ [nask(t)]) = \{(\{t\}, \delta^-)\}$. In fact, any computation of $\mathscr{C}([tell(t)])$ starting on the empty store succeeds since $\mathscr{O}_f([tell(t)]) = \{(\{t\}, \delta^+)\}$. Similarly, any computation of $\mathscr{C}([nask(t)])$ starting on the empty store succeeds and consequently so does any general computation starting on any store since $\mathscr{C}([nask(t)])$ is composed of get and tell primitives only. It

follows that any computation of $\mathscr{C}([tell(t)])$ starting on the empty store can be continued by (successful) computations of $\mathscr{C}([nask(t)])$. □

Finally, in the presence of the other primitives, the ask primitive does not bring any additional expressiveness.

Proposition 5. *One has* $\mathscr{L}_g(nask, get, tell) = \mathscr{L}_g(ask, nask, get, tell)$.

Proof. By language inclusion, one has $\mathscr{L}_g(nask, get, tell) \leq \mathscr{L}_g(ask, nask, get, tell)$. Moreover, given the atomicity of the guarded list construct, it is possible to rewrite any $ask(t)$ primitive as $get(t)$ followed by $ask(t)$. □

Propositions 2 to 5 can be summarized in the following theorem, which depicts the hierarchy inside the $\mathscr{L}_g$ family of languages.

Theorem 1. *One has*

$$\begin{aligned} \mathscr{L}_g(ask, tell) &< \mathscr{L}_g(get, tell) = \mathscr{L}_g(ask, get, tell) \\ &< \mathscr{L}_g(nask, get, tell) = \mathscr{L}_g(ask, nask, get, tell). \end{aligned}$$

Proof. The theorem directly results from Propositions 2 to 5. □

6.4 Comparing the $\mathscr{L}_r$ and $\mathscr{L}_g$ Families of Languages

Let us now turn to the relations between the $\mathscr{L}_r$ and $\mathscr{L}_g$ families of languages. By language inclusion, a first obvious result is that the Multi-Bach sublanguages with guarded lists embed their counterparts without guarded lists.

Theorem 2. *For any subset* $\mathscr{X}$ *of primitives, one has* $\mathscr{L}_r(\mathscr{X}) \leq \mathscr{L}_g(\mathscr{X})$.

The converse relations do not hold. Intuitively, this is due to the fact that, in contrast to $\mathscr{L}_r$, the languages $\mathscr{L}_g$ have the possibility of atomically testing the simultaneous presence of two si-terms on the store. The formal proofs require of course deeper treatments. It turns out however that the techniques employed in [28] can be adapted to guarded lists. One of them, which results from classical concurrency theory, is that any agent can be reformulated in a so-called normal form.

Definition 21. *Agents (of* $\mathscr{L}_g$*) in normal forms are agents of* $\mathscr{L}_g$ *which obey the following grammar, where N is an agent in normal form, p is a primitive (either graphical or store-related) or a guarded list of primitives and A denotes an arbitrary (non restricted) agent*

$$N \quad ::= \quad p \mid p\,;\, A \mid N + N.$$

The following lemma is reproduced from [28].

Lemma 1. *For any agent A of* $\mathscr{L}_g$*, there is an agent N of* $\mathscr{L}_g$ *in normal form which has the same derivation sequences as A.*

Proof. The proof is reproduced from [28] to provide the reader with the construction of an agent in normal form associated to an unrestricted agent. It consists of associating to any agent A an agent $\tau(A)$ (in normal form) by using the following translation defined inductively on the structure of A:

$$\begin{aligned}
\tau(p) &= p \\
\tau(X;Y) &= \tau(X);Y \\
\tau(X+Y) &= \tau(X)+\tau(Y) \\
\tau(X \parallel Y) &= \tau(X)\between Y + \tau(Y)\between X
\end{aligned}$$

$$\begin{aligned}
p\between Z &= p;Z \\
(p;A)\between Z &= p;(A \parallel Z) \\
(N_1+N_2)\between Z &= N_1\between Z + N_2\between Z
\end{aligned}$$

It is easy to verify that, for any agent A, the agent $\tau(A)$ is in normal form. Moreover, it is straightforward to verify that A and $\tau(A)$ share the same derivation sequences. □

We are now in a position to establish that $\mathscr{L}_g(ask,tell)$ cannot be embedded in $\mathscr{L}_r(ask,tell)$.

Proposition 6. *One has* $\mathscr{L}_g(ask,tell) \not\leq \mathscr{L}_r(ask,tell)$

Proof. Let us proceed by contradiction and assume the existence of a coder $\mathscr{C}$ and a decoder $\mathscr{D}$. The proof is composed of three main steps.

STEP 1: on the coding of $tell(a)$ and $tell(b)$. Let a, b be two distinct si-terms. Since $\mathscr{O}_f([tell(a)]) = \{(\{a\},\delta^+)\}$, any computation of $\mathscr{C}([tell(a)])$ starting in the empty store succeeds by property P_3. Let

$$\langle \mathscr{C}([tell(a)]) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \{a_1,\cdots,a_m\}\rangle$$

be one computation of $\mathscr{C}([tell(a)])$. Similarly, any computation of $\mathscr{C}([tell(b)])$ starting on the empty store succeeds. Let

$$\langle \mathscr{C}([tell(b)]) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \{b_1,\cdots,b_n\}\rangle$$

be one computation of $\mathscr{C}([tell(b)])$. Note that, as we only consider ask and tell primitives, this computation can be reproduced on any store τ. We thus have also that

$$\langle \mathscr{C}([tell(b)]) \mid \tau\rangle \longrightarrow \cdots \longrightarrow \langle E \mid \tau \cup \{b_1,\cdots,b_n\}\rangle$$

In particular, as $\mathscr{C}([tell(a)];[tell(b)]) = \mathscr{C}([tell(a)]);\mathscr{C}([tell(b)])$, we have that

$$\begin{aligned}
\langle \mathscr{C}([tell(a)];[tell(b)]) \mid \emptyset\rangle &\longrightarrow \cdots \\
&\longrightarrow \langle \mathscr{C}([tell(b)]) \mid \{a_1,\cdots,a_m\}\rangle \longrightarrow \cdots \\
&\longrightarrow \langle E \mid \{a_1,\cdots,a_m,b_1,\cdots,b_n\}\rangle
\end{aligned}$$

STEP 2: coding of an auxiliary statement AB. Consider now $AB = [ask(a) \rightarrow ask(b)]$. Obviously, as it requires a to be present, the execution of AB on the empty store cannot do any step and thus $\mathscr{O}_f(AB) = \{(\emptyset, \delta^-)\}$. Let us now turn to its coding $\mathscr{C}(AB)$. By Proposition 1, it can be regarded in its normal form. As it is in $\mathscr{L}_r(tell, ask)$, its more general form is as follows

$$\begin{aligned} & tell(t_1)\,;\, A_1 + \cdots + tell(t_p)\,;\, A_p \\ & + ask(u_1)\,;\, B_1 + \cdots + ask(u_q)\,;\, B_q \\ & + gp_1\,;\, C_1 + \cdots + gp_r\,;\, C_r \end{aligned}$$

where $gp_1, \ldots, gp_r$ are graphical primitives. Let us first establish that there is no alternative guarded by a $tell(t_i)$ operation. Indeed, if this was the case, then

$$D = \langle \mathscr{C}(AB) \mid \emptyset \rangle \longrightarrow \langle A_i \mid \{t_i\} \rangle$$

would be a valid computation prefix of $\mathscr{C}(AB)$. As $\mathscr{O}_f(AB) = \{(\emptyset, \delta^-)\}$, this prefix should deadlock afterwards. However, as $\mathscr{C}(AB + [tell(a)]) = \mathscr{C}(AB) + \mathscr{C}([tell(a)])$, the computation step D is also a valid computation prefix of $\mathscr{C}(AB + [tell(a)])$. Hence, $\mathscr{C}(AB + [tell(a)])$ admits a failing computation which, by property P_3, contradicts the fact that $\mathscr{O}_f(AB + [tell(a)]) = \{(\{a\}, \delta^+)\}$. The proof of the absence of an alternative guarded by a graphical primitive gp_i proceeds similarly.

Let us now establish that none of the u_i's belong to $\{a_1, \cdots, a_m\} \cup \{b_1, \cdots, b_n\}$. Indeed, if $u_j \in \{a_1, \cdots, a_m\}$ for some $j \in \{1, \cdots, q\}$, then, as $\mathscr{C}([tell(a)]\,;\, AB) = \mathscr{C}([tell(a)])\,;\, \mathscr{C}(AB)$, the derivation

$$\begin{aligned} D' = \langle \mathscr{C}([tell(a)]\,;\, AB) \mid \emptyset \rangle \longrightarrow \cdots \longrightarrow & \langle \mathscr{C}(AB) \mid \{a_1, \cdots, a_m\} \rangle \\ \longrightarrow & \langle B_j \mid \{a_1, \cdots, a_m\} \rangle \end{aligned}$$

is a valid computation prefix of $\mathscr{C}([tell(a)]\,;\, AB)$. However, by applying rule (T),

$$\langle [tell(a)]\,;\, AB \mid \emptyset \rangle \longrightarrow \langle AB \mid \{a\} \rangle \not\longrightarrow$$

By Property P_3, it follows that D' can only be continued by failing suffixes. However, thanks to the fact that $\mathscr{C}([tell(a)]\,;\, (AB + [ask(a)])) = \mathscr{C}([tell(a)])\,;\, (\mathscr{C}(AB) + \mathscr{C}([ask(a)]))$ the prefix D' induces the following computation prefix D'' for $\mathscr{C}([tell(a)]\,;\, (AB + [ask(a)]))$

$$\begin{aligned} D'' = \langle \mathscr{C}([tell(a)]\,;\, (AB + [ask(a)])) \mid \emptyset \rangle \longrightarrow & \cdots \\ \longrightarrow & \langle \mathscr{C}(AB) + \mathscr{C}([ask(a)]) \mid \{a_1, \cdots, a_m\} \rangle \\ \longrightarrow & \langle B_j \mid \{a_1, \cdots, a_m\} \rangle. \end{aligned}$$

which can only be continued by failing suffixes whereas $[tell(a)]\,;\, (AB + [ask(a)])$ only admits a successful computation.

The proof proceeds similarly in the case $u_j \in \{b_1, \cdots, b_n\}$ for some $j \in \{1, \cdots, q\}$ by then considering $[tell(b)]\,;\, AB$ and $[tell(b)]\,;\, (AB + [ask(b)])$.

STEP 3: combining the first two steps to produce a contradiction. The u_i's are thus forced not to belong to $\{a_1,\cdots,a_m\}\cup\{b_1,\cdots,b_n\}$. However, this induces a contradiction. To that end, let us first observe that $\mathscr{C}(AB)$ cannot do any step on the store $\{a_1,\cdots,a_m,b_1,\cdots,b_n\}$ since none of the $ask(u_i)$ primitives can do a step. As a result,

$$\langle AB \mid \{a_1,\cdots,a_m,b_1,\cdots,b_n\}\rangle \not\longrightarrow$$

Now, by compositionality of the coder with respect to the sequential composition (property P_2), $\mathscr{C}([tell(a)]\,;\,[tell(b)]\,;\,AB) = \mathscr{C}([tell(a)])\,;\,\mathscr{C}([tell(b)])\,;\,\mathscr{C}(AB)$, and consequently the following derivation is valid:

$$\langle \mathscr{C}([tell(a)]\,;\,[tell(b)]\,;\,AB) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle AB \mid \{a_1,\cdots,a_m,b_1,\cdots,b_n\}\rangle$$

and yields a failing computation for $\mathscr{C}([tell(a)]\,;\,[tell(b)]\,;\,AB)$. However, as easily checked, $[tell(a)]\,;\,[tell(b)]\,;\,AB$ has only one successful computation. □

Using similar arguments as in the previous proof it is possible to establish that the same result holds when the ask primitive is replaced by the get primitive.

Proposition 7. *One has* $\mathscr{L}_g(get,tell) \not\leq \mathscr{L}_r(get,tell)$.

Proof. The proof proceeds similarly, the only difference being that $ask(u_i)$ primitives need to be replaced by $get(u_i)$ primitives. □

As the ask primitive does not bring additional expressive power in the presence of the tell and get primitives, one then gets the following result.

Proposition 8. *One has* $\mathscr{L}_g(ask,get,tell) \not\leq \mathscr{L}_r(ask,get,tell)$.

Proof. This is a direct consequence of the previous proposition since $\mathscr{L}_g(get,tell) = \mathscr{L}_g(ask,get,tell)$ and $\mathscr{L}_r(get,tell) = \mathscr{L}_r(ask,get,tell)$. □

To introduce the nask primitive in the languages, one may hope of transposing once more the proof of proposition 8. However, since $\mathscr{L}_r(ask,nask,get,tell)$ can perform negative tests, $\mathscr{C}([tell(b)])$ could in principle check whether the $\mathscr{C}([tell(a)])$ has taken place before and then place or remove suitable tokens thereby making clear to $\mathscr{C}(AB)$ that both a and b have been told or not. The same reasoning would hold for telling the same token several times. However, as all the agents of the languages are finitely branching and contain a finite number of communication primitives, this is not possible for ever, as stated in the following lemmas. They have been first published in [28] to which we refer the reader for the proofs.

Lemma 2. *For any agent* $A \in \mathscr{L}_r(tell,ask,get,nask)$*, if* $(\sigma,\delta^+) \in \mathscr{O}_f(A)$ *then the parallel composition* $B = A \,||\, \cdots \,||\, A$ *of n copies of A has a successful computation resulting in the store* $\tau = \sigma\cup\cdots\cup\sigma$ *consisting of the multi-set union of n copies of* σ*:* $(\tau,\delta^+) \in \mathscr{O}_f(B)$.

Proof. See[28]. □

Lemma 3. *Let $A \in \mathscr{L}_r(tell, get, ask, nask)$ be such that $(\sigma, \delta^+) \in \mathscr{O}_f(A)$. Let B be an agent of $\mathscr{L}_r(tell, get, nask, ask)$ composed of n ask, nask and get primitives and such that $\langle B \mid \cup_{i=1}^{n+1}\sigma\rangle \longrightarrow^* \langle E \mid \tau\rangle$. Then, for any $p > 0$ $((\cup_{i=1}^{p}\sigma) \cup \tau, \delta^+) \in \mathscr{O}_f((\,||_{i=1}^{n+1+p} A)\,;\, B)$.*

Proof. See[28]. □

Lemma 4. *Let S be a finite set of si-terms and $f : \mathscr{I} \to \mathscr{P}_{fi}(\mathscr{I})$ be a function associating to each si-term a finite set of si-terms. Assume there is a si-term a such that for any other si-term b either $f(a) \cap f(b) \neq \emptyset$ or $f(b) \cap S \neq \emptyset$. Then there is a denumerable series of si-terms $(x_i)_i$ and an integer N such that $\bigcap_{i=1}^{N} f(x_i) \neq \emptyset$ and $\bigcap_{i=1}^{N} f(x_i) = \bigcap_{i=1}^{N} f(x_i) \cap f(x_n)$, for any $n > N$.*

Proof. See[28]. □

We are now in a position to establish our final proposition.

Proposition 9. *One has $\mathscr{L}_g(ask, nask, get, tell) \not\leq \mathscr{L}_r(ask, nask, get, tell)$.*

Proof. The proof is conducted by contradiction by assuming a coder $\mathscr{C}$ from $\mathscr{L}_g(ask, nask, get, tell)$ to $\mathscr{L}_r(ask, nask, get, tell)$. It is structured according to two cases following a set of notations that we first introduce.

NOTATIONS. Fix a si-term a and let n be the number of get and nask primitives in $\mathscr{C}([tell(a)])$. Moreover, for any primitive c and any integer p, let c^p denote the parallel composition of p copies of c: $c^p = ||_{i=1}^{p} c$.

As the computation of $[tell(b)]^{n+2}\,;\,[tell(a)]$ succeeds, for any si-term $b \neq a$, let S'_b be one store of one successful derivation of $\mathscr{C}([tell(b)])^{n+2}\,;\,\mathscr{C}([tell(a)])$. Moreover, take as S_x the store resulting from one successful derivation of $\mathscr{C}([tell(x)])$. Note that, thanks to Lemma 3, one may assume that $S_x \subseteq S'_x$, for any token x.

Two cases need to be considered.

CASE I. Assume first there is a si-term b such that $S_a \cap S'_b = \emptyset$. Then consider ABs consisting of removing a together with $n+3$ copies of b: $ABs = [get(a) \longrightarrow get(b), \cdots, get(b)]$. The normal form of $\mathscr{C}(ABs)$ can be rewritten in $\mathscr{L}_r(ask, nask, get, tell)$ as:

$$
\begin{aligned}
&tell(t_1)\,;\,A_1 + \cdots + tell(t_p)\,;\,A_p\\
&+ ask(u_1)\,;\,B_1 + \cdots + ask(u_q)\,;\,B_q\\
&+ get(v_1)\,;\,C_1 + \cdots + get(v_r)\,;\,C_r\\
&+ nask(w_1)\,;\,D_1 + \cdots + nask(w_s)\,;\,D_s\\
&+ gp_1\,;\,E_1 + \cdots + gp_g\,;\,E_g
\end{aligned}
$$

The proof proceeds in this case in three steps, with the first two reducing the alternatives and the last one bringing a contradiction in this context.

Step 1.a: no alternative with a tell primitive or a graphical primitive. Using arguments analogous to those of Proposition 6, it is possible to prove that there are no alternatives guarded by a $tell(t_i)$ primitive as well as by a gp_j graphical primitive.

Step 1.b: no alternative with a nask primitive. Let us now establish that $\{w_1,\cdots,w_s\} \subseteq S_a \cap S'_b$ which, in view of the hypothesis of this first case ($S_a \cap S'_b = \emptyset$) amounts to expressing that there are no alternatives guarded by a $nask(w_j)$ primitive. Indeed, assume that $w_j \notin S_a$, for some $j \in \{1,\cdots,s\}$. Then

$$Ep = \langle \mathscr{C}([tell(a)]\,;\,ABs) \mid \emptyset\rangle \longrightarrow \cdots \longrightarrow \langle \mathscr{C}(ABs) \mid S_a\rangle \longrightarrow \langle D_j \mid S_a\rangle$$

is a valid computation prefix for $\mathscr{C}([tell(a)]\,;\,ABs)$ which can only be continued by failing suffixes since $\mathscr{O}_f([tell(a)]\,;\,ABs) = \{(\{a\},\delta^-)\}$. However, this computation prefix induces the following computation prefix E' for the coder of $[tell(a)]\,;\,(ABs + [tell(b)]))$ which only admits successful computations:

$$\begin{aligned} E' = \langle \mathscr{C}([tell(a)]\,;\,(ABs + [tell(b)]) \mid \emptyset\rangle & \\ \longrightarrow \cdots & \\ \longrightarrow \langle \mathscr{C}(ABs) + \mathscr{C}([tell(b)]) \mid S_a\rangle & \\ \longrightarrow \langle D_j \mid S_a\rangle. & \end{aligned}$$

The proof proceeds similarly for $w_j \notin S'_b$ since then $w_j \notin S_b$.

Step 1.c: contradiction. Using the same reasoning as in Proposition 6, it is possible to establish that $\{u_1,\cdots,u_q,v_1,\cdots,v_r\} \cap (S_b \cup S'_b) = \emptyset$. As a result $\langle \mathscr{C}(ABs) \mid S_b \cup S'_b\rangle \not\longrightarrow$. The contradiction then comes from Lemma 3 which validates the following derivation:

$$\langle \mathscr{C}([tell(b)])^{n+3}\,;\,\mathscr{C}([tell(a)])\,;\,ABs \mid \emptyset\rangle \longrightarrow^* \langle \mathscr{C}(ABs) \mid S_b \cup S'_b\rangle.$$

The agent $\mathscr{C}([tell(b)])^{n+3}\,;\,\mathscr{C}(tell(a))\,;\,\mathscr{C}(ABs)$ then admits a failing computation whereas $[tell(b)]^{n+3}\,;\,[tell(a)]\,;\,ABs$ only has one successful computation.

CASE II. A key ingredient in the first case is the fact that there exist b such that $S_a \cap S'_b \neq \emptyset$. Imagine that such a b does not exist and thus that $S_a \cap S'_b \neq \emptyset$ for any si-term b distinct from a. Then, thanks to Lemma 4, there is a denumerable set of distinct si-terms x_i, also distinct from a and an integer m, such that $\cap_{i=1}^n (S_a \cap S'_{x_i}) \neq \emptyset$ and $[\cap_{i=1}^n (S_a \cap S'_{x_i})] \cap (S_a \cap S'_{x_j}) \neq \emptyset$, for $j > m$. Consider, in this case, $NT = [nask(a) \longrightarrow nask(x_1),\cdots,nask(x_n)]$ and $\mathscr{C}(NT)$ in the following normal form.

$$\begin{aligned} & tell(t_1)\,;\,A_1 + \cdots + tell(t_p)\,;\,A_p \\ & + ask(u_1)\,;\,B_1 + \cdots + ask(u_q)\,;\,B_q \\ & + get(v_1)\,;\,C_1 + \cdots + get(v_r)\,;\,C_r \\ & + nask(w_1)\,;\,D_1 + \cdots + nask(w_s)\,;\,D_s \\ & + gp_1\,;\,E_1 + \cdots + gp_g\,;\,E_g. \end{aligned}$$

The proof proceeds in this case in three steps.

Step 2.a : no alternative with a tell primitive or a graphical primitive. Let us first establish that there is no alternative guarded by a $tell(t_j)$ primitive. Indeed, if this was not the case then

$$F' = \langle \mathscr{C}([tell(x_1)]) \,;\, \mathscr{C}(NT) \mid \emptyset \rangle \longrightarrow \cdots \longrightarrow \langle \mathscr{C}(NT) \mid S_{x_1} \rangle \longrightarrow \langle A_j \mid S_{x_1} \cup \{t_j\} \rangle$$

would be, by properties (P_2), a valid computation prefix for $\mathscr{C}([tell(x_1) \,;\, NT)$ which, by (P_3), can only be continued by failing suffixes since $\mathscr{O}_f([tellx_1] \,;\, NT) = \{(\{x_1\}, \delta^-)\}$. However F' induces the following computation prefix F'' for $\mathscr{C}([tell(x_1)] \,;\, (NT + [ask(x_1)]))$ and thus a failing computation for it, which, by P_3 is absurd since the computation of $[tell(x_1)] \,;\, (NT + [ask(x_1)])$ succeeds:

$$\begin{aligned} F'' \quad = \quad & \langle \mathscr{C}([tell(x_1)]) \,;\, (\mathscr{C}(NT + [ask(x_1)])) \mid \emptyset \rangle \longrightarrow \cdots \\ & \longrightarrow \langle \mathscr{C}(NT + [ask(x_1)]) \mid S_{x_1} \rangle \longrightarrow \langle A_j \mid S_{x_1} \cup \{t_j\} \rangle \end{aligned}$$

The absence of alternatives involving graphical primitives is established similarly.

Step 2.b: auxiliary technical result. Let us now note prove that

$$\{w_1, \cdots, w_s\} \subseteq (S_{x_1} \cap \cdots \cap S_{x_n})$$

Indeed, if there was some $w_k \notin S_{x_i}$, then using similar reasonings for $[tell(x_i)] \,;\, NT$ and $[tell(x_i)] \,;\, (NT + [ask(x_i)])$ and employing the $nask(w_k) \,;\, D_k$ alternative of $\mathscr{C}(NT)$ would lead to a contradiction.

Step 2.c: contradiction. To establish the final contradiction, consider $[tell(x_{m+1})] \,;\, NT$. A possible computation prefix for $\mathscr{C}([tell(x_{m+1})] \,;\, NT)$ is, by P_2, as follows:

$$\langle \mathscr{C}([tell(x_{m+1})]) \,;\, \mathscr{C}(NT) \mid \emptyset \rangle \longrightarrow^* \langle \mathscr{C}(NT) \mid S_{x_{m+1}} \rangle.$$

Since $[tell(x_{m+1})] \,;\, NT$ has a successful computation, and since $\{w_1, \cdots, w_s\} \subseteq S_a \cap S_{x_1} \cap \cdots S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $u_j \in S_{x_{m+1}}$ or $v_j \in S_{x_{m+1}}$. Let us assume the first case hold, the other being treated similarly. In these conditions, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H = \langle \mathscr{C}([tell(x_{m+1})])^{n+2} \,;\, \mathscr{C}([tell(a)]) \,;\, \mathscr{C}(NT) \mid \emptyset \rangle \\ \longrightarrow^* \langle \mathscr{C}(NT) \mid S'_{x_{m+1}} \rangle \longrightarrow \langle B_j \mid S'_{x_{m+1}} \setminus \{u_j\} \rangle \end{aligned}$$

Moreover, as $[tell(x_{m+1})]^{n+2} \,;\, [tell(a)] \,;\, NT$ has only one failing computation, H should be continued by failing suffixes only. It follows that H introduces a failing computations for $[tell(x_{m+1})]^{n+2} \,;\, [tell(a)] \,;\, (NT + [get(a)])$ which is impossible in view of (P_3) since the latter agent has only one successful computation. □

The following theorem sums up the non embedding relations established in Propositions 6 to 9, which state the non-embedding of the languages $\mathscr{L}_g$ into their $\mathscr{L}_r$ counterparts.

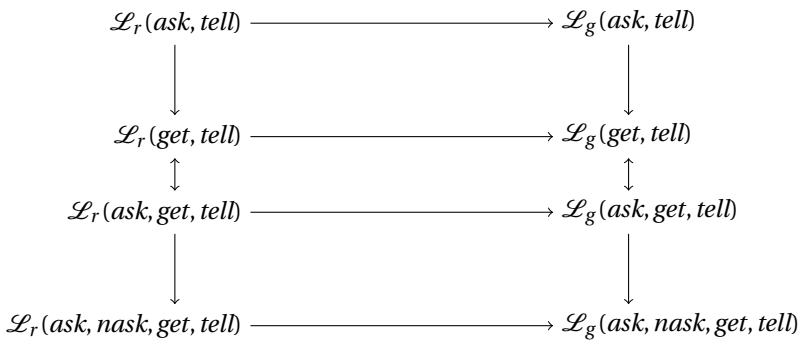


Figure 6.2: Relations between languages

Theorem 3. *For any subset $\mathscr{X}$ of primitives taken from*

$$\{\{ask, tell\}, \{get, tell\}, \{ask, get, tell\}, \{ask, nask, get, tell\}\}$$

one has $\mathscr{L}_g(\mathscr{X}) \not\leq \mathscr{L}_r(\mathscr{X})$.

Proof. The theorem follows directly from Propositions 6 to 9. □

6.5 Summary of the Relations Between the Languages

The set of separation and equivalence results are summarised in Figure 6.2. They follow directly from the results obtained in [29] for the $\mathscr{L}_r$ family of languages and from the results established in this paper in Theorems 1, 2 and 3.

In this figure, an arrow from a language $\mathscr{L}_1$ to a language $\mathscr{L}_2$ means that $\mathscr{L}_2$ embeds $\mathscr{L}_1$, that is $\mathscr{L}_1 \leq \mathscr{L}_2$. Notice that, thanks to the transitivity of embedding, the figure contains only a minimal amount of arrows. However, apart from these induced relations, no other relation holds. In particular, when there is one arrow from $\mathscr{L}_1$ to $\mathscr{L}_2$ but there is no arrow from $\mathscr{L}_2$ to $\mathscr{L}_1$, then $\mathscr{L}_1$ is strictly less expressive than $\mathscr{L}_2$, that is $\mathscr{L}_1 < \mathscr{L}_2$.

6.6 Closing Remarks

To improve the performance of the verification tools introduced in the Scan [63] and Anemone [62] workbenches, the previous chapter presented a new construct called the guarded list, designed to enhance the efficiency of the verification of temporal logic formulae. In this chapter, we demonstrated that this construct not only improves efficiency but also increases the expressiveness of Linda-like languages.

Our work paves the way for several avenues of future research. As far as expressiveness studies are concerned which will be presented in the next chapter, our

current approach applied to a few sub-languages, calls for more in-depth research encompassing all sub-languages and their comparison with the L_{MR} and L_{CS} language families studied in [29]. Although our approach is valid, other methods such as the absolute approach advocated by Zavattaro et al in [30] could offer different perspectives on the expressiveness hierarchy of languages. In addition, it would be interesting to explore expressiveness studies based on bisimulations and fully abstract semantics, as indicated in [98].

In the following part, we present our methodology for applying Anemone to socio-technical systems in Chapter 7, using a detailed theoretical example of emergency system management in an intelligent city. Then, in Chapter 8, we showcase several examples of Anemone's implementation and application in IoT projects conducted with Master's students from our faculty. These chapters highlight not only the flexibility and efficiency of Anemone in diverse contexts but also its potential to address real-world coordination challenges in complex and dynamic environments.

Part IV

Applying to Socio-Technical Systems

CHAPTER 7

TOWARDS A METHODOLOGY FOR CODING STS

This chapter presents a methodology for coding socio-technical systems by using the Multi-Bach *coordination language and the* Anemone *workbench presented in the previous chapters. As we tried to convince the reader previously,* Multi-Bach *and* Anemone *facilitate the modelling and the simulation by enabling agent-based coordination and visualizing interactions through animations. Still it is of importance to help the user of* Multi-Bach *and* Anemone *with guidelines that enable him to produce code from an informal specification of the problem at hand. To that end, our methodology involves five key steps which are addressed in subsequent sections. First, we analyse the scenario, in Section 7.2, and convert key data and behaviours in* Multi-Bach. *Then, we simulate the processes with visual feedback in Section 7.3. A case study demonstrates the application of our methodology in optimizing ambulance and traffic coordination during emergencies, highlighting improved response times and resource allocation.*

7.1 Opening

Urbanization is currently remodelling our way of life, with cities becoming denser and more complex every year. By 2050, according to a report by the United Nation [129], nearly 68% of the global population (approximately 6.7 billion people) are expected to live in urban areas, compared to 54% in 2015. This rapid growth puts immense pressure on essential services like healthcare, transportation, and emergency response systems.

Consider emergency medical services (EMS), where every second counts. Research shows that reducing ambulance response times by even one minute can save up to 10,000 lives annually in large cities like New York [20]. Yet, in cities such

as Brussels, ranked 10th globally for traffic congestion in 2024, the combination of dense populations and clogged roads creates significant barriers to delivering critical care on time [117]. With the number of vehicles and city populations set to rise significantly by 2050, the question becomes urgent: *How can we ensure cities remain accessible for life-saving interventions when it matters most?*

Smart cities offer a hopeful future. Defined as urban environments that use technology to enhance efficiency, sustainability, and quality of life, smart cities integrate IoT, artificial intelligence (AI), and real-time data analysis to solve urban challenges [129]. For example, IoT enabled traffic systems can dynamically adjust signals to prioritize emergency vehicles, while drones can deliver medical supplies directly to the scene of an emergency. According to the Global McKinsey Institute [61], these technologies have the potential to reduce emergency response times by up to 35%, saving thousands of lives annually. This is where our research comes in. We shall use our coordination language Multi-Bach and its associated workbench Anemone to code the interactions between human and technical agents in the socio-technical system offered by emergency response networks. By modelling how agents like paramedics, hospitals, and traffic systems interact, they will be able to address challenges like traffic delays, communication breakdowns, and resource allocation. With this tool, we aim to demonstrate how cities can become smarter, more efficient, and better prepared to save lives as urbanization continues to grow.

To be more concrete, image the following scenario, which we shall refer to as *Nurse Geek and Ambulance Chic: Smart Cities to the Rescue*

> It is Thursday evening. After a long day of lectures, a nursing student heads home, only to suddenly find herself face-to-face with a man collapsing in the street. He is unresponsive. She performs first aid, but there is no reaction. He needs to be taken to the hospital immediately. However, it is 5 p.m., rush hour, the worst possible time for navigating the city streets. The ambulance may not arrive in time.
>
> Now envision urban roads in 2050, with a significant population increase expected in the next 25 years. Traffic delays could become critical, turning emergencies like this into life-or-death situations. What can we do today to ensure our cities remain accessible and responsive in the future? This brings us to the concept of “smart cities”. These cities leverage technology to enhance urban life, making it more efficient, sustainable, and liveable. Examples include smart systems that estimate traffic delays or measure air quality. But smart cities are about more than just monitoring, they drive proactive solutions. They aim to improve access to services. If so, the question is here how can we prepare cities to handle the predicted surge in vehicles by 2050 while maintaining efficiency and accessibility?

Figure 7.1: Nurse Geek and Ambulance Chic: Smart Cities to the Rescue in 2050

The image above has been created using the “Microsoft Bing” artificial intelligence tool to illustrate our “Nurse Geek and Ambulance Chic: Smart Cities to the Rescue in 2050” case study. This case study has been previously presented at the University of Namur’s 180 Second Thesis Presentation competition, on April 11, 2024.

7.2 Modelling “Nurse Geek and Ambulance Chic”

7.2.1 Data Definition and Elementary Operations

To start with one needs to make precise which information is used in our modelling and how elementary synchronization points are obtained. This leads to the following two subsections.

Data Representation

The *Nurse Geek and Ambulance Chic* case study involves three types of data that need to be suitably represented to enable efficient collection, processing, and analysis in decision-making:

(i) **Citizen Data**: Data related to citizens is crucial for personalized and collective services in a Smart City, such as healthcare and navigation assistance. Examples include:

- Elementary pieces of information, which will be represented as tokens. For example, Manel as a citizen will be represented by the token `manel`.
- Structured pieces of information, which will be represented by complex si-terms. For example, a report of an emergency by Manel at location 45 will be represented by the si-term `reportEmergency(manel, emergency, location_45)`.

(ii) **IoT Integration**: IoT devices provide real-time monitoring and reporting on various aspects of the city's ecosystem. Examples include:

- the declaration of web cams, which will be represented by tokens, as in `camTraffic_23`.
- the relation between data, which will be represented by complex si-terms, as in `trafficStatus(camTraffic_23, congestion)` to indicate the detection traffic congestion near the webcam 23.

(iii) **Urban Resource Allocation**: Urban data encompasses information about the city's infrastructure and environment, essential for optimizing resources and enhancing city operations. Examples include:

- the identification of objects, which will be represented by tokens, as `ambulance_09` for ambulance 9.
- the expression of properties, which will be represented by complex si-terms. For instance, `dispatchAmbulance(ambulance_09, location_45)` expresses that ambulance 9 is dispatched to location 45.

It is worth noting that the above examples constitute only a few illustrations of what need to be coded. Modelling completely the *Nurse Geek and Ambulance Chic* example will require to introduce other similar data. However, with what has been introduced above, one may already easily handle the following sentence.

> Let's imagine that `manel` is a citizen and a nursing student. After her classes, she strolls around the smart city in the year 2050. At one point, while at `location_45` in this intelligent city, she encounters an emergency situation that she cannot handle alone.

Elementary Points of Synchronization

Coding the *Nurse Geek and Ambulance Chic* case study involves four main operations of synchronization, which can be nicely coded by the Bach primitives acting on stores:

(i) reporting information, which can be coded by the *tell* primitive. For instance, `tell(reportEmergency(manel, emergency, location_45))` may be used to logs an emergency report by Manel at location 45.
(ii) checking whether an event occur, which can be coded by the *ask* primitive. For instance, `ask(status(manel, emergency, treated))` may be employed to check if Manel's emergency report has been addressed.
(iii) removing obsolete information, which can be coded by the *get* primitive. For instance, `get(reportEmergency(manel, emergency, location_45))` may be used to remove a resolved report.
(iv) verifying the absence of problems, which can be coded by the *nask* primitive. For instance `nask(reportEmergency(manel, emergency, location_45))` may be used to verify the absence of an emergency report at location 45.

7.2.2 Complex Behaviours

Procedures allow to code complex behaviours. To start with, let us describe each actor type of the *Nurse Geek and Ambulance Chic* case study by a procedure and each concrete actor by a procedure call. The *Nurse Geek and Ambulance Chic* case study obviously features five type of actors: citizens, ambulances, hospitals, traffic systems and grid systems. Each type will be depicted by a procedure with the same name to which we add an argument to identify particular instances. As a result,

- the `Citizen(id)` procedure is used to describe citizens
- the `Ambulance(id)` procedure is used to describe ambulances
- the `Hospital(id)` procedure is used to describe hospitals
- the `TrafficSystem(id)` procedure is used to describe traffic systems
- the `SmartGrid(smartGrid)` procedure is used to describe grids.

By putting in parallel calls to these procedures one may simulate the smart city system at end. For instance,

```
1 Citizen(manel)       ||  Citizen(fatma)       ||
2 Ambulance(1)         ||  Ambulance(2)         ||
3 Hospital(chu)        ||  Hospital(bouge)      ||
4 TrafficSystem(parc)  ||  TrafficSystem(gare)  ||
5
6 GridSystem(ville)
```

model a smart city involving two citizens, Manel and Fatma, two ambulances, identified as 1 and 2, two hospitals, named after the CHU UCL Namur hospital and the Saint-Luc Bouge hospital, two traffic lights, located near the park Marie-Louise and the Namur station as well as a grid system operated by the city of Namur.

It is worth observing how easy it is to add new actor types and new actors. One has just to define new procedures and include new procedure calls in the parallel composition.

Let us now describe in turn the actor types or, restated in programming terms, let us now code the associated procedures.

Modelling a Citizen

The behaviour of a citizen identified by id in the city's emergency response system consists of login, using smart devices, reporting emergency situations and finally login out. This is reflected in the following piece of code.

```
1 proc  Citizen(id:Ids) = LogIn(id);
2                         UseSmartDevices(id);
3                         ReportEmergency(id);
4                         ask(emergencyProcessed(id));
5                         LogOut(id);
6                         Citizen(id).
```

Doing a log in or a log out is of no great interest for our modelling. They are respectively modelled through the telling and getting of the presence of the citizen, as follows:

```
1 proc  LogIn(id:Ids)  = tell(logged(id)).
2 proc  LogOut(id:Ids) = get(logged(id)).
```

The goal of the procedure *UseSmartDevices* is to share real-time data by analysing health monitor data and processing traffic camera feeds. Dually, the *ReportEmergency* procedure aims at reporting emergency with details (ID, location, sensor data) and confirms its processing. In both cases, this is modelled by telling suitable si-terms, as follows.

```
1 proc  UseSmartDevices(id:Ids) =
2           tell(analyzeHealthData(healthMonitor));
3           tell(processTrafficData(trafficCamera)).
4
5 proc  ReportEmergency(id:Ids) =
6           tell(reportEmergency(id, location, sensorData));
7           ask(emergencyProcessed(id));
8           nask(reportEmergency(id, location, sensorData)).
```

Modelling an Emergency System

The behaviour of an emergency dispatch system consists in retrieving the emergency report, evaluating it, notifying an hospital and traffic system, and confirming the emergency has been processed.

```
proc EmergencyDispatch(id:Ids) =
     get(reportEmergency(id, location, sensorData));
     EvaluateEmergency(id);
     NotifyHospital(id);
     NotifyTrafficSystem(id);
     ask(emergencyProcessed(id)).
     nask(reportEmergency(id, location, sensorData)).
```

The auxiliary `EvaluateEmergency`, `NotifyHospital` and `NotifyTrafficSystem` procedures are coded in a similar way as the above auxiliary procedures, by using *tell*, *ask*, *nask* and *get* primitives. For instance, `EvaluateEmergency` is coded as follows:

```
proc EvaluateEmergency(id:Ids) = tell(emergencyProcessed(id)).
```

Modelling a Traffic System

The behaviour of a traffic system involves retrieving ambulance route data, adjusting traffic lights dynamically using traffic camera input, and prioritizing the ambulance route. A message confirms the adjustments. This leads to the following code.

```
proc TrafficSystem(id:Ids) = get(adjustTraffic(id, ambulanceRoute));
                             AdjustTrafficLights(id).
AdjustTrafficLights(id:Ids) = tell(processTrafficData(trafficCamera));
                  nask(trafficJam(id)).
```

Modelling an Hospital System

The behaviour of an hospital involves retrieving dispatched information, preparing the ambulance with optimized resources, dispatching it to the location, and sending notifications upon its arrival and the start of patient treatment.

```
proc Hospital(id:Ids) = get(dispatchAmbulance(id, location));
                        PrepareAmbulance(id);
                        ask(ambulanceArrives(id, location));
                        DispatchAmbulance(id).
     DispatchAmbulance(id:Ids) = tell(ambulanceArrives(id, location));
                                 tell(treatPatient(id, location)).
```

Modelling an Ambulance System

The behaviour of an ambulance involves receiving arrival information, optimizing its route with GPS and traffic data, treating the patient using connected devices, and returning to the hospital. This is coded as follows.

```
1 proc Ambulance(id:Ids) = get(ambulanceArrives(id, location));
2                          ArriveAtEmergency(id);
3                          ask(patientReady(id));
4                          TreatPatient(id);
5                          nask(ambulanceBlocked(id, location));
6                          ReturnToHospital(id).
```

Modelling a Smart City System

The behaviour of a smart grid involves processing live traffic data, monitoring energy consumption, and dynamically prioritizing the power supply for emergency systems.

```
1 proc SmartGrid(id:Ids) = ask(emergencyDetected(id));
2                          tell(processTrafficData(trafficCamera));
3                          MonitorEnergyConsumption(id);
4                          nask(powerFailure(id));
5                          ask(emergencyResolved(id));
6                          AdjustPowerAllocation(id).
```

7.3 Simulation

By its graphical capabilities and the code inspection it offers, Anemone allows to simulate and debug the code just sketched. To that end, the following steps have to be undertaken:

(i) identify a background drawing for rendering graphically the application
(ii) identify objects to be draw and associate them to images, possibly varying according to the object state
(iii) associate a widget to each object and use attributes to display the image according to the object state
(iv) identify points in the program in which objects have to move or attributes have to be changed
(v) introduce the corresponding `move` and `att` primitives at these points.

Example 19. *As an illustration, consider the following code snippet:*

```
eset RCInt        = { 1, 2, 3, 4, 5, 6 }.
     Colors       = { trafficLightRed, trafficLightGreen }.
     RCPlaces     = { in_grid, out_grid }.
     Orientation = { v_orientation, h_orientation}.

scene ngScene = {
   size = (562,618).
   layers = { top }.
   background = loadImage(Images/city_map.png).

   ambulanceIdle       = loadImage(Images/ambulance_idle.png).
   ambulanceMoving_v   = loadImage(Images/ambulance_moving_v.png).
   ambulanceMoving_h   = loadImage(Images/ambulance_moving_h.png).
   trafficLightRed     = loadImage(Images/traffic_light_red.png).
   trafficLightGreen   = loadImage(Images/traffic_light_green.png).
     ...
 widget ambulance     =
  { attributes= { sens in Orientation. }
    display    = { sens = v_orientation -> ambulance_moving_v.
                   sens = h_orientation -> ambulance_moving_h.}
     init      = { wdL = top.
                   sens = h_orientation. }
  }

 widget trafficLight  =
  { attributes= { situation in Colors. }
    display    = { situation = trafficLightRed   -> traffic_light_red.
                   situation = trafficLightGreen -> traffic_light_green.}
    init      = { wdL = top.
                  situation = trafficLightRed. }
   }
     ...
 }.
proc Ambulance(id:Ids) = get(ambulanceArrives(id, location));
     move_to(ambulance,ngScene,x_coord(location),y_coord(location));
     ArriveAtEmergency(id);
     ask(patientReady(id));
     TreatPatient(id);
     nask(ambulanceBlocked(id, location));
     ReturnToHospital(id).
```

Here are how the different steps have been taken into account:

(i) identify a background drawing for rendering graphically the application

→ *the image* `city_mpa.png` *has been selected for the background drawing and has been associated to the background of the scene through the* `background` *declaration of* `ngScene`

(ii) identify objects to be drawn and associate them to images, possibly varying according to the object state

→ *as examples, the objects* `ambulance` *and* `trafficLight` *have been identified (among others) and are associated to the images* `ambulance_idle.png`,...,`traffic_light_green.png`

(iii) associate a widget to each object and use attributes to display the image according to the object state

→ *these two objects are associated with the widgets* `ambulance` *and* `trafficLight`

(iv) identify points in the program in which objects have to move or attributes have to be changed and introduce the corresponding `move` *and* `att` *primitives at these points*

→ *to illustrate this point, a* `move_to` *primitive has been inserted in the code of the* `Ambulance` *procedure.*

7.4 Advanced Topics

In this section, we highlight advanced features of the Multi-Bach language that support dynamic and reactive coordination. Using emergency management as a case study, we show how processes can be treated as active data, how rule-based inference enables automatic reactions, and how self-organization allows decentralized system behaviour. These mechanisms illustrate the language's potential in handling complex, real-world socio-technical scenarios.

7.4.1 Processes as Active Data

In the Multi-Bach coordination language, procedures can be treated as active data, allowing processes to be manipulated in the same way as passive information. This capability is particularly useful for modelling dynamic and reactive behaviours in socio-technical systems. For instance, we can define a procedure `HandleEmergency` that reacts to a reported emergency and updates its status accordingly:

```
1 proc HandleEmergency =
2       ask(reportEmergency(manel, emergency, location_45));
3       tell(status(manel, emergency, treated)).
```

Thanks to the concept of process as active data, proposed in Subsection 4.2.3, we can dynamically create or manipulate processes as active elements in the shared space: using the `tellp(HandleEmergency)` primitive, a new thread is created to manage the emergency of Manel. This thread runs at the same time as other activities in the system. The presence of such a thread can be checked using `askp(HandleEmergency)`, and it can be terminated if necessary using `getp(HandleEmergency)`. In addition, the `naskp(HandleEmergency)`

primitive allows the system to confirm that no emergency processing threads are currently active.

This mechanism illustrates how procedures can be dynamically launched, observed and controlled at runtime, offering a useful abstraction for capturing reactive and autonomous behaviour. It extends the coordination language to treat computation itself as shareable, observable data, bridging the gap between process execution and coordination logic in socio-technical contexts.

7.4.2 Rule-Based Inference

In Multi-Bach, rules make it possible to specify how the system should react automatically to certain situations, something particularly useful in scenarios like emergency response. Imagine a system where the appearance of an emergency si-term on the store should immediately trigger a set of actions: dispatching medical teams, alerting authorities, and updating the system status. Instead of manually coordinating all these effects, we can define a rule like:

$$+emergency \longrightarrow +dispatch, +alert, +status(updated)$$

to express that as soon as an emergency is declared, the corresponding consequences should follow. This style of reactive programming, supported by Multi-Bach's rule mechanism, allows us to declaratively capture the implicit consequences of explicit data. In a general form, a rule can take the following of the structure:

$$+t, -u \longrightarrow +v, -w$$

which means: if term `t` is present and `u` is absent, then `v` should be added and `w` removed. Rules can refer not only to data (si-terms) but also to behaviours (procedures), enabling sophisticated cause and effect chains. This means, for example, that the presence of a given term could trigger the activation of a new procedure, or the absence of one could cancel an ongoing action. Such dynamic rule-based reactivity brings a powerful layer of control, allowing systems to autonomously adjust their behaviour in real time based on changing conditions. They are declared with a name and can be activated, queried, or removed dynamically using commands like `tellr`, `askr`, `naskr`, and `getr`.

By supporting this clean separation between what needs to be detected and what should happen in response, rules bring flexibility to the coordination language. Whether for managing emergency protocols, solving puzzles like Rush Hour, or modelling social interactions, they offer a concise way to encode reactions to evolving conditions in the system.

7.4.3 Self-Organisation in Emergency Management Systems

In many real-world situations, like emergency management, it is crucial for systems to react quickly and adapt on their own, without relying on a central controller.

This is where self-organization becomes powerful: each part of the system can detect what's happening around it and respond accordingly. The example below shows how this kind of decentralized reaction can be modelled using rules that coordinate agents through their shared environment. For instance, a rule like:

```
1 rule emergency_trigger = +alarm_on(zone) —> +evacuate(zone),
2                          +alert_rescue_team(zone), –alarm_on(zone).
```

This rule ensures that as soon as an alarm is detected in a given zone, the system automatically triggers evacuation and alerts the rescue team. Each zone acts as an independent space, enabling agents to respond to local conditions without the need for global synchronization. This decentralized, rule-driven behaviour exemplifies self-organization: the system reconfigures itself in response to environmental changes using only locally available information, while maintaining overall coherence through logically harmonized reactions. In this way, coordination emerges from the distributed interactions of components, enhancing the system's resilience and responsiveness in urgent, dynamic situations.

7.5 Summary

Assuming a description of the application to be developed, the methodology we have proposed can be summarized as consisting of the following steps:

(i) identify key data and represent it as tokens or si-terms (see Subsection 7.2.1)
(ii) identify primitive behaviours and translate them in terms of the $tell$, ask, $nask$, and get primitives (see Subsection 7.2.1)
(iii) associate complex behaviours to procedures (see Subsection 7.2.2)
(iv) define these procedures by using simple behaviours or recursively identifying sub-procedures associated with simpler but yet complex behaviours (see Subsection 7.2.2)
(v) define systems under study as the parallel composition of procedure calls (see Subsection 7.3)
(vi) perform simulations by introducing widgets and graphical primitives at crucial points (see Subsection 7.3)

It is worth observing that the simplicity of the code obtained and the high abstraction provided by the Multi-Bach language allows to easily developed programs through refinements. Indeed, as synchronization is achieved through information being available or not, it is very easy to progressively refine procedures or to add new components.

7.6 Closing Remarks

This chapter has aimed at introducing a methodology to guide the developer transform a description of an application expressed in natural language to a program written in the coordination language Multi-Bach with simulation and property verification being performed in the Anemone workbench. Our main goal has been to identify key steps in this process.

In the next chapter, we shall explore several case studies developed by Master students at the University of Namur in the context of the course INFO M451 "Conception d'applications réactives", which aim at modelling real world IoT projects they developed in the course INFO M453 "Projet en informatique ambiante et mobile".

CHAPTER 8

CASE STUDIES

As a complement to the previous chapter, this chapter aims to further evidence the interest of Multi-Bach *and* Anemone *to develop models of practical applications. We shall subsequently tackle IoT software, for which reliability and security are crucial properties to obtain. More concretely, three innovative projects are tackled in the field of ambient and mobile computing. They have been carried out as part of master courses at the Faculty of Computer Science at the University of Namur. All of them aim to illustrate the practical application of IoT technologies in order to improve users' quality of life by offering intelligent, personalized solutions tailored to a variety of needs.*

In our coordination context, the objective has been to test how easily Multi-Bach *and* Anemone *can be used to develop models on which the students reason before developing concrete applications. Lessons learned by these experiences are reported after the presentation of each project. They will be used to develop new versions of* Multi-Bach *and* Anemone.

8.1 Opening

The landscape of software engineering has seen considerable advancements in recent years, particularly in the realm of data-based coordination languages [27]. While significant research efforts have been dedicated to refining the theory and implementation of these languages, there remains a notable gap in the development of programming environments tailored to analysing and reasoning about programs written in such languages. This oversight limits our ability to fully harness the potential of these languages in practical applications.

In this chapter, we embark on a journey to explore the practical application of IoT modelling within the context of software reliability engineering. Our approach centres on utilizing a workbench Anemone, equipped with the Multi-Bach language. These tools provide a comprehensive framework for describing concurrent systems, animating them, and reasoning about their behaviour a crucial aspect often overlooked in other programming environments. Through the lens of three innovative projects, we delve into the practical implications of IoT modelling facilitated by Multi-Bach and Anemone. These projects span diverse domains, including the simulation of dynamic sports competitions, the optimization of home automation systems for energy efficiency, and the creation of immersive interactive role-playing games. Each project serves as a unique case study, offering valuable insights into the challenges, solutions, and lessons learned in the realm of software reliability engineering within the Internet of Things (IoT) landscape.

8.2 Smart RPG: Smart Role-Playing Game

The creation of an intelligent role-playing game, "Smart RPG," is designed to help a very busy person with a passion for role-playing games. A group of three students, named Marie Carbonnelle, Simon Polet and David Tang, created the game to address this need. Due to time constraints, the user finds it difficult to engage in prolonged gaming sessions. The goal is to make the gaming experience more efficient in order to reduce the time investment. To achieve this, a number of features are implemented, including the facilitation of role-playing environments with lights, sounds, and projected interactive maps. In addition, a connected dice is introduced to adjust the atmosphere based on critical dice rolls, and a connected treasure chest is incorporated, opening with a motor when discovered.

In the revised flow of play, players take turns rolling the connected dice, influencing the lights in the room, before moving horizontally or vertically across the board. To find the treasure, they navigate to its location, and the treasure chest opens on arrival. Players also have the option of changing the *decor* of the environment at any time during the game.

Key Data Identification and Translation into Si-Terms

To introduce the "Smart RPG" intelligent role-playing game and its underlying data sets, the students begin with the following components: *Positions*, *Decors*, *NumeroDe*, *ColoursLeds*, *EtatCoffre*, and *SensJoueur*, which are shown in Figure 8.1.

```
1 eset Positions = { haut1, haut2, haut3, haut4, haut5,
2                    mihaut1, mihaut2, mihaut3, mihaut4, mihaut5,
3                    mibas1, mibas2, mibas3, mibas4, mibas5,
4                    bas1, bas2, bas3, bas4, bas5 }.
5     Decors    = { foret, montagne, volcan, desert }.
6     NumeroDe  = { de1, de2, de3, de4, de5, de6 }.
7
```

```
8       CouleursLeds = { rouge, blanc, jaune }.
9       EtatCoffre   = { ouvert, ferme }.
10      SensJoueur   = { droit, reverse }.
```

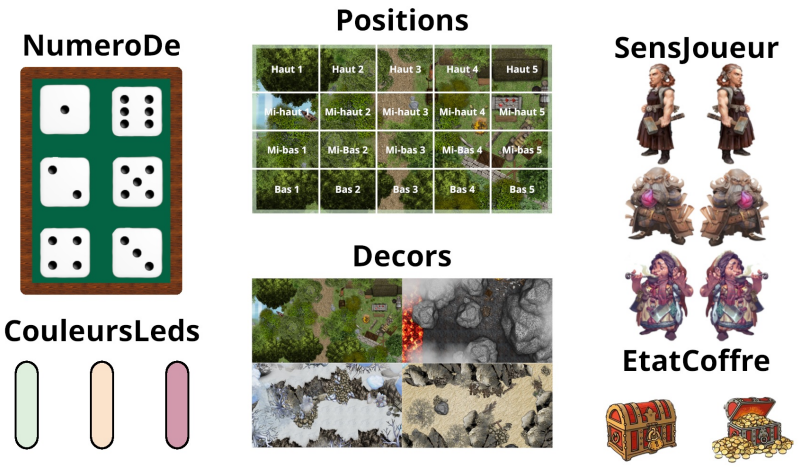


Figure 8.1: Representation of the different sets defined Smart RPG game

Within the game environment, which is represented as a game board, pawns are positioned on distinct squares. Leveraging the Multi-Bach equations, students began by defining the x and y coordinates for each square in the Positions dataset, as shown below:

```
1  eqn x_coord(haut1)    = 150. y_coord(haut1)    = 50.
2      x_coord(haut2)    = 514. y_coord(haut2)    = 50.
3      x_coord(mihaut1)  = 150. y_coord(mihaut1)  = 295.
4      x_coord(mihaut2)  = 514. y_coord(mihaut2)  = 295.
5      x_coord(mibas1)   = 150. y_coord(mibas1)   = 540.
6      x_coord(mibas2)   = 514. y_coord(mibas2)   = 540.
7      x_coord(bas1)     = 150. y_coord(bas1)     = 785.
8      x_coord(bas2)     = 514. y_coord(bas2)     = 785.
9      ...
```

The students define the successors and predecessors for each square, essential for enabling movement across the board. Horizontal (*succ* and *pred*) and vertical (*succv* and *predv*) successors and predecessors are specified to allow both horizontal and vertical movement, as illustrated in Figure 8.2.

Before exploring the game processes, it is important to describe the configuration of the Bach system widgets. Specifically, for this application, the students define the widgets *decors*, *pion1*, *pion2*, *pion3*, *coffre*, *de*, and *leds*. Using attribute definition capabilities, they condition the display of each widget to show different

images based on specific attributes. Moreover, the *Init* function is used to establish default image configurations and their respective positions.

```
succ(haut1) = haut2.  succ(haut2) = haut3.  succ(haut3) = haut4.
succ(haut4) = haut5.  succ(haut5) = haut5.

pred(haut1) = haut1.  pred(haut2) = haut1.  pred(haut3) = haut2.
pred(haut4) = haut3.  pred(haut5) = haut4.

succv(haut1) = mihaut1. succv(haut2) = mihaut2. succv(haut3) = mihaut3.
succv(haut4) = mihaut4. succv(haut5) = mihaut5.

predv(haut1) = haut1.  predv(haut2) = haut2. predv(haut3) = haut3.
predv(haut4) = haut4.  predv(haut5) = haut5.

succ(mihaut1)= mihaut2. succ(mihaut2) = mihaut3. succ(mihaut3)= mihaut4.
succ(mihaut4)= mihaut5. succ(mihaut5) = mihaut5.
...
```

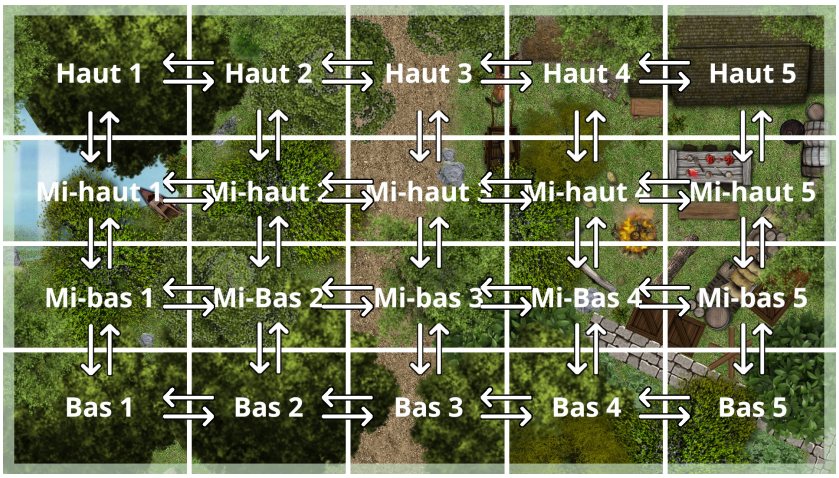


Figure 8.2: Game board illustration, and the possible movements of the checkers

Key Processes Identification and Translation into Multi-Bach Procedures

This section details the conversion of crucial processes into procedures vital for orchestrating the game simulation.

Init The *Init* process initiates all necessary components before the beginning of the role-playing game. It initiates *draw_scene* to set up the background, load required images with *loadImage*, and create widgets. Subsequently, *place_at* and

show are employed to position initial elements like the scene, *LEDs*, *die*, *chest*, and *counters* on the map. Following this *setup*, *Init* utilizes the *tell* function to mark occupied places indicated by players and designate player 1 to start. The process concludes by invoking the Main process to execute a typical turn.

```
proc  Init = draw_scene(smartrpgScene);
      place_at(decors, smartrpgScene,0,0);
      show(decors, smartrpgScene);

      place_at(led, smartrpgScene,0,0);
      show(led, smartrpgScene);

      place_at(de, smartrpgScene, 1675, 900);

      place_at(coffre, smartrpgScene, x_coord(mibas3), y_coord(mibas3));
      show(coffre, smartrpgScene);

      place_at(pion1, smartrpgScene, x_coord(haut1), y_coord(haut1));
      show(pion1, smartrpgScene);

      place_at(pion2, smartrpgScene, x_coord(bas1), y_coord(bas1));
      show(pion2, smartrpgScene);

      place_at(pion3, smartrpgScene, x_coord(haut5), y_coord(haut5));
      show(pion3, smartrpgScene);

      tell(haut1); tell(bas1); tell(haut5); tell(Pion1);
```

Main The Main process handles the execution of a turn, coordinating various sub-processes for each pawn's turn: *pion1*, *pion2*, *pion3*, or the scenery change process. After verifying the current player's turn, Main initiates a dice roll through the *RollDice* process. The pawn then chooses one of five movement options: *right*, *left*, *down*, *up*, or staying stationary. Using *nask*, the program checks the occupancy of the chosen square before signalling the pawn's move with *tell*. The *move_to* function subsequently relocates the pawn to the new square and frees the previously occupied one. The turn concludes with a choice: either opening the chest via *OpenCoffre* or signalling the next player's turn, thereby preparing the pawn for the following turn.

```
proc Main(p1: Positions, p2: Positions, p3: Positions) =
     ( (Pion1(p1) ||  Pion2(p2)) ||  Pion3(p3) ) ||  ChangeDecor.
```

ChangeDecor This process facilitates changing the game background at any point by altering the *decor* attribute in the *decors* widget. The choice operator allows selecting between mountain, forest, desert, or volcano scenery.

```
proc ChangeDecor = ((att(decor, decors, smartrpgScene, foret))    +
                    (att(decor, decors, smartrpgScene, montagne)) +
                    (att(decor, decors, smartrpgScene, volcan))   +
                    (att(decor, decors, smartrpgScene, desert)));
                    ChangeDecor.
```

RollDice Enabling players to roll the die, the RollDice process generates a single dice roll value through choice operators. It updates the number attribute in the widget and displays the corresponding image. Additionally, it adjusts the LEDs' color based on the roll outcome: red for 1, yellow for 6, and white for other numbers.

```
proc RollDice =
(
   (((att(numero, de, smartrpgScene, de1)); show(de, smartrpgScene);
     (att(couleur, led, smartrpgScene, rouge)))     +
    ((att(numero, de, smartrpgScene, de6)); show(de, smartrpgScene);
     (att(couleur, led, smartrpgScene, jaune)))
    )
+
   (((att(numero, de, smartrpgScene, de2)); show(de, smartrpgScene);
     (att(couleur, led, smartrpgScene, blanc)))     +
     ((att(numero, de, smartrpgScene, de3)); show(de, smartrpgScene) ;
     (att(couleur, led, smartrpgScene, blanc)))
   )
+
   (((att(numero, de, smartrpgScene, de4)); show(de, smartrpgScene) ;
     (att(couleur, led, smartrpgScene, blanc)))     +
    ((att(numero, de, smartrpgScene, de5)); show(de, smartrpgScene) ;
     (att(couleur, led, smartrpgScene, blanc)))
   )
); hide(de, smartrpgScene).
```

OpenCoffre Executed when a player lands on the chest square, OpenCoffre checks the chest's occupancy status and updates the chest widget's status attribute to display an open chest image, effectively concluding the game.

```
proc OpenCoffre = ask(mibas3);
                  (att(statut, coffre, smartrpgScene, ouvert));
                  tell(open).
```

Key Animations Identification

The implementation of the game in the Anemone workbench was characterised by several strong points:

(i) Firstly, the integration of the formulae verification has created a fluid mechanism for determining the end of the game, stopping the simulation when a player opens the treasure chest. This functionality, achieved using the *tell(open)* function and the *Reach(#open=1)* formula, ensures a consistent and immersive game experience.

(ii) In addition, the strategic implementation choices were decisive, in particular by allowing the pawns to move freely rather than following a predetermined path, and by allowing players to remain motionless during their turns to avoid stalemate situations. The management of probabilities for dice throws has also been carefully thought out to avoid any bias and provide a balanced experience.

(iii) Finally, the resolution of difficulties encountered, such as the rearrangement of widgets to maintain visual consistency, demonstrated a methodical and creative approach to overcoming obstacles. These strengths contributed to the successful implementation of the game, offering players an immersive and dynamic experience.

Reflection on Anemone in SmartRPG Scenarios

In reflecting on the application of Anemone in IoT scenarios, particularly in the context of the described SmartRPG project, several insights emerge. Anemone's versatility and capability to model complex interactions between actors in a shared environment proved invaluable in simulating the dynamics of the RPG gameplay.

By leveraging Anemone's features, such as defining data sets, orchestrating processes, and integrating formulae verification, the project team successfully streamlined RPG sessions, reducing the time investment required for setup and gameplay. Furthermore, Anemone's flexibility facilitated the implementation of additional features like the connected die and treasure chest, enhancing the immersive experience for players. However, challenges such as widget organization and implementation choices required careful consideration, highlighting areas for potential refinement and improvement in future iterations. Overall, the experience with Anemone underscored its utility in modelling and simulating IoT scenarios, offering valuable insights into actor coordination and system behaviour that are applicable across various industrial contexts.

8.3 Smart Garden

The primary objective of the "Smart Garden" project is to develop connected watering cans and plants managed through an API and a mobile application. Users can retrieve information about the plants and watering cans via the API, which also

oversees the filling of the watering cans by communicating with the base where they are situated. Our emphasis was on facilitating the exchange of information among the API, the plants, the watering cans, and a weather API, which provides real-time data on the plants' environmental conditions.

In this modelling, the students, Lucas Berg and Johan Rochet, assume the plant to be an outdoor species. Its movement simulates user intervention to adjust its position for optimal sunlight exposure. The watering can simulation involves four stages: placement on its base, PLC-controlled filling, user watering, and return to the initial position. Regarding weather simulation, day-night cycles are depicted with representations of the sun or moon, and plants' water requirements are randomized daily. Cloud presence signifies insufficient sunlight for a plant, streamlining the IoT system compared to more intricate modelling approaches.

Key Data Identification and Translation into Si-Terms

To achieve smoother animations, in this project, the students Lucas Berg and Johan Rochet created multiple widgets representing the scene, allowing for more flexible movement. Each widget represents an element of the animation, with attributes that modify its state and displayed image. For instance, a plant widget has an attribute "watered" that determines if it needs water, changing the displayed image accordingly. Other widgets, like a phone, use several attributes to infer the displayed image based on their combined values.

```
1  eset Environment = {day, night}.
2       PlantState = {isWatered, notWatered}.
3       WateringCanState = {empty, full, watering}.
4       ...
5
6  scene ppScene = {  size = (1280, 720). layers = { top }.
7       background  =  loadImage(Images/white_background.png).
8       widget sun  = { display = {sun_img}    init = { wdl = top. }}
9       widget moon = { display = {moon_img}   init = { wdl = top. }}
10      widget cloud= { display = {cloud_img}  init = { wdl = top. }}
11      widget wateringCan =
12      { attributes = {state in WateringCanState.}
13        display    = {state = empty -> wateringCan_empty_img.
14                      state = full -> wateringCan_full_img.
15                      state = watering -> wateringCan_watering_img.}
16              init = { wdl = top.  state = empty.}
17      }
18      widget plant =
19      { attributes = {state in PlantState.}
20        display    = {state = isWatered -> plant_img.
21                      state = notWatered -> dry_plant_img.}
22              init = { wdl = top.  state = notWatered. }
23      }
24  ...}
```

Key Processes Identification and Translation into Multi-Bach Procedures

Processes within the application are built around the shared space and the *tell, ask, nask*, and *get* primitives. This allows processes to interact, sometimes preventing one process from executing while allowing another. The "Smart Garden" project leverages the parallel execution property of these processes, or agents. The animation comprises three main agents that run in parallel:

- *Day*: Manages transitions between day and night, and moves the moon and sun, shown in 8.3.
- *Cloud*: Moves a cloud across the scene, shown 8.4.
- *PlantWateringManagement*: Manages watering a plant or moving it.

These agents run concurrently throughout the program using the parallel composition operator. They employ recursion to run indefinitely and use equations to determine image positioning in the animation. Each position is recorded as coordinates in the code, with some locations automatically linked to the next position for the image's subsequent move. This link, expressed by the *next(x)* operation, facilitates easy setup of movement cycles within the animation.

```
1  proc Init = draw_scene(ppScene);
2              (Day || Cloud || PlantWateringManagement).
3
4       Day = DayLight; Night; UpdatePlantStatus; Day.
5       DayLight = show(sun,ppScene); MoveSun(sunLeftOut).
6       Night = show(moon,ppScene); Movemoon(moonLeftOut).
7       UpdatePlantStatus =
8       (     nask(isWatered) +
9           (   (get(isWatered); att(state, plant, ppScene, notWatered);
10              att(plantState, smartphone, ppScene, notWatered))
11           +
12             ask(isWatered)
13           )
14      ).
15      Cloud = ask(cloudOut); MoveCloud(cloudOut); Cloud.
16
17      PlantWateringManagement = (WateringCan + Plant);
18                                PlantWateringManagement.
```

Key Animations Identification

The *PlantWateringManagement* agent is the most intricate element implemented in the application. This agent can either launch a plant-moving agent (similar to Day and Cloud) or, more importantly, manage the watering process. Notably, these two agents cannot run in parallel because the watering can must move towards the plant to water it, which would be complicated if the plant also moved simultaneously. Let's delve into the *WateringCan* agent in more detail:

```
1 proc WateringCan = nask(isWatered);
2                    get(empty);
3                    MoveWateringCan(wateringCanNotOnBase, empty).
```

Figure 8.3: Representation of the situation in the Smart Garden, in Daylight

Figure 8.4: Representation of the situation in the Smart Garden, in Daylight

Initially, the *nask* primitive is used to check the plant state (whether it needs water) to determine if watering is required. If the watering can is empty, it performs a series of movements and actions to refill from the valve. Once filled, the can moves

towards the plant, waters it, and then returns to its initial position. What stands out is that all these actions are handled by a single agent, *MoveWateringCan*, which uses the non-deterministic choice operator + combined with conditional instructions to create a sequence of possible movements based on the data in the shared space. The limitation here is that conditional instructions cannot access shared space data directly within the condition, as they are not executable instructions. To address this, parametrized agents are used, allowing parameters to be directly accessed and checked within the program conditions. The parameters are updated in the recursive call after completing an action, ensuring smooth and logical execution of the watering sequence.

Reflection on Anemone in Smart Garden Scenarios

In the Smart Garden simulation, agents like *PlantWateringManagement* demonstrate the effective management of tasks through parallel execution. This agent can either move the plant or manage its watering, ensuring actions are executed in a synchronized manner to prevent conflicts, such as attempting to water a moving plant. The use of recursive processes and deterministic choice operators allows for dynamic and adaptive responses to changing conditions, mimicking the real-time decision-making required in actual IoT applications.

However, while Anemone excels in creating detailed and flexible simulations, transitioning these models to practical, real-world IoT implementations can pose challenges. Ensuring real-time data synchronization and integration with physical devices requires additional layers of complexity not fully addressed within the simulation environment. These challenges highlight the need for further development and adaptation when moving from theoretical models to practical applications.

8.4 Anemove: Women's Sports Competition

In gymnastics, the students, Claudia Carbonara, Ludovic Evrard and Matthys Gaillard, modelled the competition as follows:

In team competitions, gymnasts follow a structured rotation through the different apparatus. A team typically consists of 4 to 6 gymnasts, with different formats determining how many compete per event and how many scores count toward the team total (e.g., 4-4-3 or 5-4-3). Teams follow a set event rotation: Women's Artistic Gymnastics (WAG) includes Vault, Uneven Bars, Balance Beam, and Floor Exercise, while Men's Artistic Gymnastics (MAG) features six events, starting with Floor Exercise and ending with Horizontal Bar. Each team moves together between apparatuses according to a schedule, ensuring fairness. Major competitions, like the Olympics, use a qualification round to determine finalists, with scores from different events combined to rank teams. The team final follows a stricter format, where every score counts, making consistency crucial for victory. For this model, the students chose to simulate 8 teams (Belgium, France, Italia, Holland, Germany, Romania, Austria and Ireland).

Key Data Identification and Translation into Si-Terms

Localizations Defining specific locations within the simulated environment, facilitating the placement of teams "*Place_teams*", competitors "*Place_Competitor*", flags "*Place_flags*", and scores "*Place_scores*".

```
eset Localisations = {a1,a2,a3,a4,a5,out,
                      out_belgium, out_france,
                      out_italia, out_holland,
                      out_germany, out_romania,
                      out_austria, out_ireland}.
```

Teams and Competitors Separate representations for teams *Teams* and individual competitors *Competitors*, with additional structures mapping teams to specific localizations, with *TeamsCompetitors*.

```
eset Teams = {at, be, de, fr, ie, it, nl, ro}.
     Competitors = {one, two, three, four}.
     TeamsCompetitors = {atone, attwo, atthree, atfour,
                         beone, betwo, bethree, befour,
                         deone, detwo, dethree, defour,
                         frone, frtwo, frthree, frfour,
                         ieone, ietwo, iethree, iefour,
                         itone, ittwo, itthree, itfour,
                         nlone, nltwo, nlthree, nlfour,
                         roone, rotwo, rothree, rofour}.
```

Scores and CompetitorsScore Managing the scoring system by associating scores with individual competitors.

```
eset Scores = {s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,
               s11,s12,s13,s14,s15,s16,s17,s18,s19,s20}.

     CompetitorsScore = {
     ones0, ones1, ones2, ones3, ones4, ones5, ones6, ones7, ones8,
     ones9, ones10, ones11, ones12, ones13, ones14, ones15, ones16,
     ones17, ones18, ones19, ones20,

     twos0, twos1, twos2, twos3, twos4, twos5, twos6, twos7, twos8,
     twos9, twos10, twos11, twos12, twos13, twos14, twos15, twos16,
     twos17, twos18, twos19, twos20,

     threes0, threes1, threes2, threes3, threes4, threes5, threes6,
     threes7, threes8, threes9, threes10, threes11, threes12, threes13,
     threes14, threes15, threes16, threes17, threes18, threes19,
          threes20,
```

```
16
17      fours0, fours1, fours2, fours3, fours4, fours5, fours6, fours7,
18      fours8, fours9, fours10, fours11, fours12, fours13, fours14,
            fours15, fours16, fours17, fours18, fours19, fours20 }.
```

Matches *Matchs* defining individual matches (e.g., m1, m2, m3, m4) that contribute to the overall competition.

```
1 eset  Matchs = {m1, m2, m3, m4}.
```

Directions Establishing possible movement directions within the simulation environment.

```
1 eset Directions = {left,right}.
```

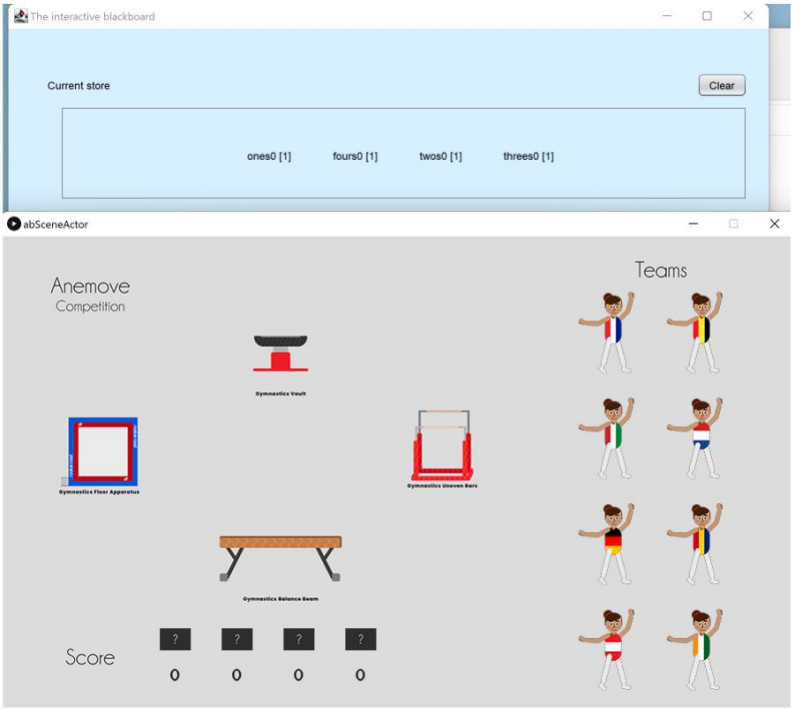


Figure 8.5: Initial representation of a running competition in Anemove

These data structures offer a comprehensive representation of the simulated environment and serve as the backbone for implementing various procedures and rules governing the sports simulation within the program.

Key Processes Identification and Translation into Multi-Bach Procedures

This section of the program outlines the translation of key processes into procedures essential for orchestrating the sports competition simulation. These procedures encompass tasks such as initialization, competitor movement, scoring, and competition management, providing a comprehensive framework for executing the simulation. Let's delve into the key procedures that drive the functionality of the program:

Initialization The *Init* and *Start* procedures *kickstart* the program by initializing the competition's initial context, with *draw_scene(stage)*. This involves setting up the scene, placing flags, scores, and teams.

```
1 proc  Init  = draw_scene(stage); Place_flags; Place_scores; Place_teams.
2       Start = Place_Competitor; Start_Competition.
```

During the Competition Once the competition is under way, teams move to successive positions, guided by strategic tactics.

```
1  proc Place_Competitor(c: Competitor) =
2     ( nask(nation);
3       tell(nation);
4       tell(team_competitor(nation, c));
5       tell(team_localisation(nation, get_position_by_competitor(c)));
6       Set_Flag(c, nation);
7       place_at( nation, stage,
8                 x_coord(get_position_by_competitor(c)),
9                 y_coord(get_position_by_competitor(c)));
10      Orientate_Team(get_position_by_competitor(c),nation)
11    ).
```

The *Place_Competitor* procedure is employed to position the initial competitors and their flags, while the *Start_Competition* procedure begins the competition by overseeing scoring judgements between competitors. In this context, each block starts with the instructions of Multi-Bach *nask(nation)*, *tell(nation)*, specifying the nation associated with the current block. Several tools are employed within these blocks to manage competition elements effectively:

- *nask(nation)*: This function verifies if the token is not already present on the "table". If it's not initially in the table, it's chosen. If the initial attempt fails, another block is randomly executed.
- *tell(nation)*: This function informs the procedure about the current nation in the competition, adding the nation to the table, ensuring it cannot be chosen again.
- *tell(team_competitor(nation, c))*: It indicates that competitor 'c' belongs to the team of the current nation.

- *tell(team_localisation(nation, position))*: This function informs about the position of the current nation's team based on the position of the competitor.
- *Set_Flag(c, nation)*: Sets a flag associated with the nation for competitor 'c'.
- *place_at(nation_team, stage, x, y)*: Places the current nation's team on the stage at the specified coordinates.
- *Orientate_Team(position, nation)*: Orients the current nation's team according to the position of the competitor.

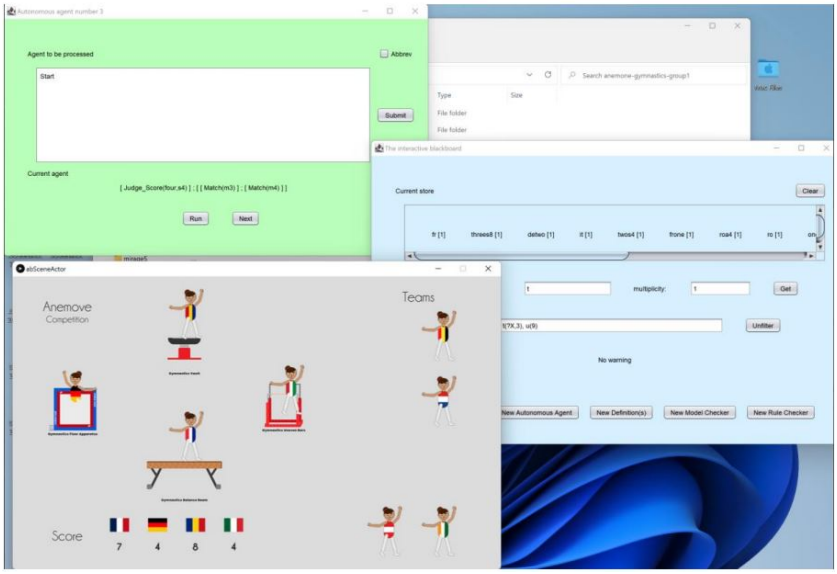


Figure 8.6: Representation of a running competition in Anemove during the execution phase, where Romane's team is leading by 8 points

Competitor Movement Competitor movement is facilitated by procedures such as *ClockWiseCompetitorMove* and *ClockWiseCompetitorMove_L*. These procedures manage the clockwise movement of competitors within the competition context, ensuring smooth transitions between positions. To manage *ClockWiseCompetitorMove*, the following tools are used:

- *ask(team_localisation(nation, position))*: asks the current team position of the current nation.
- *Orientate_Team(position, nation)*: orients the current nation's team based on the position obtained.
- *ClockWiseCompetitorMove_L(nation, position)*: called to handle the actual clockwise movement of the competitors for the specific nation. To manage this procedure, where the specified *nation(t)* or *team(t)* in the specified *location(l)*, the students use the followings primitives:

- *get(team_localisation(t, l))*: gets the current location of the team associated with the specified nation in the specified location.
- *move_to(team, stage, x, y)*: it moves the team to the new location specified by the coordinates (x, y).
- *tell(team_localisation(t, next(l)))*: informs about the new location of the team.

End of Competition After four rounds, the competition concludes, with the winning team determined by the highest score. The *Re_Start* procedure offers the option to either conclude or restart the competition for subsequent rounds. Each procedure is meticulously structured to handle specific tasks, utilizing conditional blocks and various tools such as functions and operators to ensure seamless execution. These procedures collectively form the backbone of the program, driving the simulation forward and providing users with a dynamic sports competition experience.

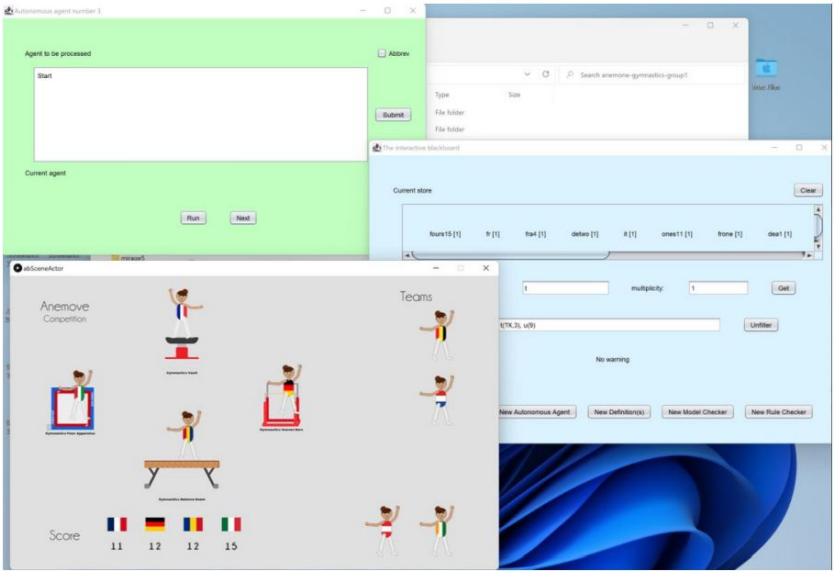


Figure 8.7: Representation of a running competition in Anemove at the end of the event, where France won with a 15 points lead

Key Animations Identification

In this process, to configure a new interactive agent within the Anemone board, the students begin by initializing the *Init* procedure. This action loads the competition setup into the current agent, where the user can sequentially activate procedures until an "empty agent" state is reached. This progression is essential for setting up the competition environment, including the placement of competitors, designing

flags, and setting initial scores (*default score = s0*). Upon completion of these initial setups, the students proceed by creating a “New Autonomous Agent” on the same board. Here, the students copy and activate the *Start* procedure to officially begin the competition. This competition is structured into four matches, with the first match primarily focused on score updates. The subsequent matches facilitate a full rotation of athletes enhancing the dynamism of the competition and generate updated scores continuously.

Each team’s participation chance in the competition is equitably distributed, with a selection probability of approximately 12.5%. The procedure allows for a fair representation and participation of teams from various nations like Belgium, France, Italy, and the Netherlands. Throughout the competition, the tokens on the table provide valuable insights:

- fr: France’s presence in the competition.
- frone: France’s first competitor.
- detwo: Germany’s second competitor.
- it: Italy’s participation.
- threes8: The third competitor’s score of 8 points.
- ro: Romania’s involvement in the competition.
- roa4: Romania’s competitor stationed at a4 in the Gymnastics Vault.
- And additional tokens providing further insights.

Following the conclusion of all matches, the *Re_Start* procedure is utilized to reset the competition environment. This involves clearing the existing tokens, repositioning teams to their original locations, and resetting the flags and scores to their initial state (s0). This setup prepares the stage for a fresh start of the competition, ensuring that all conditions are reset appropriately for a new round.

Reflection on Anemone in Anemove Scenarios

To conclude, the Anemove project offers, according to the students, a valuable perspective on how practical research can enrich academic learning, underlining the importance of practical experimentation in higher education. As a result, Anemove contributes to the discussion on the need to integrate rigorous research methodologies into academic curricula to prepare students for future professional applications, even in the absence of direct collaboration with industry.

8.5 Educational Impact

The three projects offer significant learning opportunities in various domains, with a focus on industrial applicability:

- “SmartRPG" (see Section 8.2) explore additional game mechanics, like using keys to unlock treasure chests, enriches players’ experiences and fosters strategic thinking and decision-making skills transferable to industrial contexts.

- "Smart Garden" (see Section 8.3) enables students to grasp time management and automation of actions in a simulation environment, reinforcing their understanding of system dynamics and programming logic essential skills for developing industrial IoT solutions. Moreover, modelling communication between IoT devices provides hands-on experience with network protocols and data exchange in interconnected systems, crucial for designing and implementing industrial solutions.
- "Anemove" (see Section 8.4) helps students to grasp structured scheduling, resource management, and synchronized task execution. Modelling team rotations enhances their skills in process optimization and multi-agent system design, key concepts in automation and IoT applications.

These projects collectively offer a diverse learning experience covering technical and transferable skills, underlining their relevance to industrial settings.

8.6 Lessons Learned

The workbench of Anemone in IoT scenarios, particularly in the SmartRPG and Smart Garden projects, highlighted its versatility in modelling complex interactions and processes. Anemone proved invaluable in simulating RPG game-play dynamics and managing parallel execution in smart environments, demonstrating its potential for actor coordination and system behaviour modelling. Features like dataset definition and process orchestration enhanced automation, while flexibility allowed for immersive additions such as connected devices.

However, challenges arose in widget organization, implementation choices, and real-time data synchronization with physical devices, emphasizing areas for refinement. Despite these hurdles, Anemone provided valuable insights into IoT system development, underscoring the importance of bridging simulation capabilities with real-world integration.

8.7 Closing Remarks

In conclusion, the "SmartRPG", "Smart Garden" and "Anemove" projects explored different facets of modelling and simulation in IoT environments. Despite encountered challenges, particularly in time management, widget organization, and programming complexity, these projects demonstrated the usefulness of the MultiBach language and the Anemone workbench in modelling concrete applications.

Part V

Postface

CHAPTER 9

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Throughout this thesis, we have explored how coordination languages and models can be extended not only to encode the behaviour of socio-technical systems, but also to reason about them. The reasoning developed here is not meant to serve as a generic inference engine or artificial intelligence system. Instead, it aims to provide a clearer understanding of programs through visual tools and accessible verification logic formula.

In this chapter, we begin by summarizing and concluding the main contributions of this thesis in Section 9.1. Then, in Section 9.2, we present our concluding remarks. Finally, in Section 9.3, we outline possible directions for future work.

9.1 Summary of Contributions

In **Part I**, Chapter 1, we introduced distributed systems and coordination models, and showed that Bach is a particularly suitable choice. Compared to other languages, Bach offers key advantages: it remains simple and intuitive like Linda and Gamma, while strengthening term management for distributed and mobile environments. Unlike more complex frameworks such as LINC or Klaim, Bach focuses on straightforward agent coordination, enabling faster development. Its asynchronous communication and data-driven approach makes it flexible. We also emphasized the importance of considering both technical and social aspects when designing distributed systems, as coordination languages help align technical actions with social goals.

In Chapter 2, we explored the concepts and principles of Bach through a concrete case study: Altar, a tablet-based restaurant. We described the scenario where clients use tablets to order, track, and pay for meals, while cooks and waiters coordinate service. We presented Bach core elements: data, primitives, and agents. To improve usability, we discussed the implementation and embedding of Bach in Scala. This case study illustrates how Bach enables transparent coordination within socio-technical systems.

In **Part II**, Chapter 3, we addressed the first main goal of this thesis: extending the Bach coordination language to animate execution scenarios. The resulting language is called Anim-Bach. This extension comes with a dedicated workbench Scan designed to help users better understand and reason about Bach programs. The workbench serves three main purposes: to offer step-by-step execution with visual modifications on the shared space, to connect agents to animations that reflect system behaviour, and to support property checking via the verification of temporal logic formulae and associated trace generation. It is built around two core principles: ease of use (through a standalone, command-line tool with minimal complexity), and a clear link between user input and internal representation. Then, in Chapter 4, we extended the developments introduced in the previous chapter, building upon Anim-Bach with a more advanced version called Multi-Bach. While the overall research direction remains the same, several key improvements have been made at the process algebra level, such as the proposal of generalized choices, active data manipulation, blackboard rule handling, and support for multiple scenes with widgets as main elements. The corresponding workbench, named Anemone, integrates these enhancements and offers new features, including a filter for shared space elements, new windows for agent-driven process creation, and rule management. These refinements aim to improve modelling flexibility and usability. The tool, along with tutorials and examples, is available online, as indicated in reference [13].

As the work progressed, our models and tools evolved to tackle practical issues such as complexity, scalability, and expressiveness. One key innovation was the introduction of the *Guarded List*, in **Part III**, Chapter 5, a construct that improved verification performance and expanded the expressive power of the Multi-Bach language. In Chapter 6, we also proposed a refinement method to ensure that such optimizations can be introduced by safely, preserving the meaning of the original coordination programs and the verification of logic formulae. This chapter has studied *Expressiveness Issues* and, among other results, has established that guarded list preserves the expressiveness of the family of languages derived from Bach, by progressively taking primitives into consideration, while offering a strict increase in expressiveness compared to Bach counterparts.

Beyond the technical advances, this thesis also outlines a practical methodology to guide users in translating informal, often natural-language system descriptions into executable coordination models. This has been addressed in **Part IV**. In Chapter 7, we present the methodology for applying our coordination language, Multi-Bach, and the Anemone workbench to socio-technical systems. Then, in

Chapter 8, we tested this approach through a series of case studies, including IoT projects developed by Master's students. These examples demonstrate not only the flexibility of our tools, but also their ability to support real-world applications where systems are dynamic, interactive, and multi-layered. In short, this work shows that coordination languages, when thoughtfully designed and supported by the right tools, offer a powerful and intuitive way to code socio-technical systems and reason about the resulting programs. The combination of expressive modelling, visual simulation, and formal verification opens up new possibilities for system design that is both rigorous and accessible.

9.2 Closing Remarks

Coming back to the thesis itself, the goal as stated in the future is to demonstrate that coordination languages can be effectively use to code and reason about socio-technical systems. It highlights the growing importance of seamlessly integrating technical and social dimensions in contemporary distributed systems. Our work has made significant contributions to the modelling, understanding, and verification of ensuring from coordination activities.

(i) First, we enriched the Bach coordination language, creating Anim-Bach and Multi-Bach, which extend the language capabilities to encode both distributed and socio-technical systems. Our main contribution in this respect is to introduce these extensions emphasizing the importance of separating interactional and computational aspects, offering tools that are both expressive and intuitive for addressing real-world challenges.

(ii) Second, we developed the workbenches Scan and Anemone. These tools empower users to animate, analyse, and verify system behaviours step by step, making the often abstract nature of coordination more tangible. By enabling reasoning through formulae verification and interactive visualizations, these workbenches also serve as powerful allies in understanding the dynamics of socio-technical systems.

Our findings confirm that coordination languages, when thoughtfully extended, can act as a robust foundation for designing and reasoning about socio-technical systems. Applications across diverse domains from IoT to cybersecurity demonstrate the broad relevance and potential of this approach. In doing so, we hope to have convinced the reader of the two claims stated in the preface:

Claim 1. *Enriching the Bach coordination language with primitives for dealing with processes as active data, for relating data and for animating computations creates a model suitable to code socio-technical systems.*

Claim 2. *A workbench that allows (i) to execute coordination programs step by step, (ii) to animate computations by graphical means and (iii) to check properties by verifying formulae is a good tool to reason about socio-technical systems.*

9.3 Future Work Directions

Looking forward, there are many exciting directions for future research: refining verification techniques, bridging with mainstream programming ecosystems, and exploring new applications in areas like smart cities, collaborative Artificial Intelligence, and adaptive systems.

As socio-technical systems become increasingly central to our digital world, we believe that coordination-based thinking will play a key role in shaping systems that are not just functional, but also responsive to human and organizational needs.

Optimization of Model Checking

The current state-space explosion problem in model checking for coordination languages like Multi-Bach could be further addressed by exploring advanced techniques, such as symbolic model checking or SAT/SMT solvers [19]. These approaches can significantly enhance the scalability of verification, enabling the analysis of larger and more complex socio-technical systems. Applying these techniques to Multi-Bach could help optimize performance and make it suitable for real-time systems in critical domains like healthcare or autonomous vehicles.

Integration with Machine Learning

Combining coordination languages with machine learning models opens new possibilities for adaptive socio-technical systems. For instance, reinforcement learning techniques could be used to optimize coordination policies in Multi-Bach based systems [115]. Such integration could enable systems that learn and evolve dynamically based on social and technical feedback, especially in environments like smart cities or intelligent transportation systems.

Trust and Explainability in Socio-Technical Systems

With the rise of trusted AI and ethical considerations, extending coordination models to incorporate explainability mechanisms is a promising direction. Inspired by works like [105], adding primitives for logging and tracing decisions in Multi-Bach could enhance user trust and accountability. This could be particularly valuable in domains like AI-driven decision-making or collaborative robotics, where understanding the "why" behind decisions is crucial.

Cross-Domain Applications

While Multi-Bach has demonstrated utility in IoT and cybersecurity, its application to emerging areas like blockchain based systems [128] or edge computing [114] could be explored. Extending the Anemone workbench to simulate distributed ledger interactions or edge nodes coordination could provide valuable insights into these fast-evolving fields.

Hybrid Socio-Technical Systems

As socio-technical systems become increasingly hybrid, incorporating both physical and virtual components, coordination languages could be adapted to model cyber-physical systems [73]. Extending Multi-Bach with primitives for physical processes, such as sensors and actuators, would enable its application in robotics, manufacturing, and smart grids.

Formal Verification for Security Protocols

The verification of security protocols using coordination languages, as illustrated by the analysis of the Needham-Schroeder protocol, can be expanded to include more modern protocols [102]. Enhancing Multi-Bach to incorporate primitives for cryptographic operations or adapting existing tools like [100] could bolster its utility in cybersecurity. In this context, we have contributed to the article *B2Scala tool: Integrating Bach in Scala with Security in Mind* [94], which proposes an alternative approach to verifying security protocols using asynchronous communication via a shared space. This work illustrates the interest of integrating a Domain Specific Language (DSL) like Bach in Scala to facilitate program verification while leverage the Scala ecosystem.

Part VI

Appendix

APPENDIX A

BACH TO ALTAR RESTAURANT

The upcoming sections showcases the "Scala" code depicting our intelligent restaurant, named "Altar," which has been modelled using the Bach coordination language. In the initial part, we introduce the data code, followed by the customer code, and subsequently the waiter code, which is then succeeded by the kitchen code. Finally, we present the execution code for our restaurant, following the code related to individuals.

Data Code

```
object BSC_modelling extends App {
    val manel  = Token("Manel")
    val alice  = Token("Alice")
    val emma   = Token("Emma")
    val john   = Token("John")
    val tom    = Token("Tom")

    val t1     = Token("table 1")
    val t2     = Token("table 2")
    val t3     = Token("table 3")
    val t4     = Token("table 4")

    val nigiri = Token("Nigiri")
    val maki   = Token("Maki")
    val temaki = Token("Temaki")
    val beer   = Token("Beer")

case class logged(id:SI_Term) extends SI_Term
case class client_sitting_at_table(id:SI_Term,t:SI_Term) extends SI_Term
case class order(id:SI_Term,table:SI_Term,dish:SI_Term) extends SI_Term
case class t_order(id:SI_Term) extends SI_Term
case class t_check(id:SI_Term) extends SI_Term
case class order_prepared(id:SI_Term,table:SI_Term,dish:SI_Term)
            extends SI_Term
case class order_served(id:SI_Term,table:SI_Term,dish:SI_Term)
            extends SI_Term
case class last_order(id:SI_Term,table:SI_Term,dish:SI_Term)
            extends SI_Term
```

Client Code

```
def Client( id:SI_Term) = Agent {
                          LogIn(id) * EnjoyAltar(id) * LogOut(id) }

def LogIn(id:SI_Term) = Agent {
    tell(logged(id))  * tell(t_order(id)) * tell(t_order(id)) *
    tell(t_check(id)) * tell(t_check(id)) }
def LogOut(id:SI_Term) = Agent { get(logged(id)) }

def EnjoyAltar(id:SI_Term) = Agent {
    ( SitAtTable(id,t1) * EnjoyAtTable(id,t1) ) +
    ( SitAtTable(id,t2) * EnjoyAtTable(id,t2) ) +
    ( SitAtTable(id,t3) * EnjoyAtTable(id,t3) ) +
    ( SitAtTable(id,t4) * EnjoyAtTable(id,t4) ) }
```

```
14 def SitAtTable(id:SI_Term,t:SI_Term) = Comm {
15  "Client " + id.bsc_toString + " is sitting at table " + t.bsc_toString}
16
17 def EnjoyAtTable(id:SI_Term,t:SI_Term) = Agent {
18   MakeFirstOrder(id,t) * HandleOrders(id,t) * PayOrders(id,t) }
19
20 def PayOrders(id:SI_Term,t:SI_Term) = Comm {
21   id.bsc_toString + " at " + t.bsc_toString + " just paid " }
22
23 def HandleOrders(id:SI_Term,t:SI_Term): BSC_Agent = {
24   Agent {  ( get(t_order(id)) * MakeOrder(id,t)  * HandleOrders(id,t))
25          + ( get(t_check(id)) * CheckOrder(id,t) * HandleOrders(id,t))
26          + ( nask(t_order(id)))  } }
27
28 def MakeFirstOrder(id:SI_Term,t:SI_Term) = Agent {
29    get(t_order(id)) * MakeOrder(id,t) }
30
31 def MakeOrder(id:SI_Term,t:SI_Term) = Agent {
32    ( tell(order(id,t,nigiri)) * tell(last_order(id,t,nigiri)) )
33  + ( tell(order(id,t,maki))   * tell(last_order(id,t,maki)) )
34  + ( tell(order(id,t,temaki)) * tell(last_order(id,t,temaki)))
35  + ( tell(order(id,t,beer))   * tell(last_order(id,t,beer)) )}
36
37 def CheckOrder(id:SI_Term,t:SI_Term) = Agent {
38      CheckOrderDish(id,t,nigiri) + CheckOrderDish(id,t,maki) +
39      CheckOrderDish(id,t,temaki) + CheckOrderDish(id,t,beer) }
40
41 def CheckOrderDish(id:SI_Term,t:SI_Term,d:SI_Term) = Agent
42 {  val in_prepa = d.bsc_toString + " for " + id.bsc_toString + " in
       preparation "
43    val prepared = d.bsc_toString + " for " + id.bsc_toString + "
         prepared but not served "
44    val served   = d.bsc_toString + " for " + id.bsc_toString + " served
         "
45       ( ask(order(id,t,d)) * Comm{ in_prepa } )
46    + ( ask(last_order(id,t,d)) * (( ask(order_prepared(id,t,d))
47          * Comm{ prepared }  )
48    + ( ask(order_served(id,t,d)) * Comm{ served } ) ) )
49 }
```

Kitchen Code

```
1  def Cook(id:SI_Term): BSC_Agent = Agent {
2    ( CookForCustomer(id,manel) * Cook(id) ) +
3    ( CookForCustomer(id,alice) * Cook(id) ) +
4    ( CookForCustomer(id,emma)  * Cook(id) ) }
5
```

```
def CookForCustomer(id_cook:SI_Term,id_client:SI_Term) = Agent{
  CookForCustomerAtTable(id_cook,id_client,t1) +
  CookForCustomerAtTable(id_cook,id_client,t2) +
  CookForCustomerAtTable(id_cook,id_client,t3) +
  CookForCustomerAtTable(id_cook,id_client,t4) }

def CookForCustomerAtTable(id_cook:SI_Term,
  id_client:SI_Term,id_table:SI_Term) = Agent {
    ( get(order(id_client,id_table,nigiri)) *
      tell(order_prepared(id_client,id_table,nigiri))) +
    ( get(order(id_client,id_table,maki)) *
      tell(order_prepared(id_client,id_table,maki)))   +
    ( get(order(id_client,id_table,temaki)) *
      tell(order_prepared(id_client,id_table,temaki))) +
    ( get(order(id_client,id_table,beer)) *
      tell(order_prepared(id_client,id_table,beer)))   }
```

Waiter Code

```
def Waiter(id_waiter:SI_Term): BSC_Agent = Agent {
    ( WaiterForCustomer(id_waiter,manel) * Waiter(id_waiter) ) +
    ( WaiterForCustomer(id_waiter,alice) * Waiter(id_waiter) ) +
    ( WaiterForCustomer(id_waiter,emma)  * Cook(id_waiter) )   }

def WaiterForCustomer(id_waiter:SI_Term,id_client:SI_Term) = Agent {
    WaiterForCustomerAtTable(id_waiter,id_client,t1) +
    WaiterForCustomerAtTable(id_waiter,id_client,t2) +
    WaiterForCustomerAtTable(id_waiter,id_client,t3) +
    WaiterForCustomerAtTable(id_waiter,id_client,t4) }

def WaiterForCustomerAtTable(id_waiter:SI_Term,
    id_client:SI_Term,id_table:SI_Term) = Agent {
    ( get(order_prepared(id_client,id_table,nigiri))*
      tell(order_served(id_client,id_table,nigiri)))  +
    ( get(order_prepared(id_client,id_table,maki)) *
      tell(order_served(id_client,id_table,maki)) )   +
    ( get(order_prepared(id_client,id_table,temaki))*
      tell(order_served(id_client,id_table,temaki)) ) +
    ( get(order_prepared(id_client,id_table,beer)) *
      tell(order_served(id_client,id_table,beer)) )   }
```

People Code

```
1 val Manel      = Agent  { Client(manel) }
2 val Alice      = Agent  { Client(alice) }
3 val Emma       = Agent  { Client(emma)  }
4 val John       = Agent  { Cook(john)    }
5 val Tom        = Agent  { Waiter(tom)   }
6
7 val Restaurant = Agent { Manel || Alice || Emma || John || Tom }
```

Execution Code

```
1 val real_ag       = Restaurant
2 val bsc_executor = new BSC_Runner
3 bsc_executor.execute(real_ag)
```

APPENDIX B

RUSH HOUR GAME CODE

In this chapter, we present the complete code of our "Rush Hour" game example, initially implemented in Scala and featured in our article [63]. This presentation aims to provide a comprehensive understanding of the game's implementation, offering a detailed insight into the mechanisms employed in our approach.

Data Code

```
eqn down_car(1) = 3. down_car(2) = 4. down_car(3) = 5. down_car(4) = 6.
    down_truck(1) = 4. down_truck(2) = 5. down_truck(3) = 6.

    right_car(1) =3. right_car(2) =4. right_car(3) =5. right_car(4) =6.
    right_truck(1) = 4. right_truck(2) = 5. right_truck(3) = 6.

    pred(2) = 1. pred(3) = 2. pred(4) = 3. pred(5) = 4. pred(6) = 5.
    succ(1) = 2. succ(2) = 3. succ(3) = 4. succ(4) = 5. succ(5) = 6.

    img_truck(yellow) = yellow_truck. img_truck(green)  = green_truck.
    img_truck(blue)   = blue_truck.   img_truck(purple) = purple_truck.

    img_car(red)    = red_car. img_car(green) = green_car.
    img_car(orange) = orange_car.

    x_vehicle(1) = 55.  x_vehicle(2) = 130. x_vehicle(3) = 210.
    x_vehicle(4) = 285. x_vehicle(5) = 360. x_vehicle(6) = 435.

    y_vehicle(1) = 80.  y_vehicle(2) = 155. y_vehicle(3) = 230.
    y_vehicle(4) = 310. y_vehicle(5) = 385. y_vehicle(6) = 460.

    scene (562,618).

    background = loadImage(Images/rh_empty.png).
    yellow_truck = loadImage(Images/yellow_truck.png).
    green_truck = loadImage(Images/green_truck.png).
    blue_truck = loadImage(Images/blue_truck.png).
    purple_truck = loadImage(Images/purple_truck.png).
    red_car = loadImage(Images/red_car.png).
    green_car = loadImage(Images/green_car.png).
    orange_car = loadImage(Images/orange_car.png).
```

Procedures Code

```
proc VerticalCar(r: Rows, c: Cols, p: Colors) =
  ((r>1 & r<6) -> ( get(free(pred(r),c));
                    MoveCar(p,pred(r),c);
                    tell(free(succ(r),c));
                    VerticalCar(pred(r),c,p) ))
+ ((r<5)      ->  ( get(free(down_car(r),c));
                    MoveCar(p,succ(r),c);
                    tell(free(r,c));
                    VerticalCar(succ(r),c,p) )).

```

```
11 VerticalTruck(r: Rows, c: Cols, p: Colors) =
12   ((r>1 & r<5) -> ( get(free(pred(r),c));
13                     MoveTruck(p,pred(r),c);
14                     tell(free(succ(succ(r)),c));
15                     VerticalTruck(pred(r),c,p)))
16 + ((r<5)       -> ( get(free(down_truck(r),c));
17                     MoveTruck(p,succ(r),c);
18                     tell(free(r,c));
19                     VerticalTruck(succ(r),c,p) )).
20
21 HorizontalCar(r: Rows, c: Cols, p: Colors) =
22   ((c>1 & c<6) -> ( get(free(r,pred(c)));
23                     MoveCar(p,r,pred(c));
24                     tell(free(r,succ(c)));
25                     HorizontalCar(r,pred(c),p) ))
26 + ((c<5) ->       ( get(free(r,right_car(c)));
27                     MoveCar(p,r,succ(c));
28                     tell(free(r,c));
29                     HorizontalCar(r,succ(c),p) ))
30 + ((r=3 & c=5 & p=red) -> ( tell(out); MoveCar(p,r,succ(c)) )).
31
32 HorizontalTruck(r: Rows, c: Cols, p: Colors) =
33   ((c>1 & c<5) -> ( get(free(r,pred(c)));
34                     MoveTruck(p,r,pred(c));
35                     tell(free(r,succ(succ(c))));
36                     HorizontalTruck(r,pred(c),p)))
37 + ((c<5)       -> ( get(free(r,right_truck(c)));
38                     MoveTruck(p,r,succ(c));
39                     tell(free(r,c));
40                     HorizontalTruck(r,succ(c),p))).
41
42 MoveCar(p: Colors, r:Rows, c: Cols) =
43   ( p = red      -> move_to(red_car,x_vehicle(c),y_vehicle(r))    )
44 + ( p = green    -> move_to(green_car,x_vehicle(c),y_vehicle(r)) )
45 + ( p = organge -> move_to(orange_car,x_vehicle(c),y_vehicle(r))).
46
47 MoveTruck(p: Colors, r:Rows, c: Cols) =
48   ( p = yellow -> move_to(yellow_truck,x_vehicle(c),y_vehicle(r)))
49 + ( p = green  -> move_to(green_truck,x_vehicle(c),y_vehicle(r)) )
50 + ( p = blue   -> move_to(blue_truck,x_vehicle(c),y_vehicle(r))  )
51 + ( p = purple -> move_to(purple_truck,x_vehicle(c),y_vehicle(r))).
52
53 YellowTruck = VerticalTruck(1,1,yellow).
54 GreenTruck  = HorizontalTruck(6,3,green).
55 BlueTruck   = HorizontalTruck(4,4,blue).
56 PurpleTruck = VerticalTruck(1,4,purple).
57 RedCar      = HorizontalCar(3,2,red).
58 GreenCar    = VerticalCar(4,3,green).
59 OrangeCar   = VerticalCar(5,6,orange).
```

```
60
61 PlaceVehiclesOnScene =
62   draw_scene;
63   place_at(yellow_truck,56,78);    place_at(green_truck,208,457);
64   place_at(blue_truck,280,310);    place_at(purple_truck,282,77);
65   place_at(red_car,130,230);       place_at(green_car,207,307);
66   place_at(orange_car,437,385);
67
68   show(yellow_truck);
69   show(green_truck);
70   show(blue_truck);
71   show(purple_truck);
72   show(red_car);
73   show(green_car);
74   show(orange_car).
75
76 InitEmptyPlaces =
77   tell(free(1,2)); tell(free(1,3)); tell(free(1,5)); tell(free(1,6));
78   tell(free(2,2)); tell(free(2,3)); tell(free(2,5)); tell(free(2,6));
79   tell(free(3,5)); tell(free(3,6)); tell(free(4,1)); tell(free(4,2));
80   tell(free(5,1)); tell(free(5,2)); tell(free(5,4)); tell(free(5,5));
81   tell(free(6,1)); tell(free(6,2)).
82
83 LaunchVehicles =
84  ((YellowTruck || GreenTruck) || (BlueTruck || PurpleTruck)) ||
85   (RedCar || GreenCar || OrangeCar).
```

APPENDIX C

RUSH HOUR CODE WITH GUARDED LIST AND GLOBAL SEARCH

In this chapter, we present the complete code of our "Rush Hour" game example with the Guarded List construct and Global Search, initially implemented in Scala and featured in our article [15]. This presentation aims to provide a comprehensive understanding of the game's implementation, offering a detailed insight into the mechanisms employed in our approach.

Data Code

```
eset RCInt       = { 1, 2, 3, 4, 5, 6 }.
     Colors      = { yellow, green, blue, purple, red, orange, pink,
                     dark_green, black, brown }.
     RCPlaces    = { in_grid, out_grid }.
     Orientation = { v_orientation, h_orientation}.

eqn down_car(1)= 3.  down_car(2) = 4. down_car(3) = 5. down_car(4) = 6.
    down_truck(1) = 4.  down_truck(2) = 5.  down_truck(3) = 6.

    right_car(1)= 3. right_car(2)= 4. right_car(3)= 5. right_car(4)= 6.
    right_truck(1)= 4. right_truck(2)= 5. right_truck(3)= 6.

    pred(2) = 1.  pred(3) = 2.  pred(4) = 3.  pred(5) = 4.  pred(6) = 5.
    succ(1) = 2.  succ(2) = 3.  succ(3) = 4.  succ(4) = 5.  succ(5) = 6.

    img_truck(yellow)= yellow_truck. img_truck(green)= green_truck.
    img_truck(blue)= blue_truck.     img_truck(purple)= purple_truck.
    img_truck(orange)= orange_truck. img_truck(pink)= pink_truck.
    img_truck(brown)= brown_truck.   img_truck(black)= black_truck.
    img_truck(dark_green)= dark_green_truck.

    img_car(red) = red_car.        img_car(green) = green_car.
    img_car(orange) = orange_car. img_car(blue) = blue_car.
    img_car(yellow) = yellow_car. img_car(purple) = purple_car.
    img_car(pink) = pink_car.      img_car(black) = black_car.
    img_car(brown) = brown_car.    img_car(dark_green) = dark_green_car.

    x_vehicle(1) = 55.  x_vehicle(2) = 130. x_vehicle(3) = 210.
    x_vehicle(4) = 285. x_vehicle(5) = 360. x_vehicle(6) = 435.

    y_vehicle(1) = 80.  y_vehicle(2) = 155. y_vehicle(3) = 230.
    y_vehicle(4) = 310. y_vehicle(5) = 385. y_vehicle(6) = 460.

scene rhScene = {
    size = (562,618). layers = { top }.
    background = loadImage(Images/rh_empty.png).
    red_car_img_in = loadImage(Images/red_car_in.png).
    red_car_img_out = loadImage(Images/red_car_out.png).

    yellow_truck_img_v = loadImage(Images/v_yellow_truck.png).
    yellow_truck_img_h = loadImage(Images/h_yellow_truck.png).
    yellow_car_img_v = loadImage(Images/v_yellow_car.png).
    yellow_car_img_h = loadImage(Images/h_yellow_car.png).

    green_truck_img_v = loadImage(Images/v_green_truck.png).
    green_truck_img_h = loadImage(Images/h_green_truck.png).
```

```
47      green_car_img_v = loadImage(Images/v_green_car.png).
48      green_car_img_h = loadImage(Images/h_green_car.png).
49
50      blue_truck_img_v = loadImage(Images/v_blue_truck.png).
51      blue_truck_img_h = loadImage(Images/h_blue_truck.png).
52      blue_car_img_v = loadImage(Images/v_blue_car.png).
53      blue_car_img_h = loadImage(Images/h_blue_car.png).
54
55      purple_truck_img_v = loadImage(Images/v_purple_truck.png).
56      purple_truck_img_h = loadImage(Images/h_purple_truck.png).
57      purple_car_img_v = loadImage(Images/v_purple_car.png).
58      purple_car_img_h = loadImage(Images/h_purple_car.png).
59
60      orange_truck_img_v = loadImage(Images/v_orange_truck.png).
61      orange_truck_img_h = loadImage(Images/h_orange_truck.png).
62      orange_car_img_v = loadImage(Images/v_orange_car.png).
63      orange_car_img_h = loadImage(Images/h_orange_car.png).
64
65      pink_truck_img_v = loadImage(Images/v_pink_truck.png).
66      pink_truck_img_h = loadImage(Images/h_pink_truck.png).
67      pink_car_img_v = loadImage(Images/v_pink_car.png).
68      pink_car_img_h = loadImage(Images/h_pink_car.png).
69
70      dark_green_truck_img_v = loadImage(Images/v_dark_green_truck.png).
71      dark_green_truck_img_h = loadImage(Images/h_dark_green_truck.png).
72      dark_green_car_img_v = loadImage(Images/v_dark_green_car.png).
73      dark_green_car_img_h = loadImage(Images/h_dark_green_car.png).
74
75      black_truck_img_v = loadImage(Images/v_black_truck.png).
76      black_truck_img_h = loadImage(Images/h_black_truck.png).
77      black_car_img_v = loadImage(Images/v_black_car.png).
78      black_car_img_h = loadImage(Images/h_black_car.png).
79
80      brown_truck_img_v = loadImage(Images/v_brown_truck.png).
81      brown_truck_img_h = loadImage(Images/h_brown_truck.png).
82      brown_car_img_v = loadImage(Images/v_brown_car.png).
83      brown_car_img_h = loadImage(Images/h_brown_car.png).
84
85  widget yellow_truck = {
86      attributes = { sens in Orientation. }
87      display    = { sens = v_orientation -> yellow_truck_img_v.
88                     sens = h_orientation -> yellow_truck_img_h.}
89      init       = { wdL = top.  sens = h_orientation. } }
90
91  widget yellow_car = {
92      attributes = { sens in Orientation.}
93      display    = { sens = v_orientation -> yellow_car_img_v.
94                     sens = h_orientation -> yellow_car_img_h.}
95      init       = { wdL = top.  sens= h_orientation. }}
```

```
96  widget green_truck = {
97     attributes = { sens in Orientation.}
98     display    = { sens = v_orientation -> green_truck_img_v.
99                    sens = h_orientation -> green_truck_img_h.}
100    init       = { wdL = top.  sens = h_orientation. }  }
101
102 widget green_car = {
103    attributes = { sens in Orientation.}
104    display    = { sens = v_orientation -> green_car_img_v.
105                   sens = h_orientation -> green_car_img_h.  }
106    init       = { wdL = top.  sens = h_orientation. } }
107
108 widget blue_truck = {
109    attributes = { sens in Orientation.}
110    display    = { sens = v_orientation -> blue_truck_img_v.
111                   sens = h_orientation -> blue_truck_img_h.  }
112    init       = { wdL = top. sens = h_orientation. } }
113
114 widget blue_car = {
115    attributes = { sens in Orientation.}
116    display    = { sens = v_orientation -> blue_car_img_v.
117                   sens = h_orientation -> blue_car_img_h.  }
118    init       = { wdL = top.  sens = h_orientation. } }
119
120 widget purple_truck = {
121    attributes = { sens in Orientation.}
122    display    = { sens = v_orientation -> purple_truck_img_v.
123                   sens = h_orientation -> purple_truck_img_h.  }
124    init       = { wdL = top.  sens = h_orientation. } }
125
126 widget purple_car = {
127    attributes = { sens in Orientation.}
128    display    = { sens = v_orientation -> purple_car_img_v.
129                   sens = h_orientation -> purple_car_img_h.  }
130    init       = { wdL = top.  sens = h_orientation. } }
131
132 widget orange_truck= {
133    attributes = { sens in Orientation.}
134    display    = { sens = v_orientation -> orange_truck_img_v.
135                   sens = h_orientation -> orange_truck_img_h.  }
136    init       = { wdL = top. sens = h_orientation. } }
137
138 widget orange_car = {
139    attributes = { sens in Orientation.}
140    display    = { sens = v_orientation -> orange_car_img_v.
141                   sens = h_orientation -> orange_car_img_h.  }
142    init       = { wdL = top.  sens = h_orientation. } }
143
144
```

```
145 widget red_car = {
146    attributes = { position in RCPlaces. }
147    display    = { position = in_grid -> red_car_img_in.
148                   position = out_grid -> red_car_img_out. }
149    init       = { wdL = top. position = in_grid. } }
150
151 widget pink_car = {
152    attributes = { sens in Orientation.}
153    display    = { sens = v_orientation -> pink_car_img_v.
154                   sens = h_orientation -> pink_car_img_h.  }
155    init       = { wdL = top.  sens = h_orientation. } }
156
157 widget pink_truck = {
158    attributes = { sens in Orientation.}
159    display    = { sens = v_orientation -> pink_truck_img_v.
160                   sens = h_orientation -> pink_truck_img_h.  }
161    init       = { wdL = top.  sens = h_orientation. } }
162
163 widget dark_green_car = {
164    attributes = { sens in Orientation.}
165    display    = { sens = v_orientation -> dark_green_car_img_v.
166                   sens = h_orientation -> dark_green_car_img_h.  }
167    init       = { wdL = top.  sens = h_orientation. }   }
168
169 widget dark_green_truck = {
170    attributes = { sens in Orientation.}
171    display    = { sens = v_orientation -> dark_green_truck_img_v.
172                   sens = h_orientation -> dark_green_truck_img_h.  }
173    init       = { wdL = top.  sens = h_orientation. }   }
174
175 widget black_car = {
176    attributes = { sens in Orientation.}
177    display    = { sens = v_orientation -> black_car_img_v.
178                   sens = h_orientation -> black_car_img_h.  }
179    init       = { wdL = top.  sens = h_orientation. } }
180
181 widget black_truck = {
182    attributes = { sens in Orientation.}
183    display    = { sens = v_orientation -> black_truck_img_v.
184                   sens = h_orientation -> black_truck_img_h.  }
185    init       = { wdL = top.  sens = h_orientation. } }
186
187 widget brown_car = {
188    attributes = { sens in Orientation.}
189    display    = { sens = v_orientation -> brown_car_img_v.
190                   sens = h_orientation -> brown_car_img_h.  }
191    init       = { wdL = top. sens = h_orientation. }     }
192
193
```

```
194 widget brown_truck = {
195     attributes = { sens in Orientation.}
196     display    = { sens = v_orientation -> brown_truck_img_v.
197                    sens = h_orientation -> brown_truck_img_h.  }
198     init       = { wdL = top.  sens = h_orientation. }  }
199 }.
```

Procedures Code

```
1  proc VerticalCar(r: RCInt, c: RCInt, p: Colors) =
2    ((r>1 & r<6) -> ([ get(free(pred(r),c)) ->
3                       move_to(img_car(p),
4                       rhScene,x_vehicle(c),y_vehicle(pred(r))),
5                       tell(free(succ(r),c)) ];
6                       VerticalCar(pred(r),c,p)))
7
8  + ((r>=1 & r<5) ->([ get(free(down_car(r),c)) ->
9                      move_to(img_car(p),
10                     rhScene,x_vehicle(c),y_vehicle(succ(r))),
11                     tell(free(r,c)) ];
12                     VerticalCar(succ(r),c,p))).
13
14 VerticalTruck(r: RCInt, c: RCInt, p: Colors) =
15   ((r>1 & r<5) ->  ( [ get(free(pred(r),c)) ->
16        move_to(img_truck(p),rhScene,x_vehicle(c),y_vehicle(pred(r))),
17        tell( free( succ(succ(r)), c)) ];
18        VerticalTruck(pred(r),c,p)))
19 + ((r>=1 & r<4) -> ( [ get(free(down_truck(r),c)) ->
20        move_to(img_truck(p),rhScene,x_vehicle(c),y_vehicle(succ(r))),
21        tell(free(r,c)) ];
22        VerticalTruck(succ(r),c,p) )).
23
24 HorizontalCar(r: RCInt, c: RCInt, p: Colors) =
25   ((c>1 & c<6) ->  ( [ get(free(r,pred(c))) ->
26        move_to(img_car(p),rhScene,x_vehicle(pred(c)),y_vehicle(r)),
27        tell(free(r,succ(c))) ];
28        HorizontalCar(r,pred(c),p) ))
29 + ((c>=1 & c<5) -> ( [ get(free(r,right_car(c))) ->
30        move_to(img_car(p),rhScene,x_vehicle(succ(c)),y_vehicle(r)),
31        tell(free(r,c)) ];
32        HorizontalCar(r,succ(c),p) ))
33 + ((r=3 & c=5 & p=red) -> ( tell(out);
34        move_to(img_car(p),rhScene,x_vehicle(succ(c)),y_vehicle(r)))).
35
36 HorizontalTruck(r: RCInt, c: RCInt, p: Colors) =
37   ((c>1 & c<5) ->  ( [ get(free(r,pred(c))) ->
38        move_to(img_truck(p),rhScene,x_vehicle(pred(c)),y_vehicle(r)),
```

```
39          tell(free(r,succ(succ(c)))) ];
40          HorizontalTruck(r,pred(c),p) ))
41  +   ((c>=1 & c<4) -> ( [ get(free(r,right_truck(c))) ->
42          move_to(img_truck(p),rhScene,x_vehicle(succ(c)),y_vehicle(r)),
43          tell(free(r,c)) ];
44          HorizontalTruck(r,succ(c),p) )).
45
46  MoveCar(r:RCInt, c: RCInt, p: Colors) =
47           move_to(img_car(p),rhScene,x_vehicle(c),y_vehicle(r)).
48
49  MoveTruck(r:RCInt, c: RCInt, p: Colors) =
50           move_to(img_truck(p),rhScene,x_vehicle(c),y_vehicle(r)).
51
52  PlaceTruck(r: RCInt, c: RCInt, p: Colors) =
53           place_at(img_truck(p),rhScene,x_vehicle(c),y_vehicle(r)).
54
55  PlaceCar(r: RCInt, c: RCInt, p: Colors) =
56           place_at(img_car(p),rhScene,x_vehicle(c),y_vehicle(r)).
57
58  ShowTruck(p: Colors) =  show(img_truck(p),rhScene).
59  ShowCar(p: Colors)   =  show(img_car(p),rhScene).
60
61  InitFreePlaces       =  FreeRows(1).
62
63  FreeRows(r: RCInt) =
64    ((r<6) -> ( FreePlacesForRow(r,1) || FreeRows(succ(r)) ) )
65  + ((r=6) -> ( FreePlacesForRow(r,1) ) ).
66
67  FreePlacesForRow(r: RCInt,c: RCInt) =
68    ((c<6) -> ( tell(free(r,c)); FreePlacesForRow(r,succ(c)) ) )
69  + ((c=6) -> ( tell(free(r,c)) ) ).
70
71  VTruckPlaces(r: RCInt, c: RCInt) =
72    (r<5) -> get(free(r,c));
73             get(free(succ(r),c));
74             get(free(succ(succ(r)),c)).
75
76  VCarPlaces(r: RCInt, c: RCInt) =
77    (r<6) -> get(free(r,c));
78             get(free(succ(r),c)).
79
80  HTruckPlaces(r: RCInt, c: RCInt) =
81    (c<5) -> get(free(r,c));
82             get(free(r,succ(c)));
83             get(free(r,succ(succ(c)))).
84
85  HCarPlaces(r: RCInt, c: RCInt) =
86    (c<6) -> get(free(r,c));
87             get(free(r,succ(c))).
```

```
88  RedCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,red).
89
90  HGreenCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,green).
91  VGreenCar(r: RCInt, c: RCInt) = VerticalCar(r,c,green).
92  VGreenTruck(r: RCInt, c: RCInt)  = VerticalTruck(r,c,green).
93  HGreenTruck(r: RCInt, c: RCInt)  = HorizontalTruck(r,c,green).
94
95  HBlueCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,blue).
96  VBlueCar(r: RCInt, c: RCInt) = VerticalCar(r,c,blue).
97  VBlueTruck(r: RCInt, c: RCInt)   = VerticalTruck(r,c,blue).
98  HBlueTruck(r: RCInt, c: RCInt)   = HorizontalTruck(r,c,blue).
99
100 HPurpleCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,purple).
101 VPurpleCar(r: RCInt, c: RCInt) = VerticalCar(r,c,purple).
102 VPurpleTruck(r: RCInt, c: RCInt) = VerticalTruck(r,c,purple).
103 HPurpleTruck(r: RCInt, c: RCInt) = HorizontalTruck(r,c,purple).
104
105 HOrangeCar(r: RCInt, c: RCInt) = HorizantalCar(r,c,orange).
106 VOrangeCar(r: RCInt, c: RCInt) = VerticalCar(r,c,orange).
107 HBlackCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,black).
108 VBlackCar(r: RCInt, c: RCInt) = VerticalCar(r,c,black).
109
110 HBrownCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,brown).
111 VBrownCar(r: RCInt, c: RCInt) = VerticalCar(r,c,brown).
112
113 VPinkCar(r: RCInt, c: RCInt) = VerticalCar(r,c,pink).
114 HPinkCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,pink).
115
116 HDarkGreenCar(r: RCInt, c: RCInt) = HorizontalCar(r,c,dark_green).
117 VDarkGreenCar(r: RCInt, c: RCInt) = VerticalCar(r,c,dark_green).
118
119 HYellowTruck(r: RCInt, c: RCInt) = HorizontalTruck(r,c,yellow).
120 VYellowTruck(r: RCInt, c: RCInt) = VerticalTruck(r,c,yellow).
121
122 VYellowTruckAt(r: RCInt, c: RCInt) =
123       att(sens, yellow_truck, rhScene, v_orientation);
124       VTruckPlaces(r,c); PlaceTruck(r,c,yellow); ShowTruck(yellow).
125
126 HYellowTruckAt(r: RCInt, c: RCInt) =
127       att(sens, yellow_truck, rhScene, h_orientation);
128       HTruckPlaces(r,c); PlaceTruck(r,c,yellow); ShowTruck(yellow).
129
130 VGreenTruckAt(r: RCInt, c: RCInt) =
131       att(sens, green_truck, rhScene, v_orientation);
132       VTruckPlaces(r,c); PlaceTruck(r,c,green); ShowTruck(green).
133
134 HGreenTruckAt(r: RCInt, c: RCInt) =
135       att(sens, green_truck, rhScene, h_orientation);
136       HTruckPlaces(r,c); PlaceTruck(r,c,green); ShowTruck(green).
```

```
137
138 VBlueTruckAt(r: RCInt, c: RCInt) =
139       att(sens, blue_truck, rhScene, v_orientation);
140       VTruckPlaces(r,c); PlaceTruck(r,c,blue); ShowTruck(blue).
141
142 HBlueTruckAt(r: RCInt, c: RCInt) =
143       att(sens, blue_truck, rhScene, h_orientation);
144       HTruckPlaces(r,c); PlaceTruck(r,c,blue); ShowTruck(blue).
145
146 VPurpleTruckAt(r: RCInt, c: RCInt ) =
147       att(sens, purple_truck, rhScene, v_orientation);
148       VTruckPlaces(r,c); PlaceTruck(r,c,purple); ShowTruck(purple).
149
150 HPurpleTruckAt(r: RCInt, c: RCInt ) =
151       att(sens, purple_truck, rhScene, h_orientation);
152       HTruckPlaces(r,c); PlaceTruck(r,c,purple); ShowTruck(purple).
153
154 RedCarAt(r: RCInt, c: RCInt) =
155       HCarPlaces(r,c); PlaceCar(r,c,red); ShowCar(red).
156
157 VGreenCarAt(r: RCInt, c: RCInt) =
158       att(sens, green_car, rhScene, v_orientation);
159       VCarPlaces(r,c); PlaceCar(r,c,green); ShowCar(green).
160
161 HGreenCarAt(r: RCInt, c: RCInt) =
162       att(sens, green_car, rhScene, h_orientation);
163       HCarPlaces(r,c); PlaceCar(r,c,green); ShowCar(green).
164
165 VOrangeCarAt(r: RCInt, c: RCInt) =
166       att(sens, orange_car, rhScene, v_orientation);
167       VCarPlaces(r,c); PlaceCar(r,c,orange); ShowCar(orange).
168
169 HOrangeCarAt(r: RCInt, c: RCInt) =
170       att(sens,orange_car, rhScene,h_orientation);
171       HCarPlaces(r,c); PlaceCar(r,c,orange); ShowCar(orange).
172
173 VBlueCarAt(r: RCInt, c: RCInt) =
174       att(sens, blue_car, rhScene, v_orientation);
175       VCarPlaces(r,c); PlaceCar(r,c,blue); ShowCar(blue).
176
177 HBlueCarAt(r: RCInt, c: RCInt) =
178       att(sens, blue_car, rhScene, h_orientation); HCarPlaces(r,c);
179       PlaceCar(r,c,blue); ShowCar(blue).
180
181
182 VBlackCarAt(r: RCInt, c: RCInt) =
183       att(sens, black_car, rhScene, v_orientation);
184       VCarPlaces(r,c); PlaceCar(r,c,black); ShowCar(black).
185
```

```
186 HBlackCarAt(r: RCInt, c: RCInt) =
187       att(sens, black_car, rhScene, h_orientation);
188       HCarPlaces(r,c); PlaceCar(r,c,black); ShowCar(black).
189
190 VPurpleCarAt(r: RCInt, c: RCInt) =
191       att(sens, purple_car, rhScene, v_orientation);
192       VCarPlaces(r,c); PlaceCar(r,c,purple); ShowCar(purple).
193 HPurpleCarAt(r: RCInt, c: RCInt) =
194       att(sens, purple_car, rhScene, h_orientation);
195       HCarPlaces(r,c); PlaceCar(r,c,purple); ShowCar(purple).
196
197 VBrownCarAt(r: RCInt, c: RCInt) =
198       att(sens, brown, rhScene, v_orientation);
199       VCarPlaces(r,c); PlaceCar(r,c,brown); ShowCar(brown).
200
201 HBrownCarAt(r: RCInt, c: RCInt) =
202       att(sens, brown, rhScene, h_orientation);
203       HCarPlaces(r,c); PlaceCar(r,c,brown); ShowCar(brown).
204
205 VPinkCarAt(r: RCInt, c: RCInt) =
206       att(sens, pink_car, rhScene, v_orientation);
207       VCarPlaces(r,c); PlaceCar(r,c,pink); ShowCar(pink).
208
209 HPinkCarAt(r: RCInt, c: RCInt) =
210       att(sens, pink_car, rhScene, h_orientation);
211       HCarPlaces(r,c); PlaceCar(r,c,pink); ShowCar(pink).
212
213 VDarkGreenCarAt(r: RCInt, c: RCInt) =
214       att(sens, dark_green, rhScene, v_orientation);
215       VCarPlaces(r,c); PlaceCar(r,c,dark_green); ShowCar(dark_green).
216
217 HDarkGreenCarAt(r: RCInt, c: RCInt) =
218       att(sens, dark_green, rhScene, h_orientation);
219       HCarPlaces(r,c); PlaceCar(r,c,dark_green); ShowCar(dark_green).
220
221 VYellowTruckResetAt(r: RCInt, c: RCInt)= PlaceTruck(r,c,yellow).
222 HYellowTruckResetAt(r: RCInt, c: RCInt)= PlaceTruck(r,c,yellow).
223 VGreenTruckResetAt(r: RCInt, c: RCInt) = PlaceTruck(r,c,green).
224 HGreenTruckResetAt(r: RCInt, c: RCInt) = PlaceTruck(r,c,green).
225 VBlueTruckResetAt(r: RCInt, c: RCInt)  = PlaceTruck(r,c,blue).
226 HBlueTruckResetAt(r: RCInt, c: RCInt)  = PlaceTruck(r,c,blue).
227 VPurpleTruckResetAt(r: RCInt, c: RCInt ) = PlaceTruck(r,c,purple).
228 HPurpleTruckResetAt(r: RCInt, c: RCInt ) = PlaceTruck(r,c,purple).
229
230 RedCarResetAt(r: RCInt, c: RCInt)      = PlaceCar(r,c,red).
231 VGreenCarResetAt(r: RCInt, c: RCInt)  = PlaceCar(r,c,green).
232 HGreenCarResetAt(r: RCInt, c: RCInt)  = PlaceCar(r,c,green).
233 VOrangeCarResetAt(r: RCInt, c: RCInt) = PlaceCar(r,c,orange).
234 HOrangeCarResetAt(r: RCInt, c: RCInt) = PlaceCar(r,c,orange).
```

```
235 VBlueCarResetAt(r: RCInt, c: RCInt)    = PlaceCar(r,c,blue).
236 HBlueCarResetAt(r: RCInt, c: RCInt)    = PlaceCar(r,c,blue).
237 VBlackCarResetAt(r: RCInt, c: RCInt)   = PlaceCar(r,c,black).
238 HBlackCarResetAt(r: RCInt, c: RCInt)   = PlaceCar(r,c,black).
239 VPurpleCarResetAt(r: RCInt, c: RCInt)  = PlaceCar(r,c,purple).
240 HPurpleCarResetAt(r: RCInt, c: RCInt)  = PlaceCar(r,c,purple).
241 VBrownCarResetAt(r: RCInt, c: RCInt)   = PlaceCar(r,c,brown).
242 HBrownCarResetAt(r: RCInt, c: RCInt)   = PlaceCar(r,c,brown).
243 VPinkCarResetAt(r: RCInt, c: RCInt)    = PlaceCar(r,c,pink).
244 HPinkCarResetAt(r: RCInt, c: RCInt)    = PlaceCar(r,c,pink).
245 VDarkGreenCarResetAt(r: RCInt, c: RCInt) = PlaceCar(r,c,dark_green).
246 HDarkGreenCarResetAt(r: RCInt, c: RCInt) = PlaceCar(r,c,dark_green).
247
248 RedCarOut = att(position, red_car, rhscene, out_grid).
249
250 MoveTruckUp(tr: RCInt,tc:RCInt, tco: Colors) =
251  (tr>1 & tr<5) ->  ( [ get(free(pred(tr),tc)) ->
252       move_to(img_truck(tco),rhScene,x_vehicle(tc),y_vehicle(pred(tr))),
253       tell( free( succ(succ(tr)), tc )) ] ).
254
255 MoveTruckDown(tr: RCInt,tc:RCInt, tco: Colors) =
256  (tr<4)         -> ( [ get(free(down_truck(tr),tc)) ->
257     move_to(img_truck(tco),rhScene,x_vehicle(tc),y_vehicle(succ(tr))),
258     tell(free(tr,tc)) ] ).
259
260 MoveTruckLeft(tr: RCInt,tc:RCInt, tco: Colors) =
261  (tc>1 & tc<6) ->  ( [ get(free(tr,pred(tc))) ->
262       move_to(img_truck(tco),rhScene,x_vehicle(succ(tc)),y_vehicle(tr)),
263        tell(free(tr,succ(tc))) ] ).
264
265 MoveTruckRight(tr: RCInt,tc:RCInt, tco: Colors) =
266  (tc<5)         -> ( [ get(free(tr,right_car(tc))) ->
267       move_to(img_truck(tco),rhScene,x_vehicle(succ(tc)),y_vehicle(tr)),
268     tell(free(tr,tc)) ] ).
269
270 MoveCarLeft(cr: RCInt, cc: RCInt, cco: Colors) =
271  (cc>1 & cc<6) ->  ( [ get(free(cr,pred(cc))) ->
272       move_to(img_car(cco),rhScene,x_vehicle(succ(cc)),y_vehicle(cr)),
273       tell(free(cr,succ(cc))) ] ).
274
275 MoveCarRight(cr: RCInt, cc: RCInt, cco: Colors) =
276  (cc<5)         -> ( [ get(free(cr,right_car(cc))) ->
277     move_to(img_car(cco),rhScene,x_vehicle(succ(cc)),y_vehicle(cr)),
278     tell(free(cr,cc)) ] ).
279
280 MoveCarOut(cr: RCInt, cc: RCInt, cco: Colors) =
281 (cr=3 & cc=5 & cco=red) -> ( tell(out);
282     move_to(img_car(cco),rhScene,x_vehicle(succ(cc)),y_vehicle(cr)) ).
283
```

```
284 MoveCarUp(cr: RCInt, cc: RCInt, cco: Colors) =
285 (cr>1 & cr<5) ->  ( [ get(free(pred(cr),cc)) ->
286       move_to(img_car(cco),rhScene,x_vehicle(cc),y_vehicle(pred(cr))),
287       tell( free( succ(succ(cr)), cc )) ] ).
288
289 MoveCarDown(cr: RCInt, cc: RCInt, cco: Colors) =
290 (cr<4)        -> ( [ get(free(down_car(cr),cc)) ->
291       move_to(img_car(cco),rhScene,x_vehicle(cc),y_vehicle(succ(cr))),
292       tell(free(cr,cc)) ] ).
```

Game Scene Code

```
1  InitScene     = draw_scene(rhScene).
2
3  Init_game_1   = InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2).
4  Real_game_1   = ( VPurpleTruck(2,1) || RedCar(3,2) ).
5  Reset_game_1  = VPurpleTruckResetAt(2,1); RedCarResetAt(3,2).
6  Syst_game_1   = Syst1( 2,1,purple,  3,2,red ).
7
8  Syst1( tr: RCInt,tc:RCInt,tco: Colors,  rr: RCInt,rc:RCInt,rco:Colors) =
9
10    MoveTruckUp(tr,tc,tco); Syst1( pred(tr),tc,tco,  rr,rc,rco )
11 +  MoveTruckDown(tr,tc,tco); Syst1( succ(tr),tc,tco,  rr,rc,rco )
12 +  MoveCarLeft(rr,rc,rco); Syst1( tr,tc,tco,  rr,pred(rc),rco )
13 +  MoveCarRight(rr,rc,rco); Syst1( tr,tc,tco,  rr,succ(rc),rco )
14 +  MoveCarOut(rr,rc,rco).
15
16
17 Init_game_2   = InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2);
18                 HGreenCarAt(1,1).
19 Real_game_2   = VPurpleTruck(2,1) || RedCar(3,2) || HGreenCar(1,1).
20 Reset_game_2  = VPurpleTruckResetAt(2,1); RedCarResetAt(3,2);
21                 HGreenCarResetAt(1,1).
22 Syst_game_2   = Syst2( 2,1,purple,  3,2,red,  1,1,green).
23
24 Syst2(tr: RCInt,tc:RCInt,tco: Colors, rr: RCInt,rc:RCInt,rco:Colors,
25       hr:RCInt,hc:RCInt,hco:Colors) =
26
27    MoveTruckUp(tr,tc,tco); Syst2(pred(tr),tc,tco, rr,rc,rco, hr,hc,hco)
28 +  MoveTruckDown(tr,tc,tco); Syst2(succ(tr),tc,tco, rr,rc,rco,hr,hc,hco)
29 +  MoveCarLeft(rr,rc,rco); Syst2(tr,tc,tco, rr,pred(rc),rco, hr,hc,hco)
30 +  MoveCarRight(rr,rc,rco); Syst2(tr,tc,tco, rr,succ(rc),rco, hr,hc,hco)
31 +  MoveCarLeft(hr,hc,hco); Syst2(tr,tc,tco, rr,rc,rco, hr,pred(hc),hco)
32 +  MoveCarRight(hr,hc,hco); Syst2(tr,tc,tco, rr,rc,rco, hr,succ(hc),hco)
33 +  MoveCarOut(rr,rc,rco).
```

```
Init_game_3  = InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2);
               HGreenCarAt(1,1); VOrangeCarAt(5,1).
Real_game_3  = VPurpleTruck(2,1) || RedCar(3,2) || HGreenCar(1,1) ||
               VOrangeCar(5,1).
Reset_game_3 = VPurpleTruckResetAt(2,1); RedCarResetAt(3,2);
               HGreenCarResetAt(1,1); VOrangeCarResetAt(5,1).
Syst_game_3 =  Syst3( 2,1,purple,  3,2,red,  1,1,green, 5,1,orange).

Syst3(tr: RCInt,tc:RCInt,tco: Colors, rr: RCInt,rc:RCInt,rco:Colors,
      hr:RCInt,hc:RCInt,hco:Colors, vor:RCInt,voc:RCInt,voco:Colors) =

  MoveTruckUp(tr,tc,tco);
  Syst3( pred(tr),tc,tco,  rr,rc,rco,  hr,hc,hco, vor,voc,voco)
+ MoveTruckDown(tr,tc,tco);
  Syst3( succ(tr),tc,tco,  rr,rc,rco,  hr,hc,hco, vor,voc,voco)
+ MoveCarLeft(rr,rc,rco);
  Syst3( tr,tc,tco,  rr,pred(rc),rco,  hr,hc,hco, vor,voc,voco)
+ MoveCarRight(rr,rc,rco);
  Syst3( tr,tc,tco,  rr,succ(rc),rco,  hr,hc,hco, vor,voc,voco)
+ MoveCarLeft(hr,hc,hco);
  Syst3( tr,tc,tco,  rr,rc,rco,  hr,pred(hc),hco , vor,voc,voco)
+ MoveCarRight(hr,hc,hco);
  Syst3( tr,tc,tco,  rr,rc,rco,  hr,succ(hc),hco , vor,voc,voco)
+ MoveCarUp(vor,voc,voco);
  Syst3(tr,tc,tco ,  rr,rc,rco,  hr,hc,hco , pred(vor),voc,voco)
+ MoveCarDown(vor,voc,voco);
  Syst3(tr,tc,tco ,  rr,rc,rco,  hr,hc,hco , succ(vor),voc,voco)
+ MoveCarOut(rr,rc,rco).

Init_game_4  = InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2);
               HGreenCarAt(1,1); VOrangeCarAt(5,1); VBlueTruckAt(2,4).
Real_game_4  = VPurpleTruck(2,1) || RedCar(3,2) || HGreenCar(1,1) ||
               VOrangeCar(5,1) || VBlueTruck(2,4).
Reset_game_4 = VPurpleTruckResetAt(2,4); RedCarResetAt(3,2);
               HGreenCarResetAt(1,1); VOrangeCarResetAt(5,1);
               VBlueTruckResetAt(2,4).
Syst_game_4 = Syst4( 2,1,purple,  3,2,red,  1,1,green, 5,1,orange,
                     2,4,blue).

Syst4(tr: RCInt,tc:RCInt,tco: Colors, rr: RCInt,rc:RCInt,rco:Colors,
      hr:RCInt,hc:RCInt,hco:Colors, vor:RCInt,voc:RCInt,voco:Colors,
      vbtr:RCInt,vbtc:RCInt,vbtco:Colors) =

  MoveTruckUp(tr,tc,tco);
Syst4(pred(tr),tc,tco,rr,rc,rco,hr,hc,hco,vor,voc,voco,vbtr,vbtc,vbtco)

+ MoveTruckDown(tr,tc,tco);
Syst4(succ(tr),tc,tco, rr,rc,rco,hr,hc,hco,vor,voc,voco,vbtr,vbtc,vbtco)
```

```
49 +  MoveCarLeft(rr,rc,rco);
50 Syst4(tr,tc,tco,rr,pred(rc),rco,hr,hc,hco,vor,voc,voco,vbtr,vbtc,vbtco)
51
52 +  MoveCarRight(rr,rc,rco);
53 Syst4(tr,tc,tco,rr,succ(rc),rco,hr,hc,hco,vor,voc,voco,vbtr,vbtc,vbtco)
54
55 +  MoveCarLeft(hr,hc,hco);
56 Syst4(tr,tc,tco,rr,rc,rco,hr,pred(hc),hco,vor,voc,voco,vbtr,vbtc,vbtco)
57
58 +  MoveCarRight(hr,hc,hco);
59 Syst4(tr,tc,tco,rr,rc,rco,hr,succ(hc),hco,vor,voc,voco,vbtr,vbtc,vbtco)
60
61 +  MoveCarUp(vor,voc,voco);
62 Syst4(tr,tc,tco,rr,rc,rco,hr,hc,hco,pred(vor),voc,voco,vbtr,vbtc,vbtco)
63
64 +  MoveCarDown(vor,voc,voco);
65 Syst4(tr,tc,tco,rr,rc,rco,hr,hc,hco,succ(vor),voc,voco,vbtr,vbtc,vbtco)
66
67 +  MoveTruckUp(vbtr,vbtc,vbtco);
68 Syst4(tr,tc,tco,rr,rc,rco,hr,hc,hco,vor,voc,voco,pred(vbtr),vbtc,vbtco)
69
70 +  MoveTruckDown(vbtr,vbtc,vbtco);
71 Syst4(tr,tc,tco,rr,rc,rco,hr,hc,hco,vor,voc,voco,succ(vbtr),vbtc,vbtco)
72 +  MoveCarOut(rr,rc,rco).
73
74
75 Init_game_5 =  InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2);
76                HGreenCarAt(1,1); VOrangeCarAt(5,1); VBlueTruckAt(2,4);
77                HGreenTruckAt(6,3).
78 Real_game_5 =  VPurpleTruck(2,1) || RedCar(3,2) || HGreenCar(1,1) ||
79                VOrangeCar(5,1) || VBlueTruck(2,4) || HGreenTruck(6,3).
80 Reset_game_5 = VPurpleTruckResetAt(2,4); RedCarResetAt(3,2);
81                HGreenCarResetAt(1,1); VOrangeCarResetAt(5,1);
82                VBlueTruckResetAt(2,4); HGreenTruckResetAt(6,3).
83 Syst_game_5 = Syst5( 2,1,purple,  3,2,red,  1,1,green, 5,1,orange,
84                       2,4,blue, 6,3, green).
85
86 Syst5(tr: RCInt,tc:RCInt,tco: Colors, rr: RCInt,rc:RCInt,rco:Colors,
87       hr:RCInt,hc:RCInt,hco:Colors, vor:RCInt,voc:RCInt,voco:Colors,
88       vbtr:RCInt,vbtc:RCInt,vbtco:Colors, hgtr: RCInt,hgtc:RCInt,
89       hgtco:Colors) =
90
91 MoveTruckUp(tr,tc,tco);
92 Syst5(pred(tr),tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,
     vbtco, hgtr,hgtc,hgtco)
93
94 +  MoveTruckDown(tr,tc,tco);
95 Syst5(succ(tr),tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,
     vbtco, hgtr,hgtc,hgtco)
```

```
96
97  +  MoveCarLeft(rr,rc,rco);
98  Syst5(tr,tc,tco, rr,pred(rc),rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
99
100 +  MoveCarRight(rr,rc,rco);
101 Syst5(tr,tc,tco, rr,succ(rc),rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
102
103 +  MoveCarLeft(hr,hc,hco);
104 Syst5(tr,tc,tco, rr,rc,rco, hr,pred(hc),hco , vor,voc,voco , vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
105
106 +  MoveCarRight(hr,hc,hco);
107 Syst5(tr,tc,tco, rr,rc,rco, hr,succ(hc),hco , vor,voc,voco , vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
108
109 +  MoveCarUp(vor,voc,voco);
110 Syst5(tr,tc,tco, rr,rc,rco, hr,hc,hco , pred(vor),voc,voco, vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
111
112 +  MoveCarDown(vor,voc,voco);
113 Syst5(tr,tc,tco, rr,rc,rco, hr,hc,hco , succ(vor),voc,voco, vbtr,vbtc,
        vbtco, hgtr,hgtc,hgtco)
114
115 +  MoveTruckUp(vbtr,vbtc,vbtco);
116 Syst5(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, pred(vbtr),vbtc,
        vbtco, hgtr,hgtc,hgtco)
117
118 +  MoveTruckDown(vbtr,vbtc,vbtco);
119 Syst5(tr ,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, succ(vbtr),vbtc,
        vbtco, hgtr,hgtc,hgtco)
120
121 +  MoveTruckLeft(hgtr,hgtc,hgtco);
122 Syst5(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
        hgtr,pred(hgtc),hgtco)
123
124 +  MoveTruckRight(hgtr,hgtc,hgtco);
125 Syst5(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
        hgtr,succ(hgtc),hgtco)
126 +  MoveCarOut(rr,rc,rco).
```

```
1  Init_game_6 = InitFreePlaces; VPurpleTruckAt(2,1); RedCarAt(3,2);
2                HGreenCarAt(1,1); VOrangeCarAt(5,1); VBlueTruckAt(2,4);
3                HGreenTruckAt(6,3); VYellowTruckAt(1,6).
4
5  Real_game_6 = VPurpleTruck(2,1) || RedCar(3,2) || HGreenCar(1,1) ||
6                VOrangeCar(5,1) || VBlueTruck(2,4) || HGreenTruck(6,3) ||
7                VYellowTruck(1,6).
8
9  Reset_game_6 = VPurpleTruckResetAt(2,4); RedCarResetAt(3,2);
10                HGreenCarResetAt(1,1); VOrangeCarResetAt(5,1);
11                VBlueTruckResetAt(2,4); HGreenTruckResetAt(6,3);
12                VYellowTruckResetAt(1,6).
13 Syst_game_6 = Syst6( 2,1,purple,  3,2,red,  1,1,green,  5,1,orange,
14                      2,4,blue,  6,3,green,  1,6,yellow).
15
16 Syst6(tr:RCInt,tc:RCInt,tco: Colors,  rr: RCInt,rc:RCInt,rco:Colors,
17    hr:RCInt,hc:RCInt,hco:Colors, vor:RCInt,voc:RCInt,voco:Colors,
18    vbtr:RCInt,vbtc:RCInt,vbtco:Colors, hgtr:RCInt,hgtc:RCInt,
19    hgtco:Colors,vytr:RCInt,vytc:RCInt,vytco:Colors) =
20
21 MoveTruckUp(tr,tc,tco);
22 Syst6( pred(tr),tc,tco,  rr,rc,rco,  hr,hc,hco,  vor,voc,voco,  vbtr,
       vbtc,vbtco,  hgtr,hgtc,hgtco,  vytr,vytc,vytco)
23
24 +  MoveTruckDown(tr,tc,tco);
25 Syst6( succ(tr),tc,tco,  rr,rc,rco,  hr,hc,hco, vor,voc,voco, vbtr,vbtc,
       vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
26
27 +  MoveCarLeft(rr,rc,rco);
28 Syst6( tr,tc,tco,  rr,pred(rc),rco,  hr,hc,hco, vor,voc,voco, vbtr,vbtc,
       vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
29
30 +  MoveCarRight(rr,rc,rco);
31 Syst6( tr,tc,tco,  rr,succ(rc),rco,  hr,hc,hco, vor,voc,voco, vbtr,vbtc,
       vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
32
33 +  MoveCarLeft(hr,hc,hco);
34 Syst6( tr,tc,tco,  rr,rc,rco,  hr,pred(hc),hco , vor,voc,voco , vbtr,
       vbtc,vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
35
36 +  MoveCarRight(hr,hc,hco);
37 Syst6( tr,tc,tco,  rr,rc,rco,  hr,succ(hc),hco , vor,voc,voco , vbtr,
       vbtc,vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
38
39 +  MoveCarUp(vor,voc,voco);
40 Syst6(tr,tc,tco ,  rr,rc,rco,  hr,hc,hco , pred(vor),voc,voco, vbtr,vbtc
       ,vbtco,  hgtr,hgtc,hgtco, vytr,vytc,vytco)
41
```

```
42 + MoveCarDown(vor,voc,voco);
43 Syst6(tr,tc,tco , rr,rc,rco, hr,hc,hco , succ(vor),voc,voco, vbtr,vbtc
       ,vbtco, hgtr,hgtc,hgtco, vytr,vytc,vytco)
44
45 + MoveTruckUp(vbtr,vbtc,vbtco);
46 Syst6(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, pred(vbtr),vbtc,
       vbtco, hgtr,hgtc,hgtco, vytr,vytc,vytco)
47
48 + MoveTruckDown(vbtr,vbtc,vbtco);
49 Syst6(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, succ(vbtr),vbtc,
       vbtco, hgtr,hgtc,hgtco, vytr,vytc,vytco)
50
51 + MoveTruckLeft(hgtr,hgtc,hgtco);
52 Syst6(tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
       hgtr,pred(hgtc),hgtco, vytr,vytc,vytco)
53
54 + MoveTruckRight(hgtr,hgtc,hgtco);
55 Syst6( tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
       hgtr,succ(hgtc),hgtco, vytr,vytc,vytco)
56
57 + MoveTruckUp(vytr,vytc,vytco);
58 Syst6( tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
       hgtr,hgtc,hgtco, pred(vytr),vytc,vytco)
59
60 + MoveTruckDown(vytr,vytc,vytco);
61 Syst6( tr,tc,tco, rr,rc,rco, hr,hc,hco, vor,voc,voco, vbtr,vbtc,vbtco,
       hgtr,hgtc,hgtco, succ(vytr),vytc,vytco)
62 + MoveCarOut(rr,rc,rco).
```

APPENDIX D

SMART ROLE-PLAYING GAME

In this chapter, we present the complete code for our "Smart Role-Playing Game" IoT application in socio-technical systems, introduced in the chapter 8. This presentation aims to provide a complete understanding of the game implementation, offering a detailed overview of the mechanisms employed in our approach.

Data Code

```
eset Positions = { haut1, haut2, haut3, haut4, haut5, mihaut5, mihaut4,
                   mihaut3, mihaut2, mihaut1, mibas1, mibas2, mibas3,
                   mibas4, mibas5, bas5, bas4, bas3, bas2, bas1 }.
     Decors = { foret, montagne, volcan, desert }.
     NumeroDe = { de1, de2, de3, de4, de5, de6 }.
     CouleursLeds = { rouge, blanc, jaune }.
     EtatCoffre = { ouvert, ferme }.
     SensJoueur = { droit, reverse }.

eqn x_coord(haut1) = 150. y_coord(haut1) = 50.
    x_coord(haut2) = 514. y_coord(haut2) = 50.
    x_coord(haut3) = 878. y_coord(haut3) = 50.
    x_coord(haut4) = 1242.  y_coord(haut4) = 50.
    x_coord(haut5) = 1606.  y_coord(haut5) = 50.
    x_coord(mihaut1) = 150. y_coord(mihaut1) = 295.
    x_coord(mihaut2) = 514. y_coord(mihaut2) = 295.
    x_coord(mihaut3) = 878. y_coord(mihaut3) = 295.
    x_coord(mihaut4) = 1242. y_coord(mihaut4) = 295.
    x_coord(mihaut5) = 1606. y_coord(mihaut5) = 295.
    x_coord(mibas1) = 150. y_coord(mibas1) = 540.
    x_coord(mibas2) = 514. y_coord(mibas2) = 540.
    x_coord(mibas3) = 878. y_coord(mibas3) = 540.
    x_coord(mibas4) = 1242. y_coord(mibas4) = 540.
    x_coord(mibas5) = 1606. y_coord(mibas5) = 540.
    x_coord(bas1) = 150. y_coord(bas1) = 785.
    x_coord(bas2) = 514. y_coord(bas2) = 785.
    x_coord(bas3) = 878. y_coord(bas3) = 785.
    x_coord(bas4) = 1242. y_coord(bas4) = 785.
    x_coord(bas5) = 1606. y_coord(bas5) = 785.

    succ(haut1)  = haut2.   succ(haut2) = haut3.
    succ(haut3)  = haut4.   succ(haut4) = haut5.
    succ(haut5)  = haut5.   succ(mihaut1) = mihaut2.
    succ(mihaut2)= mihaut3. succ(mihaut3) = mihaut4.
    succ(mihaut4)= mihaut5. succ(mihaut5) = mihaut5.
    succ(mibas1) = mibas2.  succ(mibas2) = mibas3.
    succ(mibas3) = mibas4.  succ(mibas4) = mibas5.
    succ(mibas5) = mibas5.  succ(bas1) = bas2.
    succ(bas2)   = bas3.    succ(bas3) = bas4.
    succ(bas4)   = bas5.    succ(bas5) = bas5.

    pred(haut1) = haut1.  pred(haut2) = haut1.
    pred(haut3) = haut2.  pred(haut4) = haut3.
    pred(haut5) = haut4.  pred(mihaut1) = mihaut1.
    pred(mihaut2) = mihaut1. pred(mihaut3) = mihaut2.
    pred(mihaut4) = mihaut3. pred(mihaut5) = mihaut4.
```

```
47      pred(mibas1) = mibas1.   pred(mibas2) = mibas1.
48      pred(mibas3) = mibas2.   pred(mibas4) = mibas3.
49      pred(mibas5) = mibas4.   pred(bas1) = bas1.
50      pred(bas2) = bas1.       pred(bas3) = bas2.
51      pred(bas4) = bas3.       pred(bas5) = bas4.
52
53      succv(haut1) = mihaut1.  succv(haut2) = mihaut2.
54      succv(haut3) = mihaut3.  succv(haut4) = mihaut4.
55      succv(haut5) = mihaut5.  succv(mihaut1) = mibas1.
56      succv(mihaut2) = mibas2. succv(mihaut3) = mibas3.
57      succv(mihaut4) = mibas4. succv(mihaut5) = mibas5.
58      succv(mibas1) = bas1.    succv(mibas2) = bas2.
59      succv(mibas3) = bas3.    succv(mibas4) = bas4.
60      succv(mibas5) = bas5.    succv(bas1) = bas1.
61      succv(bas2) = bas2.      succv(bas3) = bas3.
62      succv(bas4) = bas4.      succv(bas5) = bas5.
63
64      predv(haut1) = haut1.    predv(haut2) = haut2.
65      predv(haut3) = haut3.    predv(haut4) = haut4.
66      predv(haut5) = haut5.    predv(mihaut1) = haut1.
67      predv(mihaut2) = haut2.  predv(mihaut3) = haut3.
68      predv(mihaut4) = haut4.  predv(mihaut5) = haut5.
69      predv(mibas1) = mihaut1. predv(mibas2) = mihaut2.
70      predv(mibas3) = mihaut3. predv(mibas4) = mihaut4.
71      predv(mibas5) = mihaut5. predv(bas1) = mibas1.
72      predv(bas2) = mibas2.    predv(bas3) = mibas3.
73      predv(bas4) = mibas4.    predv(bas5) = mibas5.
```

Scene Code

```
1  scene smartrpgScene = {
2      size = (1920, 1080).
3      layers = { top }.
4
5      background = loadImage(Images/table.jpg).
6      pion1_img = loadImage(Images/pion1.png).
7      pion2_img = loadImage(Images/pion2.png).
8      pion3_img = loadImage(Images/pion3.png).
9
10     pion1Inverse_img = loadImage(Images/pion1reverse.png).
11     pion2Inverse_img = loadImage(Images/pion2reverse.png).
12     pion3Inverse_img = loadImage(Images/pion3reverse.png).
13     desert_img = loadImage(Images/desert.png).
14     foret_img = loadImage(Images/foret.png).
15     montagne_img = loadImage(Images/montagne.png).
16     volcan_img = loadImage(Images/volcan.png).
17
```

```
18    de1_img = loadImage(Images/d1.PNG).
19    de2_img = loadImage(Images/d2.PNG).
20    de3_img = loadImage(Images/d3.PNG).
21    de4_img = loadImage(Images/d4.PNG).
22    de5_img = loadImage(Images/d5.PNG).
23    de6_img = loadImage(Images/d6.PNG).
24
25    rouge_img = loadImage(Images/led_purple.png).
26    blanc_img = loadImage(Images/led_white.png).
27    jaune_img = loadImage(Images/led_orange.png).
28    coffreOuvert_img = loadImage(Images/coffreOuvert.png).
29    coffreFerme_img = loadImage(Images/coffreFerme.png).
30
31    widget decors = {
32        attributes = { decor in Decors. }
33        display    = { decor = foret -> foret_img.
34                       decor = montagne -> montagne_img.
35                       decor = desert -> desert_img.
36                       decor = volcan -> volcan_img.  }
37       init        = {    wdL = top.    decor = foret.}
38    }
39
40    widget pion1 = {
41        attributes = { sens in SensJoueur. }
42        display    = { sens = droit -> pion1_img.
43                       sens = reverse -> pion1Inverse_img. }
44        init       = {  wdL = top.       sens = reverse. }
45    }
46
47    widget pion2 = {
48        attributes = { sens in SensJoueur. }
49        display    = { sens = droit -> pion2_img.
50                       sens = reverse -> pion2Inverse_img. }
51        init       = {  wdL = top.       sens = reverse. }
52    }
53
54    widget pion3 = {
55        attributes = { sens in SensJoueur. }
56        display    = { sens = droit -> pion3_img.
57                       sens = reverse -> pion3Inverse_img. }
58        init       = {  wdL = top.     sens = droit.     }
59    }
60
61    widget coffre = {
62        attributes = { statut in EtatCoffre. }
63        display    = { statut = ouvert -> coffreOuvert_img.
64                       statut = ferme -> coffreFerme_img.    }
65        init       = {   wdL = top.    statut = ferme.  }
66    }
```

```
67
68      widget de = {
69          attributes = { numero in NumeroDe. }
70          display    = { numero = de1 -> de1_img.
71                         numero = de2 -> de2_img.
72                         numero = de3 -> de3_img.
73                         numero = de4 -> de4_img.
74                         numero = de5 -> de5_img.
75                         numero = de6 -> de6_img. }
76          init       = {  wdL = top.   numero = de1.  }
77      }
78
79      widget led = {
80          attributes = { couleur in CouleursLeds. }
81          display    = { couleur = rouge -> rouge_img.
82                         couleur = blanc -> blanc_img.
83                         couleur = jaune -> jaune_img. }
84          init       = {   wdL = top.   couleur = blanc. }
85      }
86  }.
```

Agents Code

```
1  proc Pion1(p: Positions) = get(Pion1);  RollDice;
2  ((( nask(succ(p)); tell(succ(p));
3    ( att(sens, pion1, smartrpgScene, reverse));
4      move_to(pion1, smartrpgScene, x_coord(succ(p)), y_coord(succ(p)));
5      get(p);
6  (((succ(p) = mibas3) -> OpenCoffre) + (tell(Pion2); Pion1(succ(p)))))
7   +
8  (( nask(pred(p));
9     tell(pred(p));
10   (att(sens, pion1, smartrpgScene, droit));
11    move_to(pion1, smartrpgScene, x_coord(pred(p)), y_coord(pred(p)));
12    get(p);
13 (((pred(p) = mibas3) -> OpenCoffre) + (tell(Pion2); Pion1(pred(p))))))
14 +
15 (( nask(succv(p));
16    tell(succv(p));
17    move_to(pion1, smartrpgScene,x_coord(succv(p)), y_coord(succv(p)));
18    get(p);
19 (((succv(p) = mibas3) -> OpenCoffre) + (tell(Pion2); Pion1(succv(p)))))
20 +
21 ( nask(predv(p));
22   tell(predv(p));
23   move_to(pion1, smartrpgScene, x_coord(predv(p)), y_coord(predv(p)));
24   get(p);
```

```
25 (((predv(p) = mibas3) -> OpenCoffre) + (tell(Pion2); Pion1(predv(p))))))
26 +
27 ( move_to(pion1, smartrpgScene, x_coord(p), y_coord(p));
28   tell(Pion2);
29   Pion1(p))    ).
30
31 Pion2(p: Positions) = get(Pion2);  RollDice;
32 ((( nask(succ(p));
33     tell(succ(p));
34    (att(sens, pion2, smartrpgScene, reverse));
35     move_to(pion2, smartrpgScene, x_coord(succ(p)), y_coord(succ(p)));
36     get(p);
37 (((succ(p) = mibas3) -> OpenCoffre) + (tell(Pion3); Pion2(succ(p)))))
38 +
39 ( nask(pred(p));
40   tell(pred(p));
41   (att(sens, pion2, smartrpgScene, droit));
42   move_to(pion2, smartrpgScene, x_coord(pred(p)), y_coord(pred(p)));
43   get(p);
44 (((pred(p) = mibas3) -> OpenCoffre) + (tell(Pion3); Pion2(pred(p))))))
45 +
46 (( nask(succv(p));
47    tell(succv(p));
48    move_to(pion2, smartrpgScene, x_coord(succv(p)), y_coord(succv(p)));
49    get(p);
50 (((succv(p) = mibas3) -> OpenCoffre) + (tell(Pion3);  Pion2(succv(p)))))
51 +
52 ( nask(predv(p));
53   tell(predv(p));
54   move_to(pion2, smartrpgScene, x_coord(predv(p)), y_coord(predv(p)));
55   get(p);
56 (((predv(p)= mibas3) -> OpenCoffre) + (tell(Pion3); Pion2(predv(p))))))
57 +
58 ( move_to(pion2, smartrpgScene, x_coord(p), y_coord(p));
59   tell(Pion3);
60   Pion2(p)  )).
61
62 Pion3(p: Positions) = get(Pion3);   RollDice;
63 ((( nask(succ(p));
64     tell(succ(p));
65     (att(sens, pion3, smartrpgScene, reverse));
66     move_to(pion3, smartrpgScene, x_coord(succ(p)), y_coord(succ(p)));
67     get(p);
68 (((succ(p) = mibas3) -> OpenCoffre) + (tell(Pion1); Pion3(succ(p)))) )
69 +
70 (  nask(pred(p));
71    tell(pred(p));
72    (att(sens, pion3, smartrpgScene, droit));
73    move_to(pion3, smartrpgScene, x_coord(pred(p)), y_coord(pred(p)));
```

```
74     get(p);
75  (((pred(p) = mibas3) -> OpenCoffre) + (tell(Pion1);  Pion3(pred(p))))))
76  +
77  (( nask(succv(p));
78     tell(succv(p));
79     move_to(pion3, smartrpgScene, x_coord(succv(p)), y_coord(succv(p)));
80     get(p);
81  (((succv(p) = mibas3) -> OpenCoffre) + (tell(Pion1); Pion3(succv(p)))))
82  +
83  (  nask(predv(p));
84     tell(predv(p));
85     move_to(pion3, smartrpgScene, x_coord(predv(p)), y_coord(predv(p)));
86     get(p);
87  (((predv(p) = mibas3) -> OpenCoffre) + (tell(Pion1); Pion3(predv(p))))))
88  +
89  (  move_to(pion3, smartrpgScene, x_coord(p), y_coord(p));
90     tell(Pion1);
91     Pion3(p) )).
92
93  ChangeDecor =  ((att(decor, decors, smartrpgScene, foret))
94                 + (att(decor, decors, smartrpgScene, montagne))
95                 + (att(decor, decors, smartrpgScene, volcan))
96                 + (att(decor, decors, smartrpgScene, desert)));
97                 ChangeDecor.
98
99  OpenCoffre = ask(mibas3);
100              (att(statut, coffre, smartrpgScene, ouvert));
101               tell(open).
102
103 RollDice = (
104 (    ((att(numero, de, smartrpgScene, de1)) ;
105      show(de, smartrpgScene); (att(couleur, led, smartrpgScene, rouge)))
106   + ((att(numero, de, smartrpgScene, de6));
107      show(de, smartrpgScene); (att(couleur, led, smartrpgScene, jaune))
108 )
109 +
110 ( (((att(numero, de, smartrpgScene, de2)) ;
111      show(de, smartrpgScene); (att(couleur, led, smartrpgScene, blanc)))
112   + ((att(numero, de, smartrpgScene, de3)) ;
113      show(de, smartrpgScene); (att(couleur, led, smartrpgScene, blanc)))
114 )
115 +
116   ((att(numero, de, smartrpgScene, de4)) ;
117     show(de, smartrpgScene); (att(couleur, led, smartrpgScene, blanc)))
118 + ((att(numero, de, smartrpgScene, de5)) ;
119     show(de, smartrpgScene); (att(couleur, led, smartrpgScene, blanc))))
120 ));
121
122 hide(de, smartrpgScene).
```

```
1  Init = draw_scene(smartrpgScene);
2
3          place_at(decors, smartrpgScene, 0, 0);
4          show(decors, smartrpgScene);
5
6          place_at(led, smartrpgScene, 0, 0);
7          show(led, smartrpgScene);
8
9          place_at(de, smartrpgScene, 1675, 900);
10
11         place_at(coffre, smartrpgScene, x_coord(mibas3),y_coord(mibas3));
12         show(coffre, smartrpgScene);
13
14         place_at(pion1, smartrpgScene, x_coord(haut1), y_coord(haut1));
15         place_at(pion2, smartrpgScene, x_coord(bas1), y_coord(bas1));
16         place_at(pion3, smartrpgScene, x_coord(haut5), y_coord(haut5));
17
18         show(pion1, smartrpgScene);
19         show(pion2, smartrpgScene);
20         show(pion3, smartrpgScene);
21
22         tell(haut1);
23         tell(bas1);
24         tell(haut5);
25         tell(Pion1);
26
27         Main(haut1, bas1, haut5).
28
29 Main(p1: Positions, p2: Positions, p3: Positions) =
30     (( Pion1(p1) ||  Pion2(p2) ) ||  Pion3(p3) ) ||  ChangeDecor.
```

BIBLIOGRAPHY

[1] J-R. Abrial. **Modeling in Event-B - System and Software Engineering**. Cambridge University Press, 2010 (cit. on p. 99).

[2] L. Acciai, M. Boreale, and S. Dal-Zilio. "A Concurrent Calculus with Atomic Transactions". In: **Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings**. Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 48–63 (cit. on p. 100).

[3] G. Agha. **Actors: a Model of Concurrent Computation in Distributed Systems**. MIT press, 1986 (cit. on pp. xx, 8).

[4] Y. Abd Alrahman, R. De Nicola, and M. Loreti. "On the Power of Attribute-Based Communication". In: **Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings**. Ed. by Elvira Albert and Ivan Lanese. Vol. 9688. Lecture Notes in Computer Science. Springer, 2016, pp. 1–18 (cit. on p. 15).

[5] J-M. Andreoli et al. "Multiparty Negotiation of Dynamic Distributed Object Services". In: **Sci. Comput. Program.** 31.2-3 (1998), pp. 179–203 (cit. on p. 9).

[6] G. R. Andrews. "Synchronizing Resources". In: **ACM Trans. Program. Lang. Syst.** 3.4 (1981), pp. 405–430 (cit. on p. 4).

[7] Ö. Babaoglu et al. "Design Patterns from Biology for Distributed Computing". In: **ACM Trans. Auton. Adapt. Syst.** 1.1 (2006), pp. 26–66 (cit. on p. 17).

[8] C. Baier and J-P. Katoen. **Principles of Model Checking**. MIT Press, 2008 (cit. on pp. 55, 57).

[9] M. Baldoni, C. Baroglio, and F. Capuzzimati. "A Commitment-Based Infrastructure for Programming Socio-Technical Systems". In: **ACM Trans. Internet Techn.** 14.4 (2014), 23:1–23:23 (cit. on p. 18).

[10] M. Baldoni, C. Baroglio, and F. Capuzzimati. “Typing Multi-Agent Systems via Commitments”. In: **Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers**. Ed. by Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk. Vol. 8758. Lecture Notes in Computer Science. Springer, 2014, pp. 388–405 (cit. on p. 18).

[11] M. Baldoni et al. “Commitment-based Agent Interaction in JaCaMo+”. In: **Fundam. Informaticae** 159.1-2 (2018), pp. 1–33 (cit. on p. 19).

[12] J-P. Banâtre and D. Le Métayer. “Gamma and the Chemical Reaction Model: Ten Years After”. In: **Coordination programming: mechanisms, models and semantics** (1996), pp. 3–41 (cit. on p. 9).

[13] M. Barkallah and J-M. Jacquet. **Anemone**. https://github.com/UNamurCSFaculty/anemone. Accessed May 19, 2025. 2021 (cit. on pp. 69, 81, 154).

[14] M. Barkallah and J-M. Jacquet. **B2Scala**. https://github.com/UNamurCSFaculty/B2Scala. Accessed May 19, 2025. 2023 (cit. on p. 38).

[15] M. Barkallah and J-M. Jacquet. “On the Expressiveness and Efficiency of Guarded Lists in Bach”. In: **J. Log. Algebraic Methods Program.** 142 (2025), p. 101017 (cit. on pp. xxv, 89, 171).

[16] M. Barkallah and J-M. Jacquet. “On the Introduction of Guarded Lists in Bach: Expressiveness, Correctness, and Efficiency Issues”. In: EPTCS 383 (2023). Ed. by Clément Aubert et al., pp. 55–72 (cit. on pp. xxiv, xxv, 89, 100).

[17] M. Barkallah and J.-M. Jacquet. **The Rush Hour Bach program**. https://staff.info.unamur.be/mbarkall/ICE_2023, or https://staff.info.unamur.be/jmj/ICE_2023. Created on May 30th 2023. 2023 (cit. on pp. 97–99).

[18] L. Bettini, E. Merelli, and F. Tiezzi. “X-Klaim is Back”. In: **Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday**. Ed. by Michele Boreale et al. Vol. 11665. Lecture Notes in Computer Science. Springer, 2019, pp. 115–135 (cit. on p. 83).

[19] A. Biere et al. “Symbolic Model Checking Using SAT Procedures instead of BDDs”. In: **Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999**. Ed. by Mary Jane Irwin. ACM Press, 1999, pp. 317–320 (cit. on p. 156).

[20] Ian E Blanchard et al. “Emergency Medical Services Response Time and Mortality in an Urban Setting”. In: **Prehospital emergency care** 16.1 (2012), pp. 142–151 (cit. on p. 119).

[21] F. S. de Boer and C. Palamidessi. “Embedding as a Tool for Language Comparison”. In: **Inf. Comput.** 108.1 (1994), pp. 128–157 (cit. on pp. 104, 105).

[22] O. Boissier et al. “Multi-Agent Oriented Programming with JaCaMo”. In: **Sci. Comput. Program.** 78.6 (2013), pp. 747–761 (cit. on p. 19).

[23] R. H. Bordini, Jo. F. Hübner, and M. Wooldridge. **Programming Multi-Agent Systems in AgentSpeak using Jason**. John Wiley & Sons, 2007 (cit. on p. 19).

[24] K. de Bosschere and J.-M. Jacquet. "$\mu^2 Log$: Towards Remote Coordination". In: **Proceedings of the International Conference on Coordination Models and Languages**. Ed. by P. Ciancarini and C. Hankin. Lecture Notes in Computer Science. Springer-Verlag, 1996, pp. 34–43 (cit. on p. 16).

[25] M. Bravetti and G. Zavattaro. "Foundations of Coordination and Contracts and Their Contribution to Session Type Theory". In: **Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings**. Ed. by Giovanna Di Marzo Serugendo and Michele Loreti. Vol. 10852. Lecture Notes in Computer Science. Springer, 2018, pp. 21–50 (cit. on pp. xix, 14).

[26] M. Bravetti and G. Zavattaro. "Service Oriented Computing from a Process Algebraic Perspective". In: **J. Log. Algebraic Methods Program.** 70.1 (2007), pp. 3–14 (cit. on pp. xix, 14).

[27] "Global Software Engineering: The Future of Socio-Technical Coordination". In: **International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA**. Ed. by L. C. Briand and A. L. Wolf. IEEE Computer Society, 2007, pp. 188–198 (cit. on p. 133).

[28] A. Brogi and J-M. Jacquet. "On the Expressiveness of Coordination via Shared Dataspaces". In: **Science of Computer Programming** 46.1-2 (2003), pp. 71–98 (cit. on pp. xix, xx, 12, 14, 104, 106, 108, 109, 111, 112).

[29] A. Brogi and J.-M. Jacquet. "On the Expressiveness of Linda-like Concurrent Languages". In: Electronic Notes in Theoretical Computer Science 16.2 (1998). Ed. by Ilaria Castellani and Catuscia Palamidessi, pp. 75–96 (cit. on pp. xx, 104, 106, 115, 116).

[30] N. Busi, R. Gorrieri, and G. Zavattaro. "On the Expressiveness of Linda Coordination Primitives". In: **Inf. Comput.** 156.1-2 (2000), pp. 90–121 (cit. on p. 116).

[31] N. Busi et al. "Coordination Models: A Guided Tour". In: **Coordination of Internet Agents: Models, Technologies, and Applications**. Ed. by Andrea Omicini et al. Springer, 2001, pp. 6–24 (cit. on p. 8).

[32] F. Calzolai et al. "TAPAs: A Tool for the Analysis of Process Algebras". In: **Trans. Petri Nets Other Model. Concurr.** 1 (2008), pp. 54–70 (cit. on pp. 66, 67, 82).

[33] M. W. A. Caminada et al. "Scrutable Plan Enactment via Argumentation and Natural Language Generation". In: **International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014**. Ed. by Ana L. C. Bazzan et al. IFAAMAS/ACM, 2014, pp. 1625–1626 (cit. on p. 19).

[34] N. Carriero and D. Gelernter. “Linda in Context”. In: **Commun. ACM** 32.4 (1989), pp. 444–458 (cit. on pp. xx, 9, 24, 43).

[35] C. Castlefranchi. “From Conversation to Interaction via Behavioral Communication: For a Semiotic Design of Objects, Environments, and Behaviors”. In: **Theories and practice in interaction design** (2006), pp. 157–79 (cit. on p. 16).

[36] A. K. Chopra and M. P. Singh. “From Social Machines to Social Protocols: Software Engineering Foundations for Sociotechnical Systems”. In: **Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016**. Ed. by Jacqueline Bourdeau et al. ACM, 2016, pp. 903–914 (cit. on pp. 17, 18).

[37] G. Ciatto et al. “Twenty Years of Coordination Technologies: COORDINATION Contribution to the State of Art”. In: **J. Log. Algebraic Methods Program.** 113 (2020), p. 100531 (cit. on p. 9).

[38] S. Cranen et al. “An Overview of the mCRL2 Toolset and Its Recent Advances”. In: **Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings**. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 199–213 (cit. on pp. 62, 66, 82).

[39] R. Cruz and J. Proença. “ReoLive: Analysing Connectors in Your Browser”. In: **Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers**. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Vol. 11176. Lecture Notes in Computer Science. Springer, 2018, pp. 336–350 (cit. on pp. 66, 82).

[40] V. Danos and J. Krivine. “Transactions in RCCS”. In: **CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings**. Ed. by Martín Abadi and Luca de Alfaro. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 398–412 (cit. on p. 100).

[41] D. Darquennes, J.-M. Jacquet, and I. Linden. “On Density in Coordination Languages”. In: **Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers**. Ed. by Carlos Canal and Massimo Villari. Vol. 393. Communications in Computer and Information Science. Springer, 2013, pp. 189–203 (cit. on p. xx).

[42] D. Darquennes, J.-M. Jacquet, and I. Linden. “On Distributed Density in Tuple-based Coordination Languages”. In: **Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014**. Ed. by Javier Cámara and José Proença. Vol. 175. EPTCS. 2014, pp. 36–53 (cit. on p. xx).

[43] D. Darquennes, J-M. Jacquet, and I. Linden. “On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study”. In: **Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings**. Ed. by Giovanna Di Marzo Serugendo and Michele Loreti. Vol. 10852. Lecture Notes in Computer Science. Springer, 2018, pp. 81–109 (cit. on pp. xix, xx, 14, 29, 45, 74, 104).

[44] D. Darquennes, J.-M. Jacquet, and I. Linden. “On the Introduction of Density in Tuple-Space Coordination Languages”. In: vol. 115-116. 2016, pp. 149–176 (cit. on p. xx).

[45] R. De Nicola, L. Di Stefano, and O. Inverso. “Multi-Agent Systems with Virtual Stigmergy”. In: **Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers**. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Vol. 11176. Lecture Notes in Computer Science. Springer, 2018, pp. 351–366 (cit. on p. 83).

[46] J-P Denis. “Conception d’un Système de Controle du Développement du Groupe: Architecture et Principes Ingénieriques”. PhD thesis. Lyon 3, 2000 (cit. on p. 17).

[47] E. W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: **Commun. ACM** 18.8 (1975), pp. 453–457 (cit. on p. 99).

[48] M. Dukielska and J. Sroka. “JavaSpaces NetBeans: A Linda Workbench for Distributed Programming course”. In: **Proceedings of the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2010, Bilkent, Ankara, Turkey, June 26-30, 2010**. Ed. by Reyyan Ayfer, John Impagliazzo, and Cary Laxer. ACM, 2010, pp. 23–27 (cit. on pp. 67, 83).

[49] L. D. Erman et al. “The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty”. In: **ACM Comput. Surv.** 12.2 (1980), pp. 213–253 (cit. on p. 58).

[50] M. Felleisen. “On the Expressive Power of Programming Languages”. In: **ESOP’90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings**. Ed. by Neil D. Jones. Vol. 432. Lecture Notes in Computer Science. Springer, 1990, pp. 134–151 (cit. on p. 104).

[51] C-L. Fok, G-C. Roman, and G. Hackmann. “A Lightweight Coordination Middleware for Mobile Computing”. In: **Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings**. Ed. by Rocco De Nicola, Gian-Luigi Ferrari, and Greg Meredith. Vol. 2949. Lecture Notes in Computer Science. Springer, 2004, pp. 135–151 (cit. on p. 11).

[52] M. Fukuda et al. “Intra-and Inter-Object Coordination with MESSENGERS”. In: **International Conference on Coordination Languages and Models**. Springer. 1996, pp. 179–196 (cit. on p. 10).

[53] D. Gelernter. “Generative Communication in Linda”. In: **ACM Trans. Program. Lang. Syst.** 7.1 (1985), pp. 80–112 (cit. on pp. 3, 4, 6–8).

[54] D. Gelernter and N. Carriero. “Coordination Languages and Their Significance”. In: **Commun. ACM** 35.2 (1992), pp. 96–107 (cit. on pp. xx, 6, 8, 9, 43).

[55] P. Godefroid and P. Wolper. “Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties”. In: **Computer Aided Verification, 3rd International Workshop, CAV ’91, Aalborg, Denmark, July, 1-4, 1991, Proceedings**. Ed. by Kim Guldstrand Larsen and Arne Skou. Vol. 575. Lecture Notes in Computer Science. Springer, 1991, pp. 332–342 (cit. on p. 100).

[56] M. Goold and A. Campbell. **Strategies and Styles: The Role of the Centre in Managing Diversified Corporations**. Basil Blackwell, 1987 (cit. on p. 17).

[57] R. Gorrieri, S. Marchetti, and U. Montanari. “A2CCS: Atomic Actions for CCS”. In: **Theoretical Computer Science** 72.2&3 (1990), pp. 203–223 (cit. on p. 100).

[58] P-P. Grasse. “La Reconstruction du Nid et les Coordinations Interindividuelles chez Bellicositermes Natalensis et Cubitermes sp. la Théorie de la Stigmergie: Essai d’Interprétation du Comportement des Termites Constructeurs”. In: **Insectes sociaux** 6.1 (1959), pp. 41–80 (cit. on p. 15).

[59] S. G. Gray. “The Tavistock Institute of Human Relations”. In: **Fifty Years of the Tavistock Clinic (Psychology Revivals)**. Routledge, 2014, pp. 206–227 (cit. on p. xx).

[60] D. Harel, S. Efroni, and I.R. Cohen. “Reactive Animation”. In: **Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures**. Ed. by Frank S. de Boer et al. Vol. 2852. Lecture Notes in Computer Science. Springer, 2002, pp. 136–153 (cit. on p. 83).

[61] **Institute McKinsey Global**. "https://www.mckinsey.com/". 2018 (cit. on p. 120).

[62] J-M. Jacquet and M. Barkallah. “Anemone: A Workbench for the Multi-Bach Coordination Language”. In: **Sci. Comput. Program.** 202 (2021), p. 102579 (cit. on pp. xxv, 70, 79, 101, 115).

[63] J-M. Jacquet and M. Barkallah. “Scan: A Simple Coordination Workbench”. In: **Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings**. Ed. by Hanne Riis Nielson and Emilio Tuosto. Vol. 11533. Lecture Notes in Com-

puter Science. Springer, 2019, pp. 75–91 (cit. on pp. xxiv, xxv, 44, 46, 69, 83, 101, 115, 167).

[64] J.-M. Jacquet, K. De Bosschere, and A. Brogi. "On Timed Coordination Languages". In: **Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Limassol, Cyprus, September 11-13, 2000, Proceedings**. Ed. by António Porto and Gruia-Catalin Roman. Vol. 1906. Lecture Notes in Computer Science. Springer, 2000, pp. 81–98 (cit. on p. xx).

[65] J-M. Jacquet and I. Linden. "Coordinating Context-Aware Applications in Mobile Ad-Hoc Networks". In: **Proceedings of the first ERCIM workshop on eMobility**. The University of Bern. 2007, pp. 107–118 (cit. on pp. 12, 14, 21, 45).

[66] J.-M. Jacquet and I. Linden. "Fully Abstract Models and Refinements as Tools to Compare Agents in Timed Coordination Languages". In: **Theor. Comput. Sci.** 410.2-3 (2009), pp. 221–253 (cit. on p. xx).

[67] J-M. Jacquet, I. Linden, and M-O. Staicu. "Blackboard Rules: From a Declarative Reading to its Application for Coordinating Context-Aware Applications in Mobile ad-Hoc Networks". In: **Sci. Comput. Program.** 115-116 (2016), pp. 79–99 (cit. on pp. 16, 74).

[68] S.-S.T.Q. Jongmans et al. "Automatic Code Generation for the Orchestration of Web Services with Reo". In: **Service-Oriented and Cloud Computing - First European Conference, ESOCC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings**. Ed. by Flavio De Paoli, Ernesto Pimentel, and Gianluigi Zavattaro. Vol. 7592. Lecture Notes in Computer Science. Springer, 2012, pp. 1–16 (cit. on pp. xix, 14).

[69] N. Kokash and F. Arbab. "Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools". In: **IEEE Trans. Serv. Comput.** 6.2 (2013), pp. 186–200 (cit. on pp. 66, 82).

[70] E. Kühn, S.T. Radschek, and N. Elaraby. "Distributed Coordination Runtime Assertions for the Peer Model". In: **Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings**. Ed. by Giovanna Di Marzo Serugendo and Michele Loreti. Vol. 10852. Lecture Notes in Computer Science. Springer, 2018, pp. 200–219 (cit. on pp. 67, 83).

[71] L. Ladenberger, J. Bendisposto, and M. Leuschel. "Visualising Event-B Models with B-Motion Studio". In: **Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings**. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Vol. 5825. Lecture Notes in Computer Science. Springer, 2009, pp. 202–204 (cit. on p. 83).

[72] L. Ladenberger and M. Leuschel. "BMotionWeb: A Tool for Rapid Creation of Formal Prototypes". In: **Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings**. Ed. by Rocco De Nicola and Eva Kühn. Vol. 9763. Lecture Notes in Computer Science. Springer, 2016, pp. 403–417 (cit. on p. 83).

[73] E. A. Lee. "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models". In: **Sensors** 15.3 (2015), pp. 4837–4869 (cit. on p. 157).

[74] I. Linden and J.-M. Jacquet. "On the Expressiveness of Absolute-Time Coordination Languages". In: **Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings**. Ed. by Rocco De Nicola, Gian-Luigi Ferrari, and Greg Meredith. Vol. 2949. Lecture Notes in Computer Science. Springer, 2004, pp. 232–247 (cit. on pp. xx, 104).

[75] I. Linden and J.-M. Jacquet. "On the Expressiveness of Timed Coordination via Shared Dataspaces". In: Electronic Notes in Theoretical Computer Science 180.2 (2004). Ed. by Antonio Brogi, Jean-Marie Jacquet, and Ernesto Pimentel, pp. 71–89 (cit. on pp. xx, 104).

[76] I. Linden et al. "On the Expressiveness of Relative-Timed Coordination Models". In: Electronic Notes in Theoretical Computer Science 97 (2003). Ed. by Antonio Brogi, Jean-Marie Jacquet, and Ernesto Pimentel, pp. 125–153 (cit. on pp. xx, 104).

[77] M. Lippi et al. "Coordinating Distributed Speaking Objects". In: **37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017**. Ed. by Kisung Lee and Ling Liu. IEEE Computer Society, 2017, pp. 1949–1960 (cit. on p. 19).

[78] M. Louvel and F. Pacull. "LINC: A Compact Yet Powerful Coordination Environment". In: **Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings**. Ed. by Eva Kühn and Rosario Pugliese. Vol. 8459. Lecture Notes in Computer Science. Springer, 2014, pp. 83–98 (cit. on p. 9).

[79] S. Mariani. "Coordination in Socio-Technical Systems: Where are we Now? Where do we go Next?" In: **Sci. Comput. Program.** 184 (2019) (cit. on pp. 15–17, 19).

[80] S. Mariani. "Coordination of Complex Socio-Technical Systems: Challenges and Opportunities". In: **Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers**. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Vol. 11176. Lecture Notes in Computer Science. Springer, 2018, pp. 295–310 (cit. on pp. xix, 16).

[81] S. Mariani. **Coordination of Complex Sociotechnical Systems - Self-organisation of Knowledge in MoK**. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016 (cit. on pp. xix, 14, 17).

[82] S. Mariani and A. Omicini. "Molecules of Knowledge: Self-Organisation in Knowledge-Intensive Environments". In: **Intelligent Distributed Computing VI - Proceedings of the 6th International Symposium on Intelligent Distributed Computing - IDC 2012, Calabria, Italy, September 2012**. Ed. by Giancarlo Fortino et al. Vol. 446. Studies in Computational Intelligence. Springer, 2012, pp. 17–22 (cit. on p. 16).

[83] P. Matiello and A.C. Vieira de Melo. "PiStache: Implementing π-Calculus in Scala". In: **Formal Methods, Foundations and Applications - 14th Brazilian Symposium, SBMF 2011, São Paulo, Brazil, September 26-30, 2011, Revised Selected Papers**. Ed. by Adenilso da Silva Simão and Carroll Morgan. Vol. 7021. Lecture Notes in Computer Science. Springer, 2011, pp. 76–91 (cit. on p. 32).

[84] K.L. McMillan. "Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits". In: **Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings**. Ed. by Gregor von Bochmann and David K. Probst. Vol. 663. Lecture Notes in Computer Science. Springer, 1992, pp. 164–177 (cit. on p. 100).

[85] L Medsker. **"Algorithmic Transparency and Accountability–AI Matters**. 2017 (cit. on p. 19).

[86] D. Méry and N.K. Singh. "Real-Time Animation for Formal Specification". In: **Complex Systems Design & Management - Proceedings of the First International Conference on Complex System Design & Management, CSDM 2010, Paris, France, October 27-29, 2010**. Ed. by Marc Aiguier, Francis Bretaudeau, and Daniel Krob. Springer, 2010, pp. 49–60 (cit. on p. 83).

[87] R. De Nicola, G-L. Ferrari, and R. Pugliese. "KLAIM: A Kernel Language for Agents Interaction and Mobility". In: **IEEE Trans. Software Eng.** 24.5 (1998), pp. 315–330 (cit. on p. 10).

[88] R. De Nicola, L. Di Stefano, and O. Inverso. "Multi-Agent Systems with Virtual Stigmergy". In: **Sci. Comput. Program.** 187 (2020), p. 102345 (cit. on p. 15).

[89] D. C. North. **Institutions, Institutional Change and Economic Performance**. Cambridge university press, 1990 (cit. on p. 15).

[90] M. Odersky, L. Spoon, and B. Venners. **Programming in Scala, A Comprehensive Step-by-Step Guide**. Artemis, 2016 (cit. on pp. 61, 62).

[91] A. Omicini and E. Denti. "From Tuple Spaces to Tuple Centres". In: **Sci. Comput. Program.** 41.3 (2001), pp. 277–294 (cit. on p. 12).

[92] A. Omicini and F. Zambonelli. “TuCSoN: a Coordination Model for Mobile Information Agents”. In: **Proceedings of the 1st Workshop on Innovative Internet Information Systems**. Vol. 138. 1998, p. 24 (cit. on pp. 8, 11).

[93] A. Omicini et al. “Integrating Objective & Subjective Coordination in FIPA: A Roadmap to TuCSoN”. In: **WOA 2003: Dagli Oggetti agli Agenti. 4th AI*IA/TABOO Joint Workshop "From Objects to Agents": Intelligent Systems and Pervasive Computing, 10-11 September 2003, Villasimius, CA, Italy**. Ed. by Giuliano Armano et al. Pitagora Editrice Bologna, 2003, pp. 85–91 (cit. on pp. 67, 83).

[94] D. Ouardi, M. Barkallah, and J-M. Jacquet. “The B2Scala Tool: Integrating Bach in Scala with Security in Mind”. In: **Proceedings 17th Interaction and Concurrency Experience, ICE 2024, Groningen, The Netherlands, 21st June 2024**. Ed. by Clément Aubert et al. Vol. 414. EPTCS. 2024, pp. 58–76 (cit. on pp. xxiv, 157).

[95] G. A. Papadopoulos and F. Arbab. “Coordination Models and Languages”. In: **Adv. Comput.** 46 (1998), pp. 329–400 (cit. on pp. xx, 8, 9, 58, 79).

[96] H. Van Dyke Parunak. “A Survey of Environments and Mechanisms for Human-Human Stigmergy”. In: **Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers**. Ed. by D. Weyns, H. Van Dyke Parunak, and F. Michel. Vol. 3830. Lecture Notes in Computer Science. Springer, 2005, pp. 163–186 (cit. on p. 15).

[97] D.A. Peled. “All from One, One for All: on Model Checking Using Representatives”. In: **Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings**. Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423 (cit. on p. 100).

[98] K. Peters. “Comparing Process Calculi Using Encodings”. In: **Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019**. Ed. by Jorge A. Pérez and Jurriaan Rot. Vol. 300. EPTCS. 2019, pp. 19–38 (cit. on p. 116).

[99] C. Pinciroli and G. Beltrame. “Buzz: An Extensible Programming Language for Heterogeneous Swarm Robotics”. In: **2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016**. IEEE, 2016, pp. 3794–3800 (cit. on p. 15).

[100] R. Rajkumar et al. “Cyber-Physical Systems: The Next Computing Revolution”. In: **Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010**. Ed. by Sachin S. Sapatnekar. ACM, 2010, pp. 731–736 (cit. on p. 157).

[101] C. Reas and B. Fry. **Processing: A Programming Handbook for Visual Designers**. The MIT Press, 2014 (cit. on p. 61).

[102] E. Rescorla. **The Transport Layer Security (TLS) Protocol Version 1.3**. Tech. rep. 2018, pp. 1–160 (cit. on p. 157).

[103] C. W. Reynolds. “Flocks, Herds and Schools: A Distributed Behavioral Model”. In: **Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987**. Ed. by Maureen C. Stone. ACM, 1987, pp. 25–34 (cit. on p. 16).

[104] M. Reynolds. “A Traditional Tree-Style Tableau for LTL”. In: **CoRR** abs/1604.03962 (2016) (cit. on p. 65).

[105] M. T. Ribeiro, S. Singh, and C. Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016**. Ed. by Balaji Krishnapuram et al. ACM, 2016, pp. 1135–1144 (cit. on p. 156).

[106] A. Ricci et al. “Cognitive Stigmergy: Towards a Framework Based on Agents and Artifacts”. In: **Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers**. Ed. by Danny Weyns, H. Van Dyke Parunak, and Fabien Michel. Vol. 4389. Lecture Notes in Computer Science. Springer, 2006, pp. 124–140 (cit. on p. 15).

[107] A. Ricci et al. “Environment Programming in CArtAgO”. In: **Multi-Agent Programming, Languages, Tools and Applications**. Ed. by Rafael H. Bordini et al. Springer, 2009, pp. 259–288 (cit. on p. 19).

[108] S. Russell and P. Norvig. **Artificial Intelligence: A Modern Approach (4th Edition)**. Pearson, 2020 (cit. on p. 96).

[109] M. Jeong S. Min K. K. Fung So. “Consumer Adoption of the Uber Mobile Application: Insights from Diffusion of Innovation Theory and Technology Acceptance Model”. In: **Journal of Travel & Tourism Marketing** 36.7 (2019), pp. 770–783 (cit. on p. 4).

[110] V. A. Saraswat and M. Rinard. “Concurrent Constraint Programming”. In: **Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**. 1989, pp. 232–245 (cit. on p. 4).

[111] V. A. Saraswat and M. Rinard. “Concurrent Constraint Programming”. In: **Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990**. Ed. by Frances E. Allen. ACM Press, 1990, pp. 232–245 (cit. on p. 6).

[112] T. Servat. “BRAMA: A New Graphic Animation Tool for B Models”. In: **B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings**. Ed. by Jacques Julliand and Olga Kouchnarenko. Vol. 4355. Lecture Notes in Computer Science. Springer, 2007, pp. 274–276 (cit. on p. 83).

[113] E.Y. Shapiro. “Embeddings Among Concurrent Programming Languages (Preliminary Version)”. In: **CONCUR ’92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings**. Ed. by Rance Cleaveland. Vol. 630. Lecture Notes in Computer Science. Springer, 1992, pp. 486–503 (cit. on p. 105).

[114] W. Shi et al. “Edge Computing: Vision and Challenges”. In: **IEEE Internet Things J.** 3.5 (2016), pp. 637–646 (cit. on p. 156).

[115] R. S. Sutton and A. G. Barto. “Reinforcement learning - An Introduction, 2nd Edition”. In: (2018) (cit. on p. 156).

[116] R. Tolksdorf. “Laura - A Service-Based Coordination Language”. In: **Sci. Comput. Program.** 31.2-3 (1998), pp. 359–381 (cit. on pp. xix, 14).

[117] TomTom. **TomTom Traffic Index**. https://www.tomtom.com/traffic-index/ranking/. 2024 (cit. on p. 120).

[118] E. Trist. “ The Evolution of Socio-Technical Systems: a Conceptual Framework and an Action Research Program”. In: **Ontario Ministry of Government Services** (1981) (cit. on pp. xix, xxi, 14).

[119] K. Ueda. “Guarded Horn Clauses”. In: **Logic Programming ’85, Proceedings of the 4th Conference, Tokyo, Japan, July 1-3, 1985**. Ed. by Eiiti Wada. Vol. 221. Lecture Notes in Computer Science. Springer, 1985, pp. 168–179 (cit. on p. 99).

[120] A. Valmari. “The State Explosion Problem”. In: **Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996**. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. Lecture Notes in Computer Science. Springer, 1996, pp. 429–528 (cit. on p. 100).

[121] H.T. Van et al. “Goal-Oriented Requirements Animation”. In: **12th IEEE International Conference on Requirements Engineering (RE 2004), 6-10 September 2004, Kyoto, Japan**. IEEE Computer Society, 2004, pp. 218–228 (cit. on p. 83).

[122] F. H. Van Eemeren et al. **Fundamentals of Argumentation Theory: A Handbook of Historical Backgrounds and Contemporary Developments**. Routledge, 2013 (cit. on pp. 16, 19).

[123] C. A. Varela and G. Agha. “A Hierarchical Model for Coordination of Concurrent Activities”. In: **Coordination Languages and Models, Third International Conference, COORDINATION ’99, Amsterdam, The Netherlands, April 26-28, 1999, Proceedings**. Ed. by Paolo Ciancarini and Alexander L. Wolf. Vol. 1594. Lecture Notes in Computer Science. Springer, 1999, pp. 166–182 (cit. on p. 10).

[124] R. Virding, C. Wikström, and M. Williams. **Concurrent Programming in ERLANG**. Prentice Hall International (UK) Ltd., 1996 (cit. on pp. xx, 4).

[125] M. Westergaard and K.B. Lassen. “The BRITNeY Suite Animation Tool”. In: **Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings**. Ed. by Susanna Donatelli and P. S. Thiagarajan. Vol. 4024. Lecture Notes in Computer Science. Springer, 2006, pp. 431–440 (cit. on p. 83).

[126] B. Whitworth. “Socio-Technical Systems”. In: **Encyclopedia of Human Computer Interaction** (2006), pp. 533–541 (cit. on p. xix).

[127] D. Wyatt. **Akka Concurrency**. Artima Incorporation, 2013 (cit. on p. 4).

[128] X. Xu et al. “A Taxonomy of Blockchain-Based Systems for Architecture Design”. In: **2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017**. IEEE Computer Society, 2017, pp. 243–252 (cit. on p. 156).

[129] C. Yin et al. “A Literature Survey on Smart Cities”. In: **Sci. China Inf. Sci.** 58.10 (2015), pp. 1–18 (cit. on pp. 119, 120).

[130] F. Zambonelli et al. “Developing Pervasive Multi-Agent Systems with Nature-Inspired Coordination”. In: **Pervasive Mob. Comput.** 17 (2015), pp. 236–252 (cit. on p. 17).