GRAN SASSO SCIENCE INSTITUTE
SCHOOL OF ADVANCED STUDIES
Scuola Universitaria Superiore

DOCTORAL THESIS

Modelling and Verification of Multi-Agent Systems via Sequential Emulation

PHD PROGRAM IN COMPUTER SCIENCE: XXXII CYCLE

Author:

Luca DI STEFANO
luca.distefano@gssi.it

Supervisors:

Prof. Rocco DE NICOLA
rocco.denicola@imtlucca.it

Dr. Omar INVERSO
omar.inverso@gssi.it

October 2020

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Declaration of Authorship

I, Luca Di Stefano, declare that this thesis titled, 'Modelling and Verification of Multi-Agent Systems via Sequential Emulation' and the work presented in it are my own, under the guidance of my supervisors. Parts of Chapter 2 appeared in [80]. Chapter 3 is based on [83]. A preliminary description of the encoding proposed in Chapter 4 appeared in [83]; the current version of the encoding is under submission. Parts of Chapter 5 appeared in [87].

I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this Institute.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: ____

Date: ____

Abstract

A multi-agent system (MAS) is a collection of agents, often endowed with individual goals and a partial view of the whole system, that may be required to achieve complex goals by interacting with each other and with the environment. MASs are a convenient modelling paradigm for complex scenarios across many research fields: they may either be used to describe and reason about existing systems (such as colonies of insects, social networks, economic markets), or to design and assess the correctness of new ones (such as swarms of robots, smart transportation systems).

MASs feature an unprecedented degree of complexity, making their specification and analysis an open problem. This complexity stems from several distinctive features, such as nondeterministic individual behaviour and interactions, asynchronous communication, and a lack of central control. Because of this, formal verification of MASs is particularly challenging. Some existing MAS specification formalisms and platforms lack support for formal verification altogether, and are limited to simulation-based analysis; others focus on specific sub-classes of MASs, or come with tailored verification platforms that might not keep up with the state of the art in formal verification. Meanwhile, formal verification research mainly targets low-level formalisms or traditional programming languages. However, these languages often lack constructs to naturally capture distinctive features of MASs, and thus may be not appropriate for describing them.

To bridge this gap, we put forward a high-level specification language for MASs, where agents operate on (parts of) a decentralised data structure, called a *virtual stigmergy*, which allows to model the influence of local changes on the global behaviour by asynchronously diffusing the knowledge of the agents. This diffusion happens transparently: the user of the language simply defines one or more stigmergic variables and assigns values to them. As a consequence, stigmergies may capture the behaviour of several interesting classes of MASs in a compact and intuitive way. We also introduce a mechanised *sequential emulation* procedure that encodes a high-level system specification into a sequential imperative program. The execution traces of this program emulate the possible evolutions of the input system. Then, the question of whether a property of interest holds for the given system may be answered by performing a certain verification task on the generated program. This allows us to immediately exploit mature verification techniques developed for general-purpose languages, and to effortlessly integrate new techniques as soon as they become available. The procedure is language-agnostic, as it may generate programs in any imperative language with arrays and loops and it may be adapted to support different input formalisms. We show the feasibility of our approach by implementing a tool that applies this procedure to our original language. A thorough experimental evaluation shows that we can formally verify a selection of real-world systems using off-the-shelf program verification tools.

Acknowledgements

This work would not have been possible without the support of many people.

I would like to thank Rocco De Nicola, who first introduced me to the area of formal methods and then gave me the opportunity of working under his supervision; and Omar Inverso, for teaching me so much about formal verification. Working under their supervision has been a honor and a pleasure, and their guidance and inspiration have been very precious to me. I am deeply grateful to all the Computer Science people at GSSI, namely the coordinators of the PhD program, Rocco De Nicola (again), Luca Aceto, and Michele Flammini, and all researchers and post-docs, for their dedication to us students and the great work environment they have fostered over the years.

I would like to thank Radu Mateescu and the other members of the CONVECS team (Hubert Garavel, Frédéric Lang, Gwen Salaün, and Wendelin Serwe) for kindly hosting me in Grenoble for five months. During my stay I had the privilege of working with them and of meeting so many bright and kind young researchers and students: Ajay, Alex, Armen, Lina, Nikita, Philippe, Pierre, Supriya, Umar, and many others. I wish to thank them for the many stimulating conversations and fun moments we shared.

I would also like to thank Radu, together with Lucas Cordeiro, for their kind agreement to review my work and for improving it through their suggestions.

Many thanks to all my dear fellow PhD students at GSSI, with whom I shared so many great moments. Besides their brightness, kindness, and companionship, I am grateful to them for giving this city of mine a living chance. I am especially thankful to Alessandro, Aline, Antonio, Stella, and Xuan, for the mutual support during and after the lockdown.

Lastly, I would like to dedicate this thesis to my family, which has always encouraged my curiosity and showered me in affection. There is no “thank you” big enough for that.

Contents

List of Figures

List of Tables

Abbreviations

AbC	**A**ttribute-**b**ased **C**ommunication
ABM	**A**gent-**B**ased **M**odelling
BDD	**B**inary **D**ecision **D**iagram
BDI	**B**elief-**D**esire-**I**ntention
BMC	**B**ounded **M**odel **C**hecking
CAS	**C**ollective **A**daptive **S**ystem
CEGAR	**C**ount**e**rexample-**G**uided **A**bstraction **R**efinement
DSL	**D**omain-**S**pecific **L**anguage
LTL	**L**inear **T**emporal **L**ogic
LTS	**L**abelled **T**ransition **S**ystem
MAS	**M**ulti-**A**gent **S**ystem
PDR	**P**roperty **D**irected **R**eachability
SAT	Boolean satisfiability
SMT	**S**atisfiability **M**odulo **T**heories
SOS	**S**tructural **O**perational **S**emantics

Symbols

Symbol	*Parameters*	*Meaning*
$\mathbb{N}$		The set of *natural numbers*: $\{0, 1, 2, \dots\}$
$\mathcal{P}(X)$	X: a set	The *power set* of X: $\{Y \mid Y \subseteq X\}$
$\langle x, y, \dots, z\rangle$	$x \in X, y \in Y, \dots, z \in Z$	A *tuple*: an element of $X \times Y \times \cdots \times Z$
$X \hookrightarrow Y$	X, Y: sets	A *partial function* from X to Y
$f[x \mapsto y]$	$f : X \hookrightarrow Y; x \in X; y \in Y$	The partial function f' such that $f'(x') = y$ if $x = x'$ and $f'(x') = f(x')$ otherwise

Chapter 1

Introduction

The study of collective systems is central to several disciplines, ranging from natural sciences such as ecology and biology to sociology, economics and social sciences. At the same time, there exists an ever-growing demand for new paradigms and approaches to the design of man-made collective systems involving a very large number of components deployed over vast geographic areas, such as teams of robots, smart transportation systems and road networks, and so on. This demand is reflected by the appearance of novel terms to classify these systems, such as *complex adaptive systems* [134]; *collective adaptive systems* [129]; *cyber-physical systems* [229]; *swarm robotics* [214]. All these new classifications stress the extreme features and unprecedented complexity of the systems they designate, as well as their need to operate in, and adapt to, the physical world.

However, the more general notion of *multi-agent systems* (MASs) seems to be equally appropriate to classify these systems: they may all be seen as collections of interacting *agents*, within a possibly dynamic and modifiable *environment*, while features stressed by more specific classifications (such as adaptivity or swarm-like behaviour) emerge from the capabilities of the individuals and their mutual interactions. MASs seem to put researchers at ease, as they allow to describe a system of interest starting from its elementary components, i.e., the agents [38]. Thus, the practice of building and analysing *agent-based models* of collective systems is becoming increasingly popular [38, 92, 93, 231]. As a consequence, there is a growing need for *languages* that allow to create these models in a concise and intuitive fashion. These languages should hide the complexity of these systems as much as possible, while remaining general enough to capture a reasonably inclusive range of scenarios. It is worth noticing that the impact of such affordable languages may potentially extend well beyond the computer science community, as they may induce scholars from different disciplines into experimenting with agent-based systems within their area of expertise.

Multi-agent systems, however different, do share a number of common traits. Each agent may follow simple local rules that govern its *actions* (moving, sensing or altering the environment, etc.); at the same time, those actions may intertwine with those of other agents, possibly affecting or being influenced by them. In fact, this intertwining of actions gives rise to an exceptionally large set of possible evolutions, which may be hard to predict even for a small number of agents, and such systems seem to «operate at an intermediate state between order and chaos» [142], as they exhibit the *emergence* of collective, "orderly" behaviours that appear to have little relation to the individual rules followed by the agents and their "chaotic" interaction [21, 169]. Understanding the relation between the individual and the collective would give us a deeper insight into the nature of such systems, and may also lead to the design of better artificial MASs [194]. However, given the aforementioned unpredictability of these systems, this goal can be achieved only with the support of *techniques and tools* that can perform an automated, systematic analysis of their state space and draw reliable conclusions about their behaviour. Even simulation, which is widely used to test hypotheses about agent-based models, can be only marginally effective to this end. In fact, in a number of cases *formal* assurance about the behaviour of a system is required, which is beyond the reach of simulation-based reasoning [245]. Formal verification, instead, may provide such assurance through mathematically founded techniques, but it may struggle with obtaining conclusive results on large systems, since they often feature a combinatorial explosion in the number of states. This is known as the *state explosion problem* [64].

A great deal of formal verification research has been carried out in the context of general-purpose programming languages, such as C: however, using such languages would be out of reach for most researchers lacking a programming background which may be, nevertheless, interested in agent-based modelling. Furthermore, most of the commonly observed features of MASs cannot be easily expressed through usual programming constructs, and the resulting specifications could be difficult to understand [80]. On the other hand, existing platforms providing both a specification language and a related analysis toolbox can be successful for specific classes of systems, but at the cost of limiting the flexibility of the specifications by enforcing rigid templates for the behaviour [207], disallowing value-passing actions [162], restricting the interaction to point-to-point communication [37, 162], assuming that the properties of interest have a specific form and can predicate on the global state only [37], and the like. In general, representing complex systems under such restrictions may be unnatural, and as a result precious information about the structure of these systems may get lost in the modelling phase, compromising the efficiency of the analysis. Additionally, these platforms often implement a specific verification technique, and tailor it to the chosen specification language. This may make it difficult to keep up with the state of the art in formal verification.

Approach

The concerns raised so far call for the introduction of flexible yet rigorous domain-specific formalisms for multi-agent systems. By *flexibility*, we mean that such formalisms should allow users to model a wide range of scenarios and should not tie them to any specific system template. *Rigorousness* means that the language should be supported by a formal semantics, on which automated analysis of complex systems can be solidly grounded. Lastly, by *domain-specificity* [237] we mean that the language should provide appropriate abstractions to deal with multi-agent systems in a concise way, so as to spare users from explicitly describing low-level details about the modelled systems.

The analysis workflow of models written in such formalisms should take advantage of the constant progress in the field of automated formal verification, and should not be tied to a specific approach to systems analysis: it should rather be able to apply multiple different techniques and effortlessly evolve along with the state of the art.

Lastly, such workflows should be implemented as *push-button* analysis tools. While semi-automatic verification procedures, such as those based on interactive theorem proving, do have their advantages [118], we believe that fully automated tools could facilitate the adoption of state-of-the-art techniques for computer-aided verification among diverse research communities. These tools would put the end-user at ease, as familiarity with the domain specific formalism would be the only usage requirement.

This thesis aims at describing a possible way to fulfil these requirements.

Contributions

The main contributions of this work may be summarised as follows:

1. a domain-specific, formal language for the specification of multi-agent systems (Chapter 3);
2. a compact structure-preserving encoding of formal specifications into sequential imperative programs (Chapter 4);
3. a prototype tool that encodes our language into C or LNT [103] programs (Chapter 5).

Formal domain-specific language

In Chapter 3 we introduce a domain-specific language built around a core calculus called LAbS [83]. The design of LAbS was influenced by a study of several existing languages for multi-agent and multi-robot systems [80]. In particular, it allows to model agent-to-agent interaction through *virtual stigmergies*, i.e., decentralised data structures that allow a group of neighbouring agents to agree on the contents of a key-value store [198]. LAbS formalises the operations of a virtual stigmergy by means of a structural operational semantics. At the same time, it generalises the concept of virtual stigmergies in several ways. First, it allows the user to customise the notion of *neighbourhood*, which was originally tied to spatial proximity. In our language, instead, neighbourhood between two agents is determined by satisfaction of a user-defined predicate over their state. Moreover, multiple virtual stigmergies may exist within a system, with potentially different predicates. These generalisations increase the flexibility of virtual stigmergies as a modelling tool. To highlight the capabilities of this language, we provide several example specifications that model a selection of real-world multi-agent systems.

Structure-preserving encoding

In Chapter 4 we present a mechanised, semantics-based procedure that translates a specification of a distributed system, together with a property of interest, into a sequential imperative program that *emulates* the given system. By this, we mean that the program reproduces the feasible executions of the system, and thus verifying it can give a verdict on whether the system satisfies the given property. The procedure first builds an intermediate representation that captures the possible flow of action of each agent. The interleaving of actions is not represented explicitly, but rather encoded by a control mechanism based on symbolic expressions. This retains the compactness of the input specification. The obtained intermediate representation is then used to generate a sequential imperative program. Separate functions encode the actions of the agents, and a *scheduler* repeatedly invokes such functions. The symbolic control mechanism described above guarantees that all feasible evolutions are captured while no spurious evolution is introduced. Finally, the program is decorated with appropriate statements encoding the property to verify, so that formal verification of the input system reduces to a verification query on the decorated program. This procedure is semantics-based in the sense that it may be adapted to any input language provided with a structural operational semantics (SOS) [200], thus not limited to the language that we introduce in Chapter 3.

Prototype implementation and experimental evaluation

In Chapter 5 we introduce a prototype tool, called SLiVER, which implements our structure-preserving encoding procedure for our agent-based specification language. The tool is written in F# and Python, currently generates programs in two languages (either the C programming language [141] or the LNT process calculus [103]), and can be used either to verify a system against a given property, or to simulate its evolution up to an arbitrary number of transitions. We describe the workflow of the tool and illustrate its usage through examples. Then, to show that our encoding procedure is an effective approach to multi-agent systems verification, we perform a battery of tasks using a selection of state-of-the-art tools for the verification of C programs. These tools cover a wide range of techniques from the literature, ranging from *symbolic execution* [143] to more recent approaches such as *property directed reachability* [44]. We succeed in verifying both *invariant* properties (i.e., all reachable states satisfy a given formula) and simple *emergent* properties (i.e., all executions lead to a state where the given formula is satisfied), over the full state space of the input systems.

Structure of the thesis

The rest of this document is organised as follows. Chapter 2 defines the notions and the terminology used throughout the thesis, gives a brief overview of the necessary background concepts, and describes a selection of state-of-the art approaches to formal verification. Chapter 3 contains a description of our language for multi-agent systems. First, the core process algebra is introduced; then, the specification language is illustrated by modelling a collection of example multi-agent systems; lastly, a more formal description of the specification language is provided. Chapter 4 describes a semantics-based encoding from formal specifications into imperative programs. Chapter 5 describes a prototype tool that implements the aforementioned encoding procedure for our specification language, and evaluates its effectiveness by means of a battery of verification tasks. Chapter 6 contains some concluding remarks and possible directions for future work.

Chapter 2

Background

This chapter introduces the terminology used throughout the thesis, as well as the needed mathematical background. Moreover, it provides an overview of multi-agent systems and their main sources of complexity, describes existing languages and platforms oriented at these systems, and gives a brief account of techniques and tools for formal program analysis.

2.1 Multi-agent systems

In this section, we consider the systems we are interested in modelling and analysing, and describe their distinctive features. We collectively refer to them as *multi-agent systems*, although the literature refers to them using several other names, including *collective systems*, *distributed systems*, and *process networks*. We believe that these names do refer to the same classes of systems, and prefer the term *multi-agent systems* to stress that they are the outcome of the composition of multiple, autonomous elements (the agents).

2.1.1 Definitions

Intuitively, an *agent* is any entity with *agency*, i.e., the ability to perform *actions* [217]. Theories of action and agency are central to Western thought, and several alternative definitions of what constitutes an action have been put forward. These definitions generally agree that actions are events originating from an entity as the result of some kind of deliberation, or as a reaction to external stimuli; and that they may alter the state of the world surrounding the agent, i.e., its *environment* [244].

Artificial intelligence (AI) is the branch of computer science that deals with the design of artificial agents [213]. As agents are inevitably tied to actions, theories of agency play a relevant role in

AI. The *logical approach* [188] proposes that, given a formal model of the world and of the effect of each possible action, an agent could just plan a sequence of actions toward its goal (which is itself represented as a desired state of the world). Thus, much effort has been spent on developing formal theories of action [7, 152, 175], and planners that could exploit them [8, 95, 186]. Other works have questioned the logical approach and showed that reactive agents (i.e., whose actions are mainly driven by their current perceptions) can still display complex, apparently plan-driven behaviour [4, 5, 48]. It has been argued that the logical approach tries to mimic human intelligence, while these reactive agents take their inspiration from biology [47].

A *multi-agent system* (MAS) is a collection of agents that interact with one another and with their shared environment [248]. Research on multi-agent systems was initially known as *distributed artificial intelligence* (DAI) [147, 176]: in fact, a popular line of research aims at generalising the aforementioned logical approach to include concepts such as communication, collective knowledge, and shared goals [146, 223, 248]. This approach is sometimes labelled as "top-down": advocates of alternative, "bottom-up", methodologies include the *artificial life* [158] and *swarm intelligence* [39] communities.

Agent-based modelling, or individual-based modelling, is the activity of abstracting a real-world scenario as a multi-agent system. An agent-based model is mainly concerned with specifying the individual behaviour of agents. Properties of the system that may be seen as foundational in other kinds of model, such as the presence of patterns, hierarchies, equilibria, etc., are expected to be *emergent* phenomena of the system. They are the result of the interactions among agents, and may in turn influence their future behaviour [38]. The study of natural systems from an agent-based perspective may in turn influence the development of artificial ones, such as swarms of robots [39, 194].

2.1.2 Examples

In this section, we give an informal description of simple multi-agent systems, which we will refer to in the course of this thesis. We also discuss a selection of real-world scenarios that have been analysed through agent-based modelling, in order to show the wide range of applications of this technique. Other works offer further insight on the role of agent-based models in economics [231, 232], epidemiology [91], ecology [113, 114], and sociology [92].

Example 2.1 (Dining philosophers [88])**.** A group of n philosophers is sitting at a round dinner table. A fork lies to the left of each philosopher, and a large bowl of spaghetti stands in the centre of the table (Figure 2.1). In order to eat, a philosopher has to grab both the fork on his left and the one on his right. Of course, only one philosopher may hold a fork at any given time. Thus, a problem may arise when all the philosophers pick the fork at their left at about the same time,

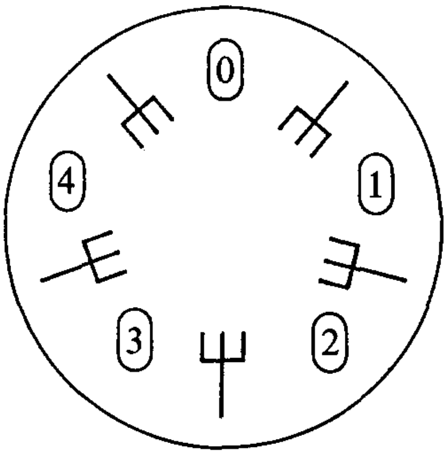


FIGURE 2.1: The *dining philosophers* example, with $n = 5$. Image from the original description of the problem [88].

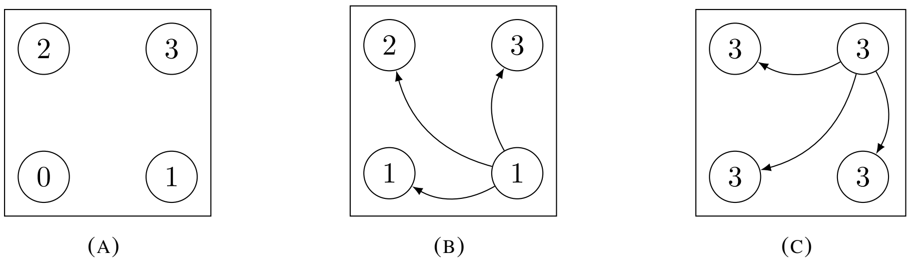
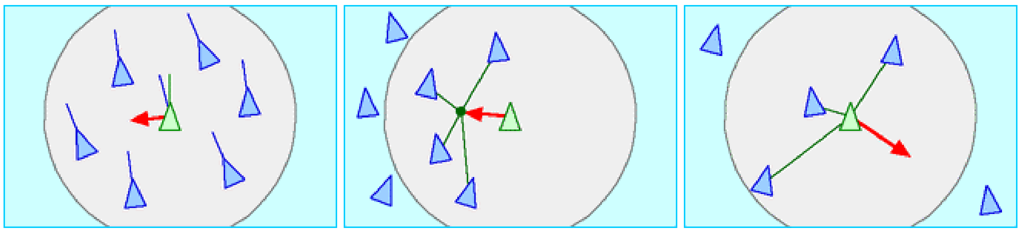


FIGURE 2.2: A possible evolution of a leader election among four agents.

and no one is willing to put down their own fork before eating. «In this undignified situation, they will all inevitably starve» [132].

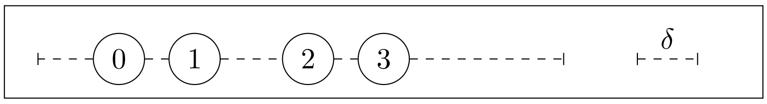
Example 2.2 (Leader election)**.** A leader election mechanism allows a distributed system to choose an agent as its leader, or coordinator [104]. Assuming that each agent in the system has a unique identifier and that messages are always delivered reliably, a basic leader election protocol is the one where each agent repeatedly advertises its own identifier until it receives a message from an agent with a higher one. After some time, all agents will stop sending messages except the one with the highest identifier, which is then selected as leader. Figure 2.2 shows an example evolution of this election protocol when performed by four agents. Initially, each agent considers itself to be the leader (Figure 2.2a). If, say, node 1 advertises itself as leader, it only persuades node 0, while the other nodes keep advertising themselves (Figure 2.2b). Eventually, the node with the highest identifier, i.e., 3, sends a message and gets elected as leader (Figure 2.2c).

Example 2.3 (Flocking)**.** Flocking is the emergent behaviour commonly observed in flocks of birds, schools of fish, swarms of social insects, and other natural systems, and can generally be described as «the collective coherent motion of large numbers of self-propelled organisms» [236]. Such a behaviour may emerge from a set of simple local rules followed by the individual agents. More specifically, the term *Boids* [208] refers to a specific set of such rules, which was originally proposed as part of an efficient algorithm to generate realistic computer animations of flocks

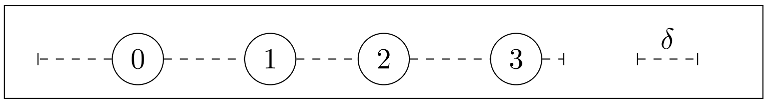


(A) Alignment (B) Cohesion (C) Separation

FIGURE 2.3: Individual rules for *Boids*. Images from the personal web page of Craig Reynolds.[1]



(A) One possible initial state.



(B) An acceptable final state.

FIGURE 2.4: Evolution of a line formation system.

and has been very largely spread since, with countless implementations available. Figure 2.3 contains a graphical representation of these rules, which are: (a) *alignment* (agents head in the same direction as other flockmates nearby); (b) *cohesion* (agents move closer to nearby groups of flockmates); and (c) *separation* (agents avoid collisions with flockmates that are too close).

Example 2.4 (Line formation)**.** This is an elementary example of a pattern-forming system [227]. Agents lie on a one-dimensional segment and they must reach a state where they all maintain a given minimum distance from each other. Consider for instance Figure 2.4a. Segment δ represents the minimum distancing that any two agents should maintain, but every agent has at least one neighbour that is closer than δ. A correct line-forming behaviour should allow the agents to reach a state similar to that of Figure 2.4b, where the minimum distancing requirement is satisfied by all agents.

Example 2.5 (Opinion formation protocols)**.** These protocols assume that each agent has an initial *opinion* and that interactions between agents may lead to changes in their opinions. In the *voter model*, an agent that meets another instantly imitates the opinion of the latter [183]. A *k-unanimity model*, instead, requires an agent to meet k other agents with a given opinion before it switches to that opinion [216]. *Majority protocols*, finally, arrange the interactions of agents in such a way that they eventually agree on the opinion that had the initial majority [13]. As an example, the agents of Figure 2.5 have either a Y- or a N-opinion. Notice that the Y-opinion has the majority in the initial state of the system (Figure 2.5a). Agents randomly wander across the arena. Upon meeting Y-agents, the N-agents change their own opinion to Y (Figure 2.5b). Since the Y-agents were in a majority, eventually all agents have the Y-opinion (Figure 2.5c).

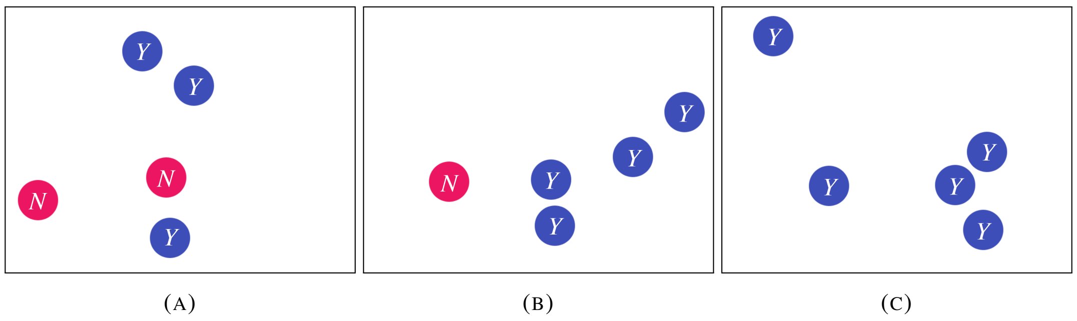


FIGURE 2.5: A possible evolution of a majority protocol. Images adapted from web demo[2] of the Peregrine tool [37].

Real-world scenarios. *Epidemiology* studies the spread of an infection through a community. Social contact patterns are an important factor in the spread of the infection. Agent-based models that are built upon statistical and demographic data can provide good approximations of such patterns and closely fit epidemiological findings from real-world infections [139].

In *archaeology*, conjectures about the social norms of an extinct community can be at least partially validated by showing that an agent-based model built upon these conjectures evolves in a way that matches evidence from archaeological findings [17].

In *economy*, actors such as households, banks, and firms, are often tied by credit relations between one another. These relations induce a *credit network* on the economic system. If one element of this network defaults (i.e., it fails to repay a debt), the topology of the network and the different kinds of entities involved may affect the spread of the resulting financial shock. Agent-based models seem to capture these features more closely than the aggregate approaches of mainstream economics [225].

In *spatial economics*, agents incur in costs that vary according to their current location (e.g., commuting, rent). Agent-based models are a good fit for spatial economics and may provide deep insight into the properties of economic exchanges within this setting [191]. For instance, they may show that *spatial equilibrium*, i.e., the assumption that no agent may unilaterally improve its own status by moving to a better location, is actually an emerging property of spatial economic systems [192].

2.1.3 Features

In this section, we describe a collection of features that are commonly observed in multi-agent systems. As we will see (Section 2.1.4), these traits introduce specific challenges to the modelling and analysis of these systems.

[1]`https://www.red3d.com/cwr/boids/`
[2]`https://peregrine.model.in.tum.de/demo/`

Open-endedness. A system is *open-ended* when agents can join or leave it at any time. This may happen either because of a deliberate choice (e.g., a robot leaving a mission due to a critically low battery level; a social network user deleting his/her account) or as the result of unexpected events, such as hardware failure. Open-endedness complicates specifications and reasoning. Agents that leave the system raise issues similar to those caused by faulty processes within a distributed computing environment [155]. New agents joining the system might also need to acquire information from their peers in order to operate correctly. For instance, a new member of a flock of boids (see Example 2.3) must detect the position and heading of other flockmates in order to imitate them.

Adaptivity. An *adaptive* system can modify its behaviour in response to unexpected changes in the environment, or the system itself [134]. For instance, flocks of birds are known to split in two or more groups if a predator appears, and reassemble after the threat has disappeared [20]. Some species, instead, tend to cluster around the predator: this behaviour is known as *mobbing* [193]. Artificial agents and systems with some degree of adaptivity are quite desirable, as they might be able to operate under unforeseen circumstances with little to no supervision.

Stigmergic interaction. The concept of *stigmergies* originated from the study of the collective behaviour of social insects. Since then, it has been used to describe several classes of collective behaviour in natural and human-made systems [127]. Generally, a stigmergy is an indirect interaction mechanism based on *signs*, or cues, that agents leave in the environment during their activity and that influence the future behaviour of other agents [182]. The first occurrence of the term [111] referred to the nest-building behaviour of some species of termites. These termites build nests by soaking small pellets of soil in pheromone and dropping them on the ground. Other termites detect the scent of pheromone as it evaporates, and tend to accumulate soil pellets where the scent is stronger. This positive feedback leads to the creation of complex structures, such as strips or pillars [233]. Foraging ants also display a form of stigmergic behaviour [86]. Each ant randomly explores the area around their nest until it finds a food source: when that happens, the ant returns to the nest with some food and leaves a pheromone trail on its way back. Other ants follow the trail until they find the food source, and they reinforce that same trail by depositing additional pheromone molecules as they return to the nest. Eventually, most ants find food by trail-following rather than by exploration. This behaviour is also adaptive to changes in the environment. If an established pathway is disrupted by an obstacle, ants wander near the point of disruption until they establish a new trail toward the food. At first different such trails may arise, but in the end the shortest one will be preferred [67].

Numerousness. In some cases, collective features emerge only when the system features a sufficient number of agents. For instance, smaller ant colonies tend to rely on ant-to-ant interaction to spread information about food sources; in larger ones, indirect diffusion of knowledge through chemical trails becomes prevalent [84].

Anonymity. Agents within natural multi-agent systems are mostly anonymous to each other, i.e., they have little to no concept of *identities*. This greatly influences their patterns of cooperation and communication. Cooperation may be guided by the capabilities offered by the agents, or simply by spatial closeness, as in the *Boids* system (Example 2.3).

Decentralisation. A system is decentralised when it lacks central control of any kind. This is usually seen as a desirable property, as decentralised system lack a single point of failure; however, it also means that achieving complex goals may be harder. For instance, the dining philosophers of Example 2.1 may end up starving if they do not coordinate their actions.

Distribution. A multi-agent system is distributed from both a physical and a logical standpoint. By *physical* distribution, we mean that agents generally do not occupy the same real-world location. Rather, they are separate entities, potentially placed far apart from each other, therefore their interactions are exposed to significant delays and potential failures. At the same time, agents may only have limited resources and partial knowledge of the whole system. This results in *logical* distribution, i.e., agents perform a complex task by interacting with each other, rather than leaving it to a single agent. For instance, a majority protocol (Example 2.5) can determine which opinion is held by a majority of agents through a collective, distributed computation.

Asynchrony. The behaviour of agents may be partially or fully asynchronous, i.e., there may be little to no temporal guarantees to local computation and interaction. For instance, agents may take an arbitrarily long time to send a message or react to a request. As a result, it becomes impossible to distinguish an agent that has stopped working from one that is simply taking a long time to respond [154]. A well-known consequence of this is the impossibility of reaching a consensus (e.g., successfully performing a leader election: see Example 2.2) in an asynchronous system with one or more faulty components [96].

Heterogeneity. Many scenarios of interest feature different kinds of agents with different capabilities and goals. For instance, variations of the voter model (Example 2.5) introduce a separate kind of agents, called *zealots*, that never change their own opinion [183].

Environment Dependence. The fact that agents operate within an external environment is a distinguishing feature of multi-agent systems [47, 242]. For instance, the forks that a dining philosopher (Example 2.1) must obtain in order to eat can be seen as part of its environment. The environment is often assumed to be dynamic and partially observable. Due to partial observability, agents may take apparently bad decisions because they seem correct, or even optimal, from their local standpoint. At the same time, agents may have little to no power of altering the environment through their actions. Moreover, acting on the environment may not even result in the expected outcome [247].

Nondeterminism. A multi-agent system features several levels of nondeterminism. At local level, the behaviour of a single agent may be intrinsically nondeterministic. For instance, foraging ants look for food by means of a random walk. Furthermore, an individual behaviour may itself be the result of multiple computations that the agent performs concurrently. For instance, a flocking bird in the *Boids* model (Example 2.3) simultaneously checks direction and position of its flockmates before adjusting its own direction and speed. When we model an agent with such a behaviour, we have to assume that these computations may be freely (i.e., nondeterministically) interleaved, and that different interleavings may lead the agent to behave differently. At global level, the system may have multiple feasible initial states. Additionally, even when the system has only one initial state and each agent has a fully deterministic behaviour, nondeterminism inevitably arises from the fact that agents evolve by randomly alternating their actions. In fact, interaction between agents (or between an agent and the environment) is inherently nondeterministic. For instance, messages broadcast by an agent may be received in different orders by the others; additionally, the communication medium may be unreliable, e.g., it may drop messages or alter their contents.

2.1.4 Modelling challenges

Modelling a multi-agent system is a necessary step in order to analyse it. A popular choice [49, 191, 199] is to reuse traditional, general-purpose programming languages (such as C, C++, or Java) to model a given system as a program, and analyse the evolution of the system by executing such a program. In this section, we look back at the features of Section 2.1.3 and contend that this choice may lead to overcomplicated programs that are hard to maintain and reason about. This, in turn, makes analysis of the modelled system harder and less reliable. The overall issue is that general-purpose languages are mainly concerned with describing computational processes, which often makes them ill-suited for agent-based models. We should stress that these shortcomings do not completely rule out general-purpose languages from multi-agent modelling and analysis. Nevertheless, having to work around these limitations at the specification level is not intuitive, and can lead to increased code complexity and higher chances of introducing programming errors. For instance, the behavioural specifications supporting [199] and [49] amount to about 2100 and 1700 lines of C++ code, respectively.[3] As for code complexity, the two codebases contained respectively 7 and 13 methods with a cyclomatic complexity [172] higher than 10, which is usually seen as a reasonable upper bound beyond which testing and maintaining code becomes difficult [173]; both feature at least one method with cyclomatic complexity higher than 50. In general, such lack of naturalness severely affects the chances of exploiting systems' structure, raising issues about the efficiency of the analysis. At the same time, it discourages scholars from different disciplines to experiment with agent-based models within their area of expertise.

[3]The two codebases are available at `http://dx.doi.org/10.5258/SOTON/385724` and `https://github.com/edgarbuchanan/dps`, respectively.

Open-endedness. Modelling an agent leaving a system is essentially equivalent to preventing it from changing the program state. In fact, there is no way to distinguish a failed process (or even a non-existing one) from one that never performs any action [154]. Similarly, one could model a fixed number of agents that join the system by disabling their models in the initial state and enabling them in a later stage of the program execution. If agents have the ability to *spawn* new ones, or if an arbitrary number of agents entering the system has to be modelled, then the program must necessarily rely on dynamic memory allocation to accommodate the state of these agents. This may overcomplicate formal analysis of the program.

Adaptivity. Modelling adaptive systems is known to be difficult. An adaptive agent, for instance, may alter its behaviour in significant and unpredictable ways as it interacts with other agents and with the environment. However, a model of such an agent is constrained by the limits of the modelling formalism itself. To properly model truly unpredictable adaptivity, such formalism should allow the model itself to change in ways that were not considered in the original specification. The field of artificial life [158], for instance, attempts to mimic *species* that can undergo evolutionary processes by means of models that can randomly mutate [221]. Such features, however, are hard to replicate in the context of traditional programming languages, induce a vast state space, and seem hardly amenable to formal verification.

Stigmergic interaction. Several traditional programming languages completely lack communication primitives, and are thus ill-suited to concisely describe any kind of interaction. Stigmergic interaction poses additional challenges, due to its indirect nature and the fact that messages should linger in the system waiting to be picked up by agents. The *blackboard architecture* [94] shares a number of similarities with stigmergies. Processes in a blackboard system, rather than messaging one another, cooperate indirectly by reading and writing data on a shared memory space.

Numerouseness. As stated above, agents in a MAS evolve concurrently by randomly alternating their actions. As a consequence, the number of feasible evolutions of a system increases exponentially in the number of its agents. Thus, modelling large systems may result in programs with a large state space, which may put a high stress on the underlying runtime environment and on the data structures of the chosen language.

Furthermore, a large state space is detrimental to verification, and even simulation may become harder due to it. In some cases, *counter abstractions* [171, 202] may be an effective solution. Intuitively, a counter abstraction of a system is a model containing a counter for each possible state of an agent. An agent that transits from state s to s' is simply modelled by decreasing the counter of s and increasing that of s'. Extensions of this approach are able to represent unbounded-size systems with finite models. As a drawback, the requirement of having a separate counter for each state may make modelling overcomplicate or even unfeasible when the agents have complex behaviour.

Anonymity. Unlike natural collective systems, anonymity is not a typical trait in computer systems. In fact, many existing programming languages and runtime environments assume that each component has a unique identifier and may interact with others via point-to-point communication. However, abstraction layers that provide some degree of anonymity have proven to be desirable, since this kind of point-to-point interaction may negatively affect the reliability of complex software systems [34]. The already-mentioned blackboard architecture, for instance, completely replaces message passing with operations on the shared memory space: since all communication is mediated by the blackboard, interaction is inherently anonymous. Other approaches, instead, still rely on message-passing primitives. *Group-oriented communication* allows a process to send messages to a group of recipients, rather than a single one [35]. The *publisher-subscribe* pattern (pub-sub) is a popular example of group-oriented communication. Processes *publish* messages that are decorated with a topic identifier, and may *subscribe* to one or more topics. When a message is published, all processes that are subscribed to the corresponding topic receive it [34]. Such a system is implemented, for instance, by the ROS robotics middleware [204]. A similar concept is that of *channels*. A process may use a channel to *send* a value, or it may wait until it *receives* a value over a channel. Multiple processes may share one or more channels: thus, indirect and anonymous communication may be achieved. While in group-oriented architectures a sent value is received by *all* processes in the group, a value sent over a channel is received by a single waiting process, which is selected nondeterministically. Channels originated in process algebras such as CSP [132] and the π-calculus [215] (where channel *names* themselves may be communicated across channels), and are implemented for instance in the Go programming language. *Attribute-based communication* is a generalisation of group-oriented communication, in which groups of agents are dynamically formed or dissolved according to their exposed features (*attributes*). Messages may be addressed to all agents that satisfy a given predicate over these features. Recipients may also filter incoming messages according to a predicate over the sender's attributes [2].

Decentralisation and Distribution. Some widely used traditional languages, such as C, C++, and Java, lack proper constructs to model decentralised and distributed computing. Users of these languages either need to develop their own abstractions or use existing libraries. The former option is costly and error-prone; the latter requires adapting the semantics of the library to that of the system to model. The blackboard architecture was originally seen as a centralised data store that a set of distributed worker processes could use to coordinate their evolution. Decentralised blackboards have also been proposed [51], but guaranteeing atomicity and consistency in such a setting is a real challenge. The aforementioned generalisations of message passing, i.e., group-oriented communication, publish-subscribe, and attribute-based communication, also help in dealing with distributed systems.

Asynchrony. Many traditional languages rely on a sequential model of computation, which makes them ill-suited to model systems that feature asynchronous operations, such as message passing with arbitrary delays. Other languages, such as Erlang [15], do support asynchronous computation and interaction, but these features lead to increased complexity in the model, and make specification and analysis harder. For instance, agents may need to take interaction delays into account when they try to cooperate; pending messages may have to be modelled as separate entities; and so on.

Heterogeneity. In principle, a formalism that only supports homogeneous behaviour may still be able to express a heterogeneous one, if it provides adequate control-flow statements. This approach can, however, greatly increase the complexity of the resulting specifications. Therefore, very heterogeneous systems would quickly become hard to understand and maintain. Due to the complex control-flow structure, tractability of analysis would be affected as well.

Environment. In principle, it is possible to model the interaction between an agent and its surrounding environment by altering the program state appropriately. However, if such an interaction represents the manipulation of a physical space rather than an exchange of data, it would require additional guarantees of synchrony, atomicity and consistency.

Nondeterminism. Traditional programming languages are either deterministic, or they feature a different kind of nondeterminism than the one present in multi-agent systems. For instance, the *evaluation order* of a C expression is unspecified by the language standard: a program containing an expression `f() + g()` may first evaluate `f()` and then `g()`, or vice versa. Potentially, a different order may be chosen for each evaluation of the same expression. Another instance of a nondeterministic construct is *channel selection* in Go. A Go program may wait on multiple channel operations. If two or more such operations are enabled at the same time, the program will select and perform only one of them. This selection happens nondeterministically. In any case, this mismatch poses an additional challenge: to accurately model a MAS, one has to take into account all of its sources of nondeterminism, and replicate them with appropriate language constructs; at the same time, one has to verify that nondeterminism introduced by the language does not affect the behaviour of the model (say, by introducing spurious evolutions).

2.2 Property specification

In this section, we introduce some logic formalisms that are commonly used to specify the desired properties of a system under analysis.

2.2.1 Linear Temporal Logic

A *temporal logic* allows to attach time-related information to a logic predicate, thus allowing to formalise statements such as “p holds at all times” or “if p holds now, then q will be true sometime in the future" [109]. Temporal logics are often used to reason about the correctness of programs. Correctness is typically defined by a set of properties that all executions of the program must satisfy. These properties belong to two categories: *safety* properties (meaning that undesired events never happen) and *liveness* properties (meaning that some desired event eventually happens) [153]. Temporal logics provide a natural way to formalise these properties.

A popular example of a temporal logic formalism is Linear Temporal Logic (LTL) [201], whose syntax is shown in Table 2.1a. In LTL, time is a discrete sequence of instants, while formulas are constructed from a set of *atomic propositions*, i.e., facts that may or may not hold at any instant in time. So, the LTL formula a (where a is an atomic proposition) means "a is true in this instant". $\neg a$ ("a is not true in this instant") and $a \wedge b$ ("both a and b hold in this instant") are also valid LTL formulas. One can then introduce the contradiction $\bot \triangleq a \wedge \neg a$ (for some atomic proposition a) and its negation $\top \triangleq \neg\bot$ (which holds in all instants). LTL provides two temporal operators, called *next* ($\bigcirc$) and *until* (U). The formula $\bigcirc a$ means "a will be true in the *next* instant". while $p \mathbin{\mathsf{U}} q$ means "q holds at some instant in the future, and p holds *until then*". More complex operators could be defined in terms of U. For instance, $\Diamond\varphi$ ("eventually φ"), meaning that φ should hold at some instant, is equivalent to $\top \mathbin{\mathsf{U}} \varphi$. On the other hand, $\Box\varphi$ ("always φ") means that φ should hold at every instant, and can be expressed as $\neg\Diamond\neg\varphi$.

Let us consider the *dining philosophers* of Example 2.1, and let us define for each philosopher an atomic proposition $wait_i$ that holds if and only if the i-th philosopher is waiting for the second fork. Then, the LTL formula $\Box\neg(wait_1 \wedge wait_2 \wedge \cdots \wedge wait_n)$ specifies that there should always be at least one philosopher who is *not* waiting for the second fork. If this formula is violated, then there exists a state where no philosopher can perform any further action and the system is deadlocked.

As another example, let us consider the *majority protocols* introduced in Example 2.5. A majority protocol is correct if, eventually, all n agents agree on the opinion that was initially held by a majority of agents. Let us assume that the opinion of each agent is either *Yes* or *No*. Then, we can define for each $i = 1, 2, \ldots, n$ an atomic proposition Yes_i that holds if and only if the i-th agent has opinion *Yes*. We further define a proposition *MajorityYes* that holds if and only if at least $n/2$ agents have opinion *Yes*.[4] This allows us to formalise a necessary condition for correctness, namely that if *Yes* is the majority opinion in the initial state, then at some point all agents should have opinion *Yes*: this is specified in LTL as $\mathit{MajorityYes} \Rightarrow \Diamond(Yes_1 \wedge Yes_2 \wedge \cdots \wedge Yes_n)$.

Kripke structures. LTL properties are typically evaluated over a *Kripke structure* that represents the system of interest. Let AP be a set of atomic propositions. A Kripke structure is a tuple $\langle S, I, R, L\rangle$ where S is a finite set of *states*, $I \subseteq S$ a set of initial states, $R \subseteq S \times S$ a

[4]This proposition is a first-order predicate over the atomic propositions $Yes_1, Yes_2, \ldots, Yes_n$. For instance, for $n = 3$ we have $\mathit{MajorityYes} \triangleq (Yes_1 \wedge Yes_2) \vee (Yes_2 \wedge Yes_3) \vee (Yes_1 \wedge Yes_3)$.

TABLE 2.1: Linear Temporal Logic.

(A) Grammar.

$$
\begin{array}{rcl}
\varphi & ::= & a \\
& | & \neg\varphi \\
& | & \varphi_1 \wedge \varphi_2 \\
& | & \bigcirc\varphi \\
& | & \varphi_1 \mathbin{\mathsf{U}} \varphi_2
\end{array}
$$

(B) Formal semantics.

$$
\begin{array}{rclcl}
\sigma & \models & a & \iff & a \in \sigma_0 \\
\sigma & \models & \neg\varphi & \iff & \sigma \not\models \varphi \\
\sigma & \models & \varphi_1 \wedge \varphi_2 & \iff & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma & \models & \bigcirc\varphi & \iff & \sigma^1 \models \varphi \\
\sigma & \models & \varphi_1 \mathbin{\mathsf{U}} \varphi_2 & \iff & \exists i \geq 0. \sigma^i \models \varphi_2 \text{ and} \\
& & & & \forall j. 0 \leq j < i \Rightarrow \sigma^j \models \varphi_1
\end{array}
$$

transition relation, and $L : S \to \mathcal{P}(AP)$ a labelling function. Intuitively, each *label* $L(s)$ is the set of atomic propositions that hold in state s, while the transition relation captures the temporal relations between states: given two states s, s', we say that s' is a *successor* of s if $(s, s') \in R$. This means that the system can evolve from state s to s' in one instant. We assume that every state has at least one successor. Given a state s_0, a *path* rooted in s_0 is a sequence of states $\pi = \langle s_0, s_1, \ldots \rangle$ such that each s_{i+1} is a successor of s_i. Finally, we can associate to each path π a *trace* $\sigma = \langle L(s_0), L(s_1), \ldots \rangle$, i.e., the sequence of labels associated to each state of π.

Semantics of LTL. A trace σ *models* an LTL property φ if the pair $\langle \sigma, \varphi \rangle$ belongs to the satisfaction relation $\models$, described in Table 2.1b. In that table, we use σ_i to denote the i-th component, or *step*, of a trace σ; on the other hand, $\sigma^i \triangleq \langle \sigma_i, \sigma_{i+1}, \ldots \rangle$ is the *suffix* of σ that starts at σ_i. A trace σ satisfies the formula corresponding to an atomic proposition a if its first step contains a. Logical operators ($\neg$, $\wedge$) work in the usual way. Intuitively, $\bigcirc\varphi$ means that φ should hold from the second step of the trace onward; on the other hand, $\varphi_1 \mathbin{\mathsf{U}} \varphi_2$ means that φ_1 holds in the first $i - 1$ steps of the trace and that φ_2 holds in the i-th step (for some i). A Kripke structure models a property if all traces rooted in its initial states satisfy it.

2.2.2 Hennessy-Milner Logic

Labelled transition systems. A labelled transition system (LTS) is a tuple $\langle S, \Lambda, \to \rangle$, where S is a set of *states*, Λ a set of *labels*, and $\to \subseteq S \times \Lambda \times S$ a *labelled transition relation* describing the transitions that the system may perform in any given state. An element $\langle s, \alpha, s' \rangle$ of $\to$, commonly written as $s \xrightarrow{\alpha} s'$, represents that the system may evolve from state s to state s' by performing a transition with label α (informally, an *α-move*).

Hennessy-Milner Logic. Hennessy-Milner Logic (HML) [123] is a formalism to express properties over the states of a LTS. It is a *modal logic*, in the sense that it extends predicates with *modalities* that qualify their truth values. Specifically, HML formulas are made of one basic predicate $\top$ (which holds in every state of an LTS), the logical negation and conjunction operators ($\neg$ and $\wedge$), and one modality $\langle \alpha \rangle$ for each label α in the LTS (Table 2.2a). The HML formula

TABLE 2.2: Hennessy-Milner Logic.

(A) Grammar.

$$\begin{array}{rcl} \varphi & ::= & \top \\ & \mid & \neg\varphi \\ & \mid & \varphi_1 \wedge \varphi_2 \\ & \mid & \langle\alpha\rangle\varphi \\ & \mid & [\alpha]\varphi \end{array}$$

(B) Formal semantics.

$$\begin{array}{rclcl} s & \models & \top & & \\ s & \models & \neg\varphi & \iff & s \not\models \varphi \\ s & \models & \varphi_1 \wedge \varphi_2 & \iff & s \models \varphi_1 \text{ and } s \models \varphi_2 \\ s & \models & \langle\alpha\rangle\varphi & \iff & \exists s'.s \xrightarrow{\alpha} s' \text{ and } s' \models \varphi \\ s & \models & [\alpha]\varphi & \iff & s \models \neg\langle\alpha\rangle\neg\varphi \end{array}$$

$\langle\alpha\rangle\varphi$ (“after α, possibly φ”) holds in a state s if the system may perform an α-move from s to a state s' where φ holds. A second modality $[\alpha]$, denoting *necessity* rather than possibility, may be derived from possibility: $[\alpha]\varphi$ (“after α, necessarily φ”) holds in s if every α-move that may be performed in s leads to a state where φ holds. In fact, it is equivalent to $\neg\langle\alpha\rangle\neg\varphi$. The semantics described above are formalised by a *satisfaction relation* $\models$, which is defined as the least relation induced by the rules of Table 2.2b.

Extensions to HML. Finite-length HML formulas are not suitable to express many interesting properties. For instance, the *deadlock freedom* property (i.e., each state reachable from a state s has at least one successor) would require an HML formula of infinite length, as it needs to account for arbitrarily long sequences of transitions. A possible solution to this drawback is to introduce recursive properties. Deadlock freedom, for instance, could be recursively expressed as follows: “A state s is deadlock-free if it has at least one successor and all its successors are deadlock-free”. The semantics of recursive properties is given in terms of recursive equations over sets of states in the LTS of interests. These equations admit a least and a greatest fixed point: intuitively, checking liveness properties involve finding a least fixpoint for one such equation, while checking safety properties require the computation of a greatest fixed point [43]. The modal μ-calculus [150] is a variant of HML that has recursive operators and allows to freely combine least- and greatest- fixpoint operators within a formula. It has been shown that the modal μ-calculus is expressive enough to embed many temporal logics. In particular, both LTL and Computation Tree Logic (CTL) [60] may be seen as fragments of that logic.

2.3 Languages and analysis platforms for multi-agent systems

This section provides an overview of existing languages and platforms for the specification and analysis of multi-agent systems. Domain-specific languages (DSLs) [237] with tailored, higher-level primitives can compensate for the sources of complexity described in Section 2.1.3, thus making specifications more compact and facilitating both formal analysis and informal reasoning [80]. A drawback of DSLs is that the heterogeneity of the domains might be detrimental for compositionality, and it might become harder to specify complex systems by composing available solutions. Adopting excessively specific formalisms is another risk, as the capability of the language to realistically describe scenarios of interest might be affected. Thus, linguistic support should aim at achieving an acceptable trade-off between expressiveness and generality.

2.3.1 Belief-Desire-Intention

Belief-Desire-Intention (BDI) is a theory of agency [46, 179] that has been extensively applied to the specification of artificial agents [207]. According to the BDI theory, an agent's *practical reasoning* (i.e., its choice of actions) is influenced by its *beliefs* about the current state of the world, and is directed by its *desires*. A desire may be a goal, a constraint, or in general a description of states of the world that the agent may want to reach or avoid. Beliefs and desires lead an agent to form *intentions*, which are commitments to perform a sequence of actions (i.e., a *plan*) to satisfy its desires. The theory inspired several programming languages, that support the definition of beliefs, desires, and intentions (or similar concepts) through specific primitives. One of the first formalisms to follow this paradigm was Agent-0, which also popularised the term *agent-oriented programming* to refer to BDI-inspired languages [220]. Other examples include AgentSpeak(L) [206] and its implementation Jason [42]; 3APL [130]; Agent Factory [211]; and more [41]. These languages are typically tied to an interpreter or simulation framework that allows to empirically analyse the behaviour of specified systems [40, 74, 212]. AJPF [85] can instead perform formal verification of several BDI languages, by relying on an intermediate representation in Java which is checked via Java Pathfinder [119].

2.3.2 Simulation languages

While the term *simulation* usually designates a visual representation of an evolving system, we use the term to refer to any technique that is able to produce one or more finite *traces* of a given system. Due to the finiteness of traces and the huge state space of most multi-agent systems, simulation can only provide shallow guarantees on the overall correctness of a system [245]. However, it also offers several advantages over more formal analysis techniques. A simulation platform can reproduce the laws of physics effortlessly and in deep detail, which may be of importance in several fields, such as robotics. Simulations also scale well in the number of agents and in the complexity of their behaviour, and they are greatly appreciated in those fields that would otherwise rely exclusively on real-world experiments. For instance, simulations can represent very large sample sizes, time spans and distances that would be impossible to observe in a controlled experiment. They allow to observe and control the full state of the system, at any given time; moreover, simulations may be effortlessly reproduced in every detail, whereas reproducing a comparable real-world experiment may be exceedingly difficult. The evolution of a simulation may be compared to empirical findings, and researchers may iteratively adjust parameters within their models to fit real-world patterns. In the process, they may gain novel insights into the origin and meaning of those patterns [113]. Simulations are especially valuable in those fields where reproducible experiments are impossible, such as archaeology [17].

Early examples of simulation languages include Simscript [89] and Simula [73]. However, they both anticipate the widespread adoption of agent-based modelling. Currently, the most popular agent-based simulation language might be NetLogo [243]. The basic entities provided by NetLogo are *turtles* (mobile agents), *patches* (regions of space), and *links* (connections between turtles). The user may define additional kinds of entities, known as *breeds*, or customise the language through Java extensions [222]. Another simulation-oriented language is Buzz [196], which allows to program the behaviour of robotic multi-agent systems through primitives for ensemble formation, data exchange between neighbours, and *virtual stigmergies*. A virtual stigmergy is a distributed data structure that allows a group of agents to eventually agree on a set of key-value pairs [198]. Buzz systems may be simulated on the ARGoS platform [197].

A popular alternative to the creation of a new language is to build modelling and simulation libraries on top of existing general-purpose languages. These libraries provide reusable components to define common features of a MAS, such as its spatial environment, communication constraints on the agents, and so on. However, the user has to define the behaviour of agents in one of the supported languages: thus, the issues outlined in Section 2.1.4 are only partially mitigated. Example of such libraries include the Swarm Simulation System (available for Objective-C and Java) [128]; MASON (Java-based) [164]; and ASCAPE (Java-based) [136]. Similarly, in the ROS robotic middleware [204] the behaviour of an agent is programmed through general-purpose languages (C++, Python) as a set of interacting *nodes*. Interaction happens either through remote procedure calls or via publish-subscribe messaging. ROS does not provide multi-agent capabilities, and several third-party libraries have been proposed to fill this gap [148, 161]. While ROS does have a standard simulation environment, namely Gazebo [144], its open architecture led to the development of many compatible simulation platforms, such as Stage [239], SwarmSimX [151], V-REP [210], and Morse [90].

2.3.3 Process algebras

A process algebra, or process calculus, is a mathematically rigorous formalism to describe concurrent systems [77]. A process algebra typically features an alphabet of *actions* that the system may perform, and a set of *operators* to describe complex systems by combining simpler components. Early process algebras include CCS [180], CSP [132], ACP [25, 26], and LOTOS [140]. Examples of more recent calculi are LNT [103] and mCRL2 [115]. All the aforementioned calculi, or variants thereof, provide *value-passing* actions that allow processes to exchange information during their execution.

Operational semantics. A (well-formed) *process term* in a given process calculus is a syntactic term constructed according to the grammar of that calculus. A *semantics* of a process algebra is a rigorous definition of the behaviour of well-formed process terms. Specifically, an *operational semantics* associates to each process term a *labelled transition system* (LTS). In the *structural operational semantics* (SOS) approach [200], such an LTS is defined by induction on the structure of process terms: its set of states is the set of process terms, and the transition relation $\rightarrow$ is specified by means of inference rules. Each rule is in the form

$$\frac{antecedent_1 \quad antecedent_2 \quad \cdots \quad antecedent_n}{consequent}$$

meaning that, if $\rightarrow$ satisfies all of the antecedents, then it must also satisfy the consequent. Rules with no antecedents must be unconditionally satisfied: they are called *axioms*. The transition relation $\rightarrow$ is the smallest relation that satisfies all rules. Whenever $P \xrightarrow{\alpha} Q$, one may informally say that P can perform an α-move and become Q.

Examples of process-algebraic terms and operators. We now provide a description of some terms and operators commonly found in process calculi, along with their SOS rules.

Elementary terms. Most process algebras have the *idle*, or *deadlocked* process, as an elementary term. It is typically denoted by **0** or stop. The idle process cannot perform any action at all, and is therefore associated with the empty LTS. Some calculi, such as CSP, provide a separate notion of a *successfully terminated* process, denoted as **1**, skip, or $\surd$. This process may only signal its own termination and does nothing afterwards. Thus, it corresponds to an LTS that may only perform a $\surd$-move and become the idle process (rule TICK). The *single-action* process α, where α is an element of the alphabet of actions, is also treated as an elementary process term by calculi such as CSP and ACP. Such a process may perform α and terminate (rule ACT).

$$\frac{}{\surd \xrightarrow{\surd} \mathbf{0}} \ (\textsc{tick}) \qquad\qquad \frac{}{\alpha \xrightarrow{\alpha} \surd} \ (\textsc{act})$$

Expressing sequentiality. The *action prefixing* operator is a common means of expressing sequential actions: it allows one to define a process term $\alpha.P$ that may perform action α and then continue as P (rule PREFIX). Several process calculi provide a *sequential composition* operator to either replace (ACP, LNT) or complement (CSP, LOTOS) action prefixing. Intuitively, a term $P; Q$ behaves as P until it terminates, and then continues as Q. The exact way in which "behaves as P" and "continues as Q" is formalised varies from one calculus to another: here we show one possible set of rules [77, 98]. Rule $\textsc{seq}_1$ states that $P; Q$ may evolve as $P'; Q$ as long as P may evolve into P' by means of an α-move; rule $\textsc{seq}_2$ says that, if P can signal its own termination (by means of a $\surd$-move) and Q can perform an α-move to Q', then the composite process may evolve as Q'. Overall, action prefixing allows for more compact semantics and facilitates proofs; sequential composition, on the other hand, makes complex sequential behaviours simpler to describe, and better resembles the style of traditional programming languages [98].

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\ (\textsc{prefix}) \qquad \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq \surd}{P;Q \xrightarrow{\alpha} P';Q}\ (\textsc{seq}_1) \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\alpha} Q'}{P;Q \xrightarrow{\alpha} Q'}\ (\textsc{seq}_2)$$

Alternative behaviour. The term $P+Q$ may behave either as P or as Q. The rules below describe $+$ as a *mixed choice* operator, meaning that the decision of proceeding as one process or the other may be controlled by an external process or may happen nondeterministically.

$$\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

Parallel composition. A term $P \mid Q$ represents two processes, P and Q, that evolve together. The simplest form of parallel composition is *interleaving*, where the two processes just alternate their respective actions. Most process algebra also feature a notion of *synchronisation* between processes. For instance, in CCS, a process willing to perform an action a may evolve together with another willing to perform a *complementary* action $\overline{a}$. This event represents a *handshake* between the two processes. As a result, the parallel composition performs an internal action, usually denoted by τ. Notice that the two processes may also alternate the execution of a and $\overline{a}$. Other process calculi, however, provide composition operators where processes are *forced* to synchronise on specific pairs of actions.

Synchronisation algebras provide a fairly general way to formalise the concept of synchronisation, as they reproduce the behaviour of several existing parallel composition operators, such as the one in CCS and the merge operator in ACP [246]. A synchronisation algebra is a partial function $\sigma : Act \times Act \cup \{*\} \hookrightarrow Act$, where Act is the alphabet of actions and $*$ is a distinguished symbol representing *idleness*. Intuitively, $\sigma(a,*) = a$ means that a can be performed by one party while the other is idle, while $\sigma(a,*) = \bot$ forbids that (rules $\textsc{par}_{1,2}$). On the other hand, $\sigma(a,b) = c$ means that two processes may synchronise if they can respectively perform a and b, and the composite process does perform action c; finally, $\sigma(a,b) = \bot$ indicates that a synchronisation on the pair of actions a, b (rule $\textsc{par}_3$) is not possible.

$$\frac{P \xrightarrow{\alpha} P' \quad \sigma(\alpha, *) \neq \bot}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \;(\text{PAR}_1) \qquad \frac{Q \xrightarrow{\alpha} Q' \quad \sigma(*, \alpha) \neq \bot}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \;(\text{PAR}_2)$$

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q' \quad \sigma(\alpha, \beta) \neq \bot}{P \mid Q \xrightarrow{\sigma(\alpha,\beta)} P' \mid Q'} \;(\text{PAR}_3)$$

Process algebras as a modelling tool. Process algebras have several advantages as a modelling tool for multi-agent systems (and systems of heterogeneous, interacting components, in general). First, composition and interaction between components are core process-algebraic concepts. Moreover, process calculi allow the user to choose the alphabet of basic actions according to the level of abstraction they want to achieve in their model. Lastly, they are amenable both to simulation and formal analysis [235]. Several works investigated the use of process algebras as a multi-agent modelling tool. For instance, WSCCS (Weighted Synchronous CCS) [234], which extends CCS with probabilistic operators and prioritised actions, has been used to model ant colonies [174] and population dynamics [174]. Bio-PEPA [57] has been used for epidemiological modelling [58], and to analyse the collective behaviour of crowds [167] and robotic swarms [168]. Finally, the PALPS process algebra [195] has been specifically developed to describe and analyse ecosystems.

Analysis and verification. In any process algebra, different process terms may describe the same behaviour, or slightly different behaviours that are nonetheless undistinguishable from one another under some specific context. Thus, it is convenient to formalise one or more *equivalence* relations between process terms. An *equivalence checking* algorithm, then, is a decision procedure that takes two process terms as input and returns *yes* if and only if they are equivalent. Equivalence checking is a natural way to prove the correctness of a process term representing an *implementation*: to do so, it is sufficient to encode the correctness *specification* as a separate process term, and then check that the implementation and the specification are equivalent [76]. An alternative way to prove that a process term conforms to a specification is to encode the latter as one or more formulas in an adequate formalism, and then verify that the LTS of the given process term satisfies them.

2.3.4 Other paradigms

Actor model. The concept of *actors* is closely related to that of agents. An agent receives percepts and performs actions; similarly, an actor is an entity that receives and sends *messages*. The *actor model* is based on the principle that any kind of computation may be expressed (and performed) by a set of interacting actors [126]. The behaviour of an actor may be formally defined by a function that, given a set of input messages, returns a set of output messages, a set of new actors and a possibly new behaviour that the actor will follow after receiving that input [3]. The Erlang language [15] is perhaps the most popular implementation of the actor model; Elixir[5] is another, more recent actor-based language. The popularity of this paradigm is also attested by the many libraries and frameworks that allow to create actor-based systems with general-purpose programming languages [135].

Population protocols. Population protocols are a model of distributed computing based on anonymous, mobile agents with uniform behaviour that may change their state through pairwise interactions [12]. The allowed interactions are encoded as a *transition relation* over pairs of states. Each transition, typically denoted as $(p, q) \mapsto (p', q')$, means that a pair of agents with states p and q can interact and evolve, respectively, to states p' and q'. The first element of the pair is called the *initiator* of the interaction, while the other is called the *responder*. Transition relations may be *asymmetric*, i.e., a pair of agents may evolve in different ways depending on which one is the initiator. Given a set Q of individual states, a *configuration* is a multiset over Q, which expresses how many agents are in a specific state. A system defined by a population protocol never terminates: rather, it is said to *stabilise* when it reaches a configuration that cannot be changed by further transitions. With the fairness assumption that any configuration which is reachable infinitely often is eventually reached, some population protocols always stabilise to a configuration which satisfies a given predicate over its initial configuration. In this case, we say that the protocol *computes* the predicate. The PEREGRINE tool [37] can simulate the evolution of a population protocol, and can verify whether a protocol computes a given predicate. It also supports parameterised verification, i.e., it can check that a protocol computes a predicate for an unbounded number of agents.

Attribute-based communication. The literature features several instances of languages and process algebras with attribute-based primitives, such as AbC [2] and CARMA [163]. Both feature non-blocking output and blocking input primitives, which are decorated with attribute-based predicates to select the desired partners for interaction. A message is exchanged whenever (a) an agent (the sender) performs an output action; (b) there are one or more agents (the *receivers*) waiting on an input action; (c) the sender's attributes satisfy the predicate specified by the receivers, and vice versa; (d) the structure of the sent message matches the pattern required by the receivers. This allows for implicit multi-party synchronisation between one sender and multiple receivers. Similar attribute-based primitives have also been implemented on top of existing programming languages through libraries or extensions, such as AErlang (Erlang) [81], ABEL (Erlang) [82], and GoAt (Go) [1]. The SCEL language [78, 79] does not feature an explicit

[5] `https://elixir-lang.org`.

input action: agents may perform attribute-based actions to insert, retrieve, or withdraw items from the *knowledge repositories* of other agents.

2.4 Program verification

In this section, we review several formal verification techniques, as well as a selection of tools that implement said techniques. Specifically, we focus on techniques and tools that allow to perform either *termination analysis* or *reachability analysis*, or both. Before doing so, we provide some basic terminology.

A *computational problem* $\mathfrak{P}$ is a function that associates to each *instance* a (possibly empty) set of *solutions* for that instance. We say that $\mathfrak{P}$ is a *decision problem* when the solution of every instance is either yes or no. A given decision problem may always be identified by the set of instances whose answer is *yes*. For instance, the problem "is n an odd natural number?" is identified by the set $\text{ODD} = \{1, 3, 5, \ldots\}$. Therefore, from now on we will use the notation $x \in \mathfrak{P}$ to mean that $\mathfrak{P}(x) = yes$.

A *decision procedure* is an algorithm for a decision problem. Therefore, its output is either *yes* or *no*. Given a decision procedure $\mathcal{A}$ for a problem $\mathfrak{P}$, let us denote by $\mathfrak{A} \triangleq \{x \mid \mathcal{A}(x) = yes\}$ the set of all instances x such that $\mathcal{A}(x)$ terminates with output *yes*. Then, $\mathcal{A}$ is *sound* if $\mathfrak{A} \subseteq \mathfrak{P}$; it is *complete* if $\mathfrak{P} \subseteq \mathfrak{A}$. Intuitively, a sound procedure always gives correct answers, but it may fail to provide an answers for some instances $x \in \mathfrak{P}$ (e.g., because it never terminates for those instances). On the other hand, a complete procedure gives all the correct answers, but it may also give wrong ones (i.e., it may happen that $\mathcal{A}(x) = yes$ even though $x \notin \mathfrak{P}$).

Termination analysis is the decision problem of determining whether a given program *unconditionally terminates*, i.e., all of its executions are of finite length. When discussing termination analysis, we will say that a program is *safe* if it unconditionally terminates, and is *unsafe* otherwise.

Reachability analysis is the decision problem of determining whether any *error states* of a given program are reachable from its initial states, i.e., if the program admits an execution that leads to an error state. The program is said to be *safe* if and only if no error state is reachable. Error states are either identified by labels in the source code of the program, or expressed in terms of *assertion* statements such as `assert(`φ`)`. If such a statement is reachable and the condition φ is not satisfied, the program is unsafe.

In the context of this work, *sequentialization* [203] is any source-to-source transformation that takes a concurrent program $\mathbb{P}$ as input and generates a sequential program $\mathbb{P}_{seq}$ that reproduces the traces of $\mathbb{P}$. The verification techniques discussed in this work are aimed at sequential programs;

however, they may be coupled with a sequentialization transformation to verify concurrent programs. Since this approach appears to be effective in practice [19, 138, 187], we will refrain from discussing techniques that natively handle concurrent programs.

Throughout this section, the term *under-approximation* designates techniques that explore a subset of the state space of the input program. *Over-approximating* techniques, on the other hand, consider a superset of the state space. Under-approximation may result in false positives, i.e., an unsafe program is deemed safe, or a non-terminating program is considered to unconditionally terminate. Over-approximation may instead produce false negatives.

2.4.1 Program verification techniques

Symbolic execution. In symbolic execution [143], all nondeterministic variables within the input program are replaced by *symbolic variables*. A symbolic variable is initially unconstrained, i.e., its value is assumed to be arbitrary. If the control flow of the program does not depend on any symbolic variable, then all its concrete executions may be represented by a single symbolic execution. On the other hand, if a branching condition does depend on a symbolic variable, the symbolic execution is forked into two separate ones, and constraints are added to the variable to record the branching condition. In the end, one obtains a symbolic *execution tree* where each root-leaf path represents a set of concrete executions. If no path ever visits an error state, the input program is safe. The main drawback of symbolic execution is the so-called *path-explosion problem*: the number of paths in the execution tree is exponential in the number of branching statements (over symbolic variables) in the input program. The path-explosion problem is exacerbated by loops, as each check on the loop condition is treated as a separate branching statement [29]. The use of (over-approximating) symbolic execution for termination analysis has also been investigated [240]: some of these approaches [124, 226] prove termination by constructing a *symbolic execution graph* based on *separation logic* [189], a formalism that helps reasoning about programs that feature dynamic memory allocation and complex data structures.

Model checking. Given a model $\mathcal{M}$ of a system of interest, and a *property* φ, the *model checking problem* is to determine whether $\mathcal{M} \models \varphi$ [65]. A model checking algorithm, then, is a decision procedure for the model checking problem: it takes $\langle \mathcal{M}, \varphi \rangle$ as input, and returns *yes* or *no* depending on whether $\mathcal{M} \models \varphi$ is proved or disproved. In the latter case, it is desirable that the algorithm returns also a *counterexample* (also known as a *violation witness*), i.e., a trace of $\mathcal{M}$ proving that the model violates φ. Several model checkers also return a *correctness witness* when $\mathcal{M}$ does satisfy the given property. Generally, such a witness is not a single trace, but rather a sub-model $\mathcal{M}' \subseteq \mathcal{M}$ that explains why φ holds in $\mathcal{M}$. A *software model checker* takes as input a program, computes a model of that program according to the semantics of the language, and then performs the actual model checking procedure.

Explicit-state model checking procedures represent each individual state of the input program in memory as they verify it. Examples of such representations include nondeterministic Büchi automata (NBA) [238] and Boolean equation systems (BES) [11]. Since representing the whole state space would be prohibitive even for modest programs, typically the model is generated and explored on-the-fly [105]. State space reduction techniques based on symmetries [61] or partial orders [107] are often necessary to handle larger systems. *Symbolic* model checking [52], instead, uses predicates to compactly represent states and transition relations involved in the verification process. A symbolic model checker may verify systems that are out of reach for explicit-state techniques by directly manipulating these predicates, by means of data structures such as binary decision diagrams (BDDs) [6] (in the case of Boolean predicates). Moreover, state space reduction may be combined with symbolic exploration to further improve the efficiency of the analysis [10].

Bounded model checking. The *bounded model checking* problem (BMC) is stated as follows [32]: given a model $\mathcal{M}$, a property φ, and a *bound* k, is there a counterexample *of length at most* k that proves $\mathcal{M} \not\models \varphi$? This is equivalent to check φ on a *bounded* version of $\mathcal{M}$. In fact, a bounded software model checker typically transforms the input program $\mathbb{P}$ into a *bounded* program $\mathbb{P}'$ by means of loop unwinding, function inlining, and other source-to-source transformations. The bounded program, together with property φ, can be efficiently encoded into a propositional logic formula and then fed to a Boolean satisfiability (SAT) solver [31]. An assignment that satisfies the formula corresponds to a trace of $\mathbb{P}'$ that violates φ; if no such assignment exists, then there is no counterexample of length k. SAT solvers provide several advantages over BDD-based algorithms in terms of time and memory consumption, and their use may be extended to unbounded model checking [177].

Further improvements may be obtained by relying on solvers for *satisfiability modulo theories* (SMT) [23]. Intuitively, while a propositional formula only ranges over Boolean variables, an SMT formula may contain predicates over variables of different *sorts*, such as integers, real numbers, etc. A *theory* over a given sort constraints the interpretation of predicates over variables of that sort, and may allow an SMT solver to quickly discharge predicates that would be harder to solve for a SAT-based tool [14]. Program encodings modulo theories are often more compact than their propositional counterparts, but sometimes they may not capture the actual behaviour of the program, due to differences between the used theory and machine arithmetics. Bit-precise theories exist to address this issue, but using them might reduce the advantage of using SMT over plain SAT.

BMC is an under-approximation technique: if it fails to find a counterexample of length k, one cannot conclude that the program is safe, unless the bound is big enough to cover the whole state space of the program. However, it can be used to prove safety by means of *k-induction* [219].

This algorithm relies on incrementally unwinding the input program $\mathbb{P}$. At the k-th iteration of the algorithm, a standard BMC check is performed on $\mathbb{P}_k$ (i.e., the program unwound up to depth k). If a counterexample is found, then $\mathbb{P}$ is unsafe and the procedure terminates. Otherwise, an *inductive step* is checked. This check attempts to prove the safety of $\mathbb{P}_{k+1}$, assuming that $\mathbb{P}_0, \ldots, \mathbb{P}_k$ are safe. If a proof for the inductive step is found, then the input program is safe and the procedure terminates; otherwise, the value of k is increased and the procedure starts over.

The above discussion focused on reachability analysis. A sound algorithm for termination analysis based on BMC exists. This algorithm unwinds all loops k times and adds an *unwinding assertion* at the end of each unwinding. The unwinding assertion condition is the negation of the loop condition: thus, it is satisfied if and only if the loop is fully unwound, i.e., the input program never performs more than k iterations of the loop. The bounded program is then model-checked: if all unwinding assertions are satisfied, it means that the input program unconditionally terminates. The value of k is a *completeness threshold* for the program: if the program satisfies a given property up to k steps, then it models the property [63]. If one or more unwinding assertions are violated, the algorithm starts over with a higher value of k. Thus, the algorithm may never terminate.

Abstraction-based techniques. Abstraction-based techniques build over-approximations of the program to verify. As a consequence, they do recognise all unsafe programs as such, but may incorrectly label a safe program as unsafe. In fact, a property violation found by these techniques may be either genuine or the effect of over-approximation.

Many such techniques can be defined as specialisations of *abstract interpretation* [70], «a method to design approximate semantics of programs» which may give sound answers to questions about their run-time behaviour [71]. In abstract interpretation, the term *concrete domain* refers to all those objects that describe the real behaviour of a program, as defined by its semantics. Instead of verifying a program by analysing its concrete domain, the core idea of abstract interpretation is to build a computable approximation of it, called an *abstract domain*, and use it to perform the verification task. The abstract domain is required to be a *sound* approximation of the concrete one, i.e., it must represent at least all of its elements.

Value abstraction is a common example of abstract interpretation: program variables are associated to mathematical objects taken from an abstract domain, and the concrete semantics of expressions is replaced by an abstract semantics over these objects. For instance, let us consider the abstract domain of *intervals* [69]. In this domain, a deterministic assignment $\mathtt{x} := \mathtt{v}$ becomes an assignment to a singleton interval: $\mathtt{x}^\sharp := [v, v]$. On the other hand, a nondeterministic assignment $\mathtt{x} := *$ is abstracted as $\mathtt{x}^\sharp := [-\infty, \infty]$. The addition operator is abstracted as $[a, b] +^\sharp [c, d] \triangleq [a + c, b + d]$, meaning that the sum of two variables with intervals respectively

$[a, b]$ and $[c, d]$ must belong to the interval $[a + c, b + d]$. Similar rules are defined for each arithmetic operator and to handle special cases (e.g., division by an interval that crosses the zero).

Predicate abstraction [110], instead, considers a set of predicates *Preds* over program variables, such that they partition the state space of the program, and approximates sets σ of concrete program states by the conjunction of all predicates that hold in σ, by means of an *abstraction function*:

$$\alpha(\sigma) = \bigwedge \{p \in \mathit{Preds} \mid \sigma \models p\}.$$

Reachability analysis via predicate abstraction works by first computing an abstract initial state and an abstract transition relation for the program from their concrete counterparts. Then, one obtains the set of reachable abstract states by repeatedly applying the abstract transition relation to the abstract initial state. If this set contains no error states, then the program is safe. Clearly, the choice of predicates deeply influences the efficacy of the analysis. Fine-grained predicates might produce less false negatives, but they may also induce a large abstract state space, hindering performance. Conversely, an analysis based on a coarser set of predicates may quickly produce an answer, but is more prone to run into spurious property violations.

Another technique achieves abstraction through *function summarisation*. A *summary* of a function is a formula that approximates the relation between its input and output variables [131]. If the summary is computed by means of abstraction, e.g., by using one of the aforementioned value abstract domains, then it captures all possible behaviours of its corresponding function while possibly introducing spurious ones. These summaries may be used in the encoding phase in place of encoding the actual body of corresponding functions, thus improving performance of the underlying decision procedure. Notice that under-approximating summaries also have several useful applications [55].

A common way to address the issue of spurious property violations is *counterexample-guided abstraction refinement* (CEGAR) [62]. A CEGAR algorithm starts by verifying an abstract model of the input program. If this model does not admit any counterexample, then the algorithm recognises the program as safe and terminates. Otherwise, the algorithm checks the counterexample against the concrete program. If the counterexample is genuine, then the program is unsafe and the algorithm terminates. If it is spurious, the algorithm uses it to *refine* the abstraction. In the case of predicate abstraction, the algorithm starts with a rather coarse set of predicates and invalidates spurious counterexamples by introducing new, finer-grained predicates. Then, the algorithm computes a new abstraction of the input program and starts over. Thus, a CEGAR loop only returns feasible counterexamples, but on the other hand it might never terminate. This *abstract-check-refine* approach may be improved by means of *lazy abstractions*, i.e., by refining different parts of the abstract model to different degrees of precision, driven by the verification procedure [125]. Another tool to (possibly lazily) refine the abstract model is *Interpolation* [72]. Interpolation-based algorithms may prove that error paths in the abstract model are unfeasible by computing a sequence of formulas along the path that leads to the error state, such that each formula in the sequence maps to a step in the error path and implies the next formula. Such a sequence is called an *interpolant*. Since refinement through interpolants does not require computing the abstract transition relation (which is the dominant cost when using, say, basic predicate abstraction), it may guarantee further performance gains [178].

Property directed reachability. A safety property for a program $\mathbb{P}$ can be expressed as an *invariant*, i.e., a predicate $P(s)$ that should hold for any program state s. If $I(s)$ is a predicate that holds if and only if s is an initial state of $\mathbb{P}$, and T another predicate encoding the transition relation of the program (i.e., $T(s, s')$ holds if and only if s' is a successor state of s), then one could try to prove the safety of $\mathbb{P}$ by induction. Specifically, one should prove the base case $\forall s.I(s) \Rightarrow P(s)$ and the inductive step $\forall s, s'.(P(s) \wedge T(s, s')) \Rightarrow P(s')$. However, P may be too weak to prove the inductive step. As a solution, one should find an *inductive strengthening* of P, i.e., a predicate $P' = P \wedge \psi$ such that (a) ψ is an invariant and (b) $P'(s) \wedge T(s, s') \Rightarrow P'(s')$. Since P' entails P, by finding it one also proves that P is an invariant. Of course, proving that ψ is an invariant may also involve finding an inductive strengthening, so one may need to prove this intermediate step recursively.

Essentially, a *property directed reachability* (PDR) algorithm, such as IC3 [44], automates the above procedure. In fact, the PDR term has been retroactively coined to refer to IC3 and its derivatives, such as GPDR [133] and RecMC [145]. We now give a brief description of IC3, but other PDR algorithms share the same basic concepts. The algorithm maintains a sequence of *frames* F_i, where F_0 is the set of initial program states and each F_k, $k > 0$, over-approximates the set of program states that are reachable in k transitions. Frames are not explicitly represented, but rather symbolically expressed as a conjunction of clauses. At each iteration k, assuming that the invariant P holds in all frames up to F_k, the algorithm tries to show that

$$\forall s.F_k(s) \wedge T(s, s') \Rightarrow P(s') \tag{2.1}$$

If this is the case, and furthermore F_k is inductive (i.e., $F_k(s) \wedge T(s, s') \Rightarrow F_k(s')$), then P is an invariant and the algorithm terminates. If Eq. 2.1 holds but F_k is *not* inductive, then P is added to the clauses of F_{k+1}, a new frame F_{k+2} is constructed, and the algorithm iterates on $k + 1$. The construction of F_{k+2} happens by propagating clauses from the previous frames: thus, it can be seen as a form of predicate abstraction. Finally, if Eq. 2.1 does not hold, a state $z \in F_{k+1}$ that violates P is revealed. In this case, the algorithm tries to refine $F_1, \ldots, F_{k+1}$ to prove that z is not a reachable state (at least, not in $k + 1$ steps). If it fails to do so, then a trace leading to z is returned as a counterexample and the algorithm terminates; otherwise, the algorithm can resume its execution. It is crucial to observe that a PDR algorithm never returns spurious counterexamples, and it never declares an unsafe program to be safe.

Semantics-based techniques. Many real-world verification tools are tied to a specific programming language: by contrast, a semantics-based procedure accepts as input a program together with a formal semantics of the language in which the program is written. In other words, the verification procedure is parameterised in the semantics of the given language [68]. For instance, the TVLA tool [160] exploits the operational semantics of a language to generate an abstract interpreter for concurrent programs in that language. Operational semantics can also be used to generate reachability verifiers for sequential programs in the given language [224]. For a given program and property, verification conditions in the form of Horn clauses can be automatically generated from the semantics of the language [75]. Other approaches target termination analysis [240], semantics-based simulation [185], or model checking through rewriting systems [209]. All the approaches mentioned above rely on structural operational semantics, with the exception of [209], which requires the language semantics to be specified in rewriting logic.

2.4.2 Existing program verification tools

Tools for sequential C programs. Due to the importance of the C programming language in the software industry, the literature provides a wide set of tools for the verification of C programs. We briefly describe the features of a selection of tools that have given conclusive results in the experimental phase of this work. Four of the considered tools are based on bounded model checking. Table 2.3 contains a summary of their respective features.[6] 2LS [55] supports k-induction and may perform termination analysis via function summarisation. 2LS may summarise functions through several abstract domains, but it currently lacks support for arrays. CBMC [59] is a SAT-based bounded model checker for reachability analysis over C programs, and is used by 2LS as an analysis backend. An experimental version of CBMC, submitted to the 2019 Software Verification Competition (SV-COMP) [27], can also perform termination analysis by checking unwinding assertions. BMC tools that rely on SMT include ESBMC [97], which also implements k-induction and termination analysis based on incremental unwinding. While ESBMC performs its own SMT encoding, SMACK [205] first compiles the input program to LLVM bytecode [159] and then translates this intermediate representation into the Boogie verification language [22].

Symbiotic [54] performs reachability analysis by combining symbolic execution and *program slicing* [241]. Slicing the input program eliminates instructions that are not relevant to the property under verification, and thus may help with the path-explosion problem. Symbiotic also implements a termination analysis algorithm, where termination is reduced to unreachability

[6]We only consider those features that are relevant to the experimental evaluation of Section 5.4.2, and ignore others, such as support for floating-point arithmetic, pointers, or dynamic memory allocation.

TABLE 2.3: Comparison of BMC-based tools.

Tool	**Back end**	**Summarisation**	**k-induction**	**Termination analysis**	**Arrays**
2LS	SAT	●	●	●	
CBMC	SAT			●	●
ESBMC	SMT		●	●	●
SMACK	SMT				●

of infinite loops. This is a sufficient, but unnecessary condition for termination, and thus the tool may report a terminating program as non-terminating. Ultimate Automizer [120] verifies safety properties by using automata to encode sets of traces of the program under verifications. The automata are refined via a CEGAR loop until the tool either finds an effective trace leading from the initial state of the program to an error state, or proves that no such path exists. The tool is able to dynamically choose the refinement strategy at each iteration of the CEGAR loop [122]. CPAchecker [28] is a configurable program analysis platform that supports many different techniques.

In this work we consider three configurations for reachability analysis. The first one performs *explicit-value analysis* by using a simple abstract domain where each program variable is bound to either an integer, $\top$ (denoting a nondeterministic value), or $\bot$ (denoting a contradiction: i.e., an abstract state where a variable is bound to $\bot$ does not represent any concrete state) [30]. Counterexamples found by this configuration are cross-checked (via predicate abstraction) to prevent spurious results. The second configuration is instead based on predicate abstraction with CEGAR, while the last one performs bit-precise k-induction. We also consider a configuration for termination analysis based on ranking functions [120]; as with 2LS, applying these over-approximating techniques to programs with arrays may lead to unexpected results. Finally, we consider two tools based on PDR algorithms. VVT [116] implements a CEGAR extension of IC3: this approach is known as CTIGAR (counterexample to induction-guided abstraction refinement) [33] and allows for the verification of infinite-state systems. SeaHorn [117] features another PDR algorithm, namely RecMC.

Process-algebraic tools. We cannot account for all the tools that have been developed within the process algebra community: we only provide an overview of representative tools and the calculi they support. All of them provide both equivalence checkers and model checkers. Explicit-state toolboxes include Winston (CCS) [165], CWB (CCS, CSP, LOTOS) [66], FDR (CSP) [108], and CADP (LOTOS, LNT) [99]. Only the last two are actively developed.[7] Other toolsets, such as mCRL2 (for the eponymous process calculus) [50] and LTSmin [36], implement both explicit-state and symbolic verification techniques.

[7]See `https://cocotec.io/fdr/` and `https://cadp.inria.fr`

2.5 Summary

In this chapter, we have introduced the terminology and concepts that constitute the background of this thesis. In Section 2.1, we provided an overview of the features and modelling challenges typical of multi-agent systems, along with a selection of illustrative examples. In Section 2.2, we described Linear Temporal Logic and Hennessy-Milner Logic, two well-known property specification formalisms, and gave definitions for labelled transition systems and Kripke structures. Section 2.3 lists several existing languages and platforms that have been used to specify and analyse multi-agent systems. Lastly, Section 2.4 describes the state of the art in automated reachability analysis and termination analysis, by considering a selection of techniques (Section 2.4.1) and related verification tools (Section 2.4.2).

Chapter 3

A specification language for MAS

In this chapter we introduce a language for the specification of multi-agent systems. The language is built on top of a core process calculus, LAbS (a Language with Attribute-based Stigmergies) [83], which is the result of a systematic study on the main factors of complexity in multi-agent systems [80]. The core language combines the concepts of virtual stigmergy [198], with attribute-based communication [2].

3.1 Preliminary definitions

Interfaces. Each agent in a LAbS system is equipped with a set of *attributes* (or *local variables*). An attribute has a name and a (possibly undefined) value. We use $\mathcal{K}_I$ and $\mathcal{V}$ to denote the set of attribute names and values, respectively. The store of local variables of an agent is called its *interface*, and may be seen as a partial function $I : \mathcal{K}_I \hookrightarrow \mathcal{V}$. We denote by $\mathcal{I}$ the set of all interfaces. Updating an attribute x to value v within an interface I amounts to defining a new interface I', which is the same as I except that $I'(x) = v$. We denote such an interface by $I[x \mapsto v]$.

Attributes can be specified in the initialisation phase and modified at runtime; they represent either a variable in the agent's memory, or a physical property of the agent (for instance, its position). LAbS makes no distinction between these two kinds of information. For instance, an agent may move by updating the attribute that represents its position.

Local stigmergies. Each LAbS agent is also equipped with a local stigmergy. Formally, a local stigmergy $L : \mathcal{K}_L \hookrightarrow (\mathcal{V} \times \mathbb{N})$ is a partial function from a set $\mathcal{K}_L$ of stigmergic variables, or keys, to a set of timestamped *values* (again denoted by $\mathcal{V}$). Intuitively, a timestamp captures the moment when a value was assigned to a given stigmergic variable. We use natural numbers

TABLE 3.1: Operations on the virtual stigmergy.

$$\frac{L(x) = \bot}{L \oplus (x, v, t) = L[x \mapsto (v, t)]}\ (\text{ADD}) \qquad \frac{t > \mathit{time}(L, x)}{L \oplus (x, v, t) = L[x \mapsto (v, t)]}\ (\text{UPDATE})$$

$$\frac{t \leq \mathit{time}(L, x)}{L \oplus (x, v, t) = L}\ (\text{DISCARD})$$

to represent timestamps. We assume that $\mathcal{K}_L$ is disjoint from $\mathcal{K}_I$ and denote by $\mathcal{L}$ the set of all local stigmergies. If $(x, v, t) \in L$, we say that v is the *value* of x and that t is its *timestamp* in the local stigmergy L. We refer to these as $\mathit{value}(L, x)$ and $\mathit{time}(L, x)$, respectively.

Insertion of a value in a local stigmergy is a function $\oplus : \mathcal{L} \times (\mathcal{K}_L \times \mathcal{V} \times \mathbb{N}) \longrightarrow \mathcal{L}$ defined as the smallest relation that satisfies the rules in Table 3.1. These rules imply that only new values are successfully inserted in the local stigmergy. A value is new if its key is missing from the local stigmergy or it has a more recent timestamp than the existing one.

Virtual stigmergies were first introduced as part of the Buzz language [196], but were not given a formal definition. Besides that, our description slightly deviates from Buzz stigmergies in a few points. First, the former are based on Lamport timestamps [154] and rely on unique agent identifiers to break ties, which may occur when the same timestamp is used more than once; our language is currently more limited, as it relies on a global clock (see Section 3.2.5). However, our calculus also generalises some of the concepts related to the virtual stigmergies of Buzz. Most importantly, in our language the ability to exchange information through the stigmergy is not directly constrained by spatial vicinity. In fact, there is no explicit concept of an agent's position at all. Rather, we rely upon attribute-based predicates to determine whether two agents are allowed to communicate. This is an important source of flexibility, as different means of communication for an agent can be modelled through different predicates. Furthermore, the ability of the agents to change their attributes at any time means that connections among agents can be dynamically established or removed.

3.2 Core LAbS process algebra

The syntax of LAbS is described in Table 3.2. In **expressions**, we assume that $v \in \mathcal{V}$, $x \in \mathcal{K}_L \cup \mathcal{K}_I$, and $\diamond$ stands for any binary operator over $\mathcal{V}$ (such as $+, -, \times \ldots$). In **guards**, $\bowtie$ denotes comparison relations over $\mathcal{V} \cup \{\bot\}$, namely $(=, <, >)$. We also assume that K is taken from a set of named processes.

TABLE 3.2: LAbS syntax.

$S ::=$	$a \mid a \parallel S$	Systems
$a ::=$	$\langle I, L, P, Zc, Zp \rangle$	Agents
$P ::=$	$0 \mid \surd \mid \alpha \mid P;P \mid P+P \mid P \vert P \mid K \mid g \to P$	Processes
$g ::=$	$\mathit{true} \mid e \bowtie e \mid \neg g \mid g \wedge g \mid g \vee g$	Guards
$\alpha ::=$	$x \leftarrow e \mid x \leftsquigarrow e$	Assignments
$e ::=$	$v \mid x \mid e \diamond e$	Expressions

A **system** is the parallel composition of a number of agents. An **agent** is a 5-ple $\langle I, L, P, Zc, Zp \rangle$ where $I \in \mathcal{I}$ is the *interface* of the agent; $L \in \mathcal{L}$ is the *local stigmergy* of the agent; P is a *process* describing the behaviour of the agent; Zc is the set of keys that the agent has to confirm (i.e., query); and Zp is the set of keys that the agent must propagate.

3.2.1 Processes and expressions

Processes are used to model behaviour of the agents. We present their syntax in Table 3.2 and their operational semantics in Table 3.3. There, P and Q denote processes while $\surd$ denotes successful termination, α represents the actions used to update attributes ($x \leftarrow e$) or stigmergic variables ($x \leftsquigarrow e$), with the result of the evaluation of an expression, while μ is a placeholder for either $\surd$ or α.

TABLE 3.3: Semantics of processes.

$$\frac{}{\surd \xmapsto{\surd} 0}\ (\textsc{tick}) \qquad \frac{}{\alpha \xmapsto{\alpha} \surd}\ (\textsc{act}) \qquad \frac{P \xmapsto{\mu} P'}{P+Q \xmapsto{\mu} P'}\ (\textsc{choice-l}) \qquad \frac{Q \xmapsto{\mu} Q'}{P+Q \xmapsto{\mu} Q'}\ (\textsc{choice-r})$$

$$\frac{P \xmapsto{\alpha} P'}{P;Q \xmapsto{\alpha} P';Q}\ (\textsc{seq}_1) \qquad \frac{P \xmapsto{\surd} P' \quad Q \xmapsto{\mu} Q'}{P;Q \xmapsto{\mu} Q'}\ (\textsc{seq}_2) \qquad \frac{P \xmapsto{\mu} P' \quad K \triangleq P}{K \xmapsto{\mu} P'}\ (\textsc{con})$$

$$\frac{P \xmapsto{\alpha} P'}{P \mid Q \xmapsto{\alpha} P' \mid Q}\ (\textsc{par}_1) \qquad \frac{P \xmapsto{\surd} P' \quad Q \xmapsto{\mu} Q'}{P \mid Q \xmapsto{\mu} Q'}\ (\textsc{par}_2) \qquad \frac{P_1 \mid P_2 \xmapsto{\mu} P'}{P_2 \mid P_1 \xmapsto{\mu} P'}\ (\textsc{par}_{\textsc{comm}})$$

Below, we briefly comment on the main semantic rules for each term. The term 0 represents the *idle* process and thus has no corresponding semantic rule. The term $\surd$ represents the elementary process that performs action $\surd$ and becomes idle (TICK). The term α represents the elementary process that performs an assignment and terminates (ACT). The sequential composition of two processes is denoted by the term $P;Q$, that represents the process that behaves as P until it

TABLE 3.4: Semantics of expressions.

$$
\begin{aligned}
\boldsymbol{\mathcal{E}}[\![\cdot]\!] &: \mathit{Expr} \longrightarrow \mathcal{I} \to \mathcal{L} \hookrightarrow \mathcal{V} \\
\boldsymbol{\mathcal{E}}[\![v]\!] &= \lambda I\,.\lambda L\,.v \\
\boldsymbol{\mathcal{E}}[\![x]\!] &= \begin{cases} \lambda I\,.\lambda L\,.\,I(x) & \text{if } x \in \mathcal{K}_I \\ \lambda I\,.\lambda L\,.\mathit{value}(L,x) & \text{if } x \in \mathcal{K}_L \end{cases} \\
\boldsymbol{\mathcal{E}}[\![e_1 \diamond e_2]\!] &= \lambda I\,.\lambda L\,.\boldsymbol{\mathcal{E}}[\![e_1]\!](I,L) \diamond \boldsymbol{\mathcal{E}}[\![e_2]\!](I,L) \\
\boldsymbol{\mathcal{E}}[\![e \diamond \bot]\!] &= \boldsymbol{\mathcal{E}}[\![\bot \diamond e]\!] = \lambda I\,.\lambda L\,.\bot
\end{aligned}
\qquad
\begin{aligned}
\boldsymbol{\mathcal{K}}[\![\cdot]\!] &: \mathit{Expr} \longrightarrow 2^{\mathcal{K}_L} \\
\boldsymbol{\mathcal{K}}[\![v]\!] &= \emptyset \\
\boldsymbol{\mathcal{K}}[\![x]\!] &= \begin{cases} \{x\} & \text{if } x \in \mathcal{K}_L \\ \emptyset & \text{otherwise} \end{cases} \\
\boldsymbol{\mathcal{K}}[\![e_1 \diamond e_2]\!] &= \boldsymbol{\mathcal{K}}[\![e_1]\!] \cup \boldsymbol{\mathcal{K}}[\![e_2]\!]
\end{aligned}
$$

terminates (SEQ_1), and if P does terminate and Q performs a transition μ to become Q', then $P; Q$ can perform the same μ-transition and continue as Q' (SEQ_2).

The term $P + Q$ represents a process which can nondeterministically behave either as P (CHOICE-L) or Q (CHOICE-R). The parallel composition of processes is a process $P \mid Q$, where the executions of P and Q are interleaved (PAR_1), and upon termination of one of the parallel components the other continues in isolation (PAR_2). The parallel composition operator is commutative (PAR_{COMM}). We will sometimes consider the n-ary variants of the choice and parallel operators and denote them by $\sum_{i=1}^{n} P_i$ and $\prod_{i=1}^{n} P_i$, respectively. From a theoretical standpoint, this only amounts to assuming that both operators are associative and commutative. From a practical one, using n-ary operators often makes LAbS syntactic terms more compact and readable.

LAbS also supports named process invocation. We assume that there exists a set of *process definitions* $K \triangleq P$, where P is a process term, named K, defined according to the syntax of Table 3.2 that may contain references to K itself and to other process constants; rule (CON) amounts to saying that K can perform the same actions of the process term associated to it. A process named K that contains invocations to itself (or to other named process which in turn invoke K) is *recursive* and may thus describe an infinite behaviour.

Intuitively, the guarded process $g \to P$ can only continue as P if the guard g is satisfied. We will formalise this rule when we introduce the semantics of agents (see Table 3.7) since the evaluation of a guard depends on the state of the agent.

Expressions may contain constants, references to the value of local attributes, or stigmergic keys. A guard may either be the *true* predicate, which is always satisfied, or a comparison between two expressions. Guards can also be negated ($\neg b$) or composed through the conjunction and disjunction operators, $\wedge$ and $\vee$.

The semantics of expressions is formalised by a semantic function $\boldsymbol{\mathcal{E}}[\![\cdot]\!]$ (Table 3.4). We assume that $v \in \mathcal{V}$, $x \in \mathcal{K}_I \cup \mathcal{K}_L$; $\diamond$ and $\bowtie$ are the same as in Table 3.2. We also assume that the

TABLE 3.5: Satisfaction of guards.

$$\begin{array}{lcll}
I,L & \models & \mathit{true} & \\
I,L & \models & \neg g & \iff I,L \vdash \neg g \text{ and } I,L \not\models g \\
I,L & \models & e_1 \bowtie e_2 & \iff I,L \vdash e_1 \text{ and } I,L \vdash e_2 \text{ and } \boldsymbol{\mathcal{E}}[\![e_1]\!](I,L) \bowtie \boldsymbol{\mathcal{E}}[\![e_2]\!](I,L) \\
I,L & \models & g_1 \wedge g_2 & \iff I,L \models g_1 \text{ and } I,L \models g_2 \\
I,L & \models & g_1 \vee g_2 & \iff I,L \models g_1 \text{ or } I,L \models g_2
\end{array}$$

TABLE 3.6: Well-definedness of expressions and guards.

$$\begin{array}{lcll}
I,L & \vdash & v & \\
I,L & \vdash & x & \iff (x \in \mathcal{K}_I \text{ and } I(x) \neq \bot) \text{ or } (x \in \mathcal{K}_L \text{ and } \mathit{value}(L,x) \neq \bot) \\
I,L & \vdash & e_1 \diamond e_2 & \iff I,L \vdash e_1 \text{ and } I,L \vdash e_2 \\
\\
I,L & \vdash & \mathit{true} & \\
I,L & \vdash & \neg g & \iff I,L \vdash g \\
I,L & \vdash & g_1 \wedge g_2 & \iff I,L \vdash g_1 \text{ and } I,L \vdash g_2 \\
I,L & \vdash & g_1 \vee g_2 & \iff I,L \vdash g_1 \text{ and } I,L \vdash g_2
\end{array}$$

equality $\bot = \bot$ holds, while all other relations $\bowtie$ involving $\bot$ never do. We denote with $\boldsymbol{\mathcal{K}}[\![\cdot]\!]$ a function that computes the set of stigmergy keys needed to evaluate an expression. This function is instrumental to formalise the mechanisms of virtual stigmergies. We allow $\boldsymbol{\mathcal{E}}[\![\cdot]\!]$ to return the undefined value $\bot$. For instance, this may happen when the expression refers to an undefined value or applies an operator to incompatible values (e.g. adding a number to a string). With a slight abuse of notation, we will use $\boldsymbol{\mathcal{K}}[\![b]\!]$ to denote the union of $\boldsymbol{\mathcal{K}}[\![e]\!]$ for all sub-expressions of a guard b.

Satisfaction of a guard g is formalised as a relation $I, L \models g$ (Table 3.5). We say that a guard g is *well-defined* with respect to interface I and stigmergy L if all the sub-expressions of g refer to defined attributes and stigmergy keys (this relation is denoted by $\vdash$ in Table 3.6). If g is not well-defined, then it may happen that neither g nor $\neg g$ hold. This means that the law of excluded middle is not generally valid, and this is why, although we have conjunction and negation, we have also introduced an operator for disjunction; $g_1 \vee g_2$ does not have the same meaning as $\neg(\neg g_1 \wedge \neg g_2)$. Well-definedness is not a strict requirement for all types of guards: satisfaction of $g_1 \vee g_2$ only requires at least one of the two sub-guards to hold. By defining disjunction in this way, we allow agents to operate even though their knowledge is partial: in fact, $g_1 \vee g_2 \rightarrow P$ may enable P also when one of the sub-guards is not well-defined.

3.2.2 Link predicates

A *link predicate* is a predicate over the knowledge (i.e., interface and local stigmergy) of two agents, describing the conditions that allow them to communicate. We assume that each stigmergic variable x has an associated link predicate φ_x. When multiple variables occur within the same link predicate φ_s, we say that they belong to the same *virtual stigmergy* s. Two agents are *neighbours* with respect to stigmergy s if they satisfy φ_s. This abstraction is useful, for instance, in the case of multi-robot systems, where predicates allow to effectively model different sensors and capabilities for each robot. Link predicates have the following syntax:

$$\begin{array}{rlr} \varphi ::= & \mathit{true} \mid \eta \bowtie \eta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi & \text{predicate} \\ \eta ::= & v \mid x \mid \eta \diamond \eta \quad x \in \mathcal{K}_I \cup \mathcal{K}_L & \text{expression} \end{array}$$

We denote with $\boldsymbol{\mathcal{H}}[\![\cdot]\!]$ the semantic function of expressions η. We omit a formal definition, as it is nearly identical to the function $\boldsymbol{\mathcal{E}}[\![\cdot]\!]$ described in Table 3.3. The only difference is that $\boldsymbol{\mathcal{H}}[\![\cdot]\!]$ evaluates a predicate against two interfaces and two local stigmergies. Identifiers are decorated with indexes (x_s, x_r) to clarify whether they refer to a variable in the knowledge of the sender or the potential receiver, respectively.

Similarly, the definitions of satisfaction and well-definedness closely follow the ones introduced for guards. The ability of combining link predicates offers an intuitive way to model different communication modes for agents. For instance, the predicate

$$\|pos_s - pos_r\| \leq \delta \;\vee\; (\mathit{LongRange}_s = \text{“}\mathit{true}\text{”} \wedge \mathit{LongRange}_r = \text{“}\mathit{true}\text{”}),$$

where $\|\cdot\|$ denotes the Euclidean norm, states that two agents can communicate if their positions are closer than a constant δ or if they both possess a long-range networking device.

3.2.3 Agents and systems

Agent-level transitions, triggered when an agent performs an action, are modelled in Table 3.7. We assume that $v = \mathit{value}(L, x)$ and $t = \mathit{time}(L, x)$. Rule (SKIP) states that an agent can perform a transition when its behaviour allows a $\surd$-move. According to rule (ATTR) we have that, when an agent performs an attribute update $x \leftarrow e$, the result of expression e is bound to attribute x, and the stigmergy keys used to evaluate e are added to the set Zc of keys to be confirmed.

Stigmergy updates are defined by rule (LSTIG) and result in the insertion of a value in the local stigmergy of the agent. We use $\mathit{tod}()$ to represent the timestamp (obtained from a global clock)

TABLE 3.7: Semantics of agents.

$$\frac{P \overset{\surd}{\mapsto} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I, L, P', Zc, Zp\rangle} \ (\text{SKIP})$$

$$\frac{P \xmapsto{x \leftarrow e} P' \quad \boldsymbol{\mathcal{E}}[\![e]\!](I, L) = v \quad I[x \mapsto v] = I' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L, P', Zc \cup \boldsymbol{\mathcal{K}}[\![e]\!], Zp\rangle} \ (\text{ATTR})$$

$$\frac{P \xmapsto{x \hookleftarrow e} P' \quad \boldsymbol{\mathcal{E}}[\![e]\!](I, L) = v \quad t = tod() \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I, L \oplus (x, v, t), P', Zc \cup \boldsymbol{\mathcal{K}}[\![e]\!], Zp \cup \{x\}\rangle} \ (\text{LSTIG})$$

$$\frac{I, L \models g \quad \langle I, L, P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L', P', Zc', Zp'\rangle}{\langle I, L, g \to P, Zc, Zp\rangle \xrightarrow{\varepsilon} \langle I', L', P', Zc' \cup \boldsymbol{\mathcal{K}}[\![g]\!], Zp'\rangle} \ (\text{AWAIT})$$

for the new value. Since the newly inserted value must be propagated, its key is added to Zp; Zc may also be updated, like for the attribute update case. Rule (AWAIT) specifies that a guarded process $g \to P$ can only proceed if the guard g is satisfied. Notice that, if the guarded process can proceed, the stigmergy keys contained in the guard are added to the set Zc of the agent. The above transitions are labelled ε to denote they are internal to each agent, i.e., they are invisible from the point of view of the system. All agent-level rules are guarded by the condition $Zc = Zp = \emptyset$, meaning that an agent has to propagate or confirm all pending variables before continuing its execution.

System-level transitions formalise the handling of shared knowledge inside the virtual stigmergy and are shown in Table 3.8, where λ denotes a generic transition label. Rule (PAR) simply states that parallel subsystems interleave their internal actions. The symmetrical rule to (PAR) has been omitted. Rules (COMM) and (ASSOC) describe that parallel composition is commutative and associative. Rule (PROPAGATE) states that an agent can always remove a variable from Zp and propagate its value to neighbours. Rule (CONFIRM) specifies that the same can happen with Zc keys. The different nature of the messages is reflected by different transition labels (put for propagation; qry for confirmation). The (PUT) rule allows messages to spread to other agents. When a subsystem performs a $\mathsf{put}(I', L', x, v, t)$ transition, a neighbouring agent (that is, one that satisfies the predicate φ_x together with the sender) with an expired value will update its local stigmergy and add x to the keys to propagate. Notice that x is also removed from Zc, as it is assumed that the new value does not need to be confirmed anymore. Notice also that a composite system evolves by emitting the same transition label as its subsystem. This means that the rule is recursively applied until all neighbours perform their stigmergy update.

The rules for confirmation messages are quite similar, but the actions of agents that receive a confirmation message depend on the current state of their local stigmergy. Rule (QRY$_1$) says that an agent with an older entry will react to a query transition $\mathsf{qry}(I', x, v, t)$ by updating its own

TABLE 3.8: Semantics of systems.

$$\frac{S \xrightarrow{\varepsilon} S'}{S \parallel T \xrightarrow{\varepsilon} S' \parallel T}\ (\textsc{par}) \qquad \frac{S_1 \parallel S_2 \xrightarrow{\lambda} S'}{S_2 \parallel S_1 \xrightarrow{\lambda} S'}\ (\textsc{comm}) \qquad \frac{(S_1 \parallel S_2) \parallel S_3 \xrightarrow{\lambda} S'}{S_1 \parallel (S_2 \parallel S_3) \xrightarrow{\lambda} S'}\ (\textsc{assoc})$$

$$\frac{x \in Zp \qquad L(x) = (v, t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{put}(I,L,x,v,t)} \langle I, L, P, Zc, Zp \setminus \{x\}\rangle}\ (\textsc{propagate})$$

$$\frac{x \in Zc \quad L(x) = (v, t)}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I,L,x,v,t)} \langle I, L, P, Zc \setminus \{x\}, Zp\rangle}\ (\textsc{confirm})$$

$$\frac{S \xrightarrow{\mathsf{put}(I',L',x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad L \oplus (x, v, t) \neq L}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{put}(I',L',x,v,t)} S' \parallel \langle I, L \oplus (x, v, t), P, Zc \setminus \{x\}, Zp \cup \{x\}\rangle}\ (\textsc{put})$$

$$\frac{S \xrightarrow{\mathsf{qry}(I',L'\,x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad time(L, x) < t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \parallel \langle I, L \oplus (x, v, t), P, Zc\backslash\{x\}, Zp \cup \{x\}\rangle}\ (\textsc{qry}_1)$$

$$\frac{S \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \quad I', L', I, L \models \varphi_x \quad time(L, x) \geq t}{S \parallel \langle I, L, P, Zc, Zp\rangle \xrightarrow{\mathsf{qry}(I',L',x,v,t)} S' \parallel \langle I, L, P, Zc, Zp \cup \{x\}\rangle}\ (\textsc{qry}_2)$$

stigmergy and propagating the value afterwards. On the other hand, an agent that has a more up-to-date value will just update Zp to propagate it, while discarding the received entry (rule QRY$_2$).

3.2.4 Tuples and atomic assignments

We will sometimes use compound assignments of the form $x_1, \ldots, x_n \leftarrow e_1, \ldots, e_n$. In this case, all expressions are evaluated over the current state of the agent, and the corresponding values $v_1, \ldots, v_n$ are then assigned atomically to variables $x_1, \ldots, x_n$. When performing a compound stigmergic assignment $x_1, \ldots, x_n \leftsquigarrow e_1, \ldots, e_n$, all values receive the same timestamp. Multiple assignments to variables of different kinds (e.g., an internal one and a shared one) are not allowed.

We further enrich our language with the notion of *stigmergy tuples*. Tuples are disjoint sets of stigmergic variables; all variables within the same tuple belong to the same virtual stigmergy. A tuple acts as a single stigmergic variable: whenever any element of a tuple can be propagated or requested, the rest of the tuple will be as well, in an atomic fashion. Thus, these communication steps cannot be interleaved with any other action.

These design choices are driven by the need to restrict the interleaving of agents, by giving the user of the language a natural way to express operations that should always be performed together.

Similarly, stigmergic tuples allow to store complex data in virtual stigmergies, with the guarantee that such data will be propagated atomically.

3.2.5 Clocks and verification

As stated in Section 3.2.3, the semantics of LAbS are based on the assumption that agents can always retrieve a (unique) timestamp from a global clock. Using a distributed clock would be a more realistic option for the actual implementation of multi-agent systems, and is the solution adopted by languages such as Buzz [196]. However, our language has an important difference in that its main focus is on formal verification, rather than execution on real or simulated platforms.

From a verification standpoint, the purpose of a clock is simply to enforce a total order on the transitions by assigning them a unique identifier (i.e., the timestamp). Using a distributed clock would unnecessarily inflate the state space with intermediate transitions for distributed time-stamping. Instead, we simply compute timestamps by relying on a global counter, which we increase each time a new transition takes place.

Please note that, due to interleaving, the above mechanism does not lose any feasible ordering of events with respect to a distributed schema. In fact, it captures all the total orders that would be possible with distributed clocks. In contrast, multiple (partial) orderings under the distributed clock can correspond to the same total order under the global clock. As a consequence, using a global clock helps to maintain a more compact state space, which is very desirable for verification.

3.3 Modelling the environment

Agents are mobile entities that are *situated* and operate in a physical environment. This agent-environment interaction is a fundamental feature of many real-world scenarios [242]. This kind of interaction enjoys some specific properties that are difficult to express with the constructs introduced in Section 3.2. In this section we extend the language to support a shared-memory abstraction of the environment. We allow environment variables to occur in expressions and guards, and we update the formal semantics of situated systems so that agents can atomically perform read and write operations on the environment.

3.3.1 Semantics of situated systems

Assuming that there is a set $\mathcal{K}_E$ disjoint from $\mathcal{K}_I$ and $\mathcal{K}_L$, we define an *environment* to be a partial function from $\mathcal{K}_E$ to the set of values $\mathcal{V}$. A *situated system* is a pair (E, S), where E is an environment and S is a LAbS system. We now allow expressions to also contain identifiers from

TABLE 3.9: Basic processes in situated systems.

$$\alpha ::= \quad x \leftarrow e \mid x \leftsquigarrow e \mid x \Leftarrow e \mid \surd$$

$\mathcal{K}_E$. This requires introducing a new semantic function, which extends the one in Table 3.4 as follows:

$$\begin{aligned}
\boldsymbol{\mathcal{E}}_2[\![\cdot]\!] &: \mathit{Expr} \longrightarrow \mathcal{E}nv \rightarrow \mathcal{I} \rightarrow \mathcal{L} \hookrightarrow \mathcal{V} \\
\boldsymbol{\mathcal{E}}_2[\![v]\!] &= \lambda\, E\, .\lambda\, I\, .\lambda\, L\, .v \\
\boldsymbol{\mathcal{E}}_2[\![x]\!] &= \begin{cases} \lambda\, E\, .\boldsymbol{\mathcal{E}}_1[\![x]\!] & \text{if } x \in \mathcal{K}_E \\ \lambda\, E\, .\boldsymbol{\mathcal{E}}[\![x]\!] & \text{otherwise} \end{cases} \\
&\quad \text{where } \boldsymbol{\mathcal{E}}_1[\![x]\!] = \lambda\, I\, .\lambda\, L\, .\, E(x) \\
\boldsymbol{\mathcal{E}}_2[\![e_1 \diamond e_2]\!] &= \lambda\, E\, .\lambda\, I\, .\lambda\, L\, .\boldsymbol{\mathcal{E}}_2[\![e_1]\!](E, I, L) \diamond \boldsymbol{\mathcal{E}}_2[\![e_2]\!](E, I, L)
\end{aligned}$$

In the definition above, we have used $\mathcal{E}nv$ to denote the set of all environments. Agents are now able to perform an additional basic action to store the result of an expression into an environment variable. We denote this action by $x \Leftarrow e$ (Table 3.9). Since any expression may now potentially refer to environmental variables, their evaluation can no longer be done at the individual level. We therefore revise the semantics of agents and add a transition label $e \triangleright x$ denoting the willingness of an agent to assign the value of expression e to variable x (Table 3.10). Note that we omit Rule (SKIP) as it is the same as in Table 3.7.

To define the semantics of situated systems, in Table 3.11 we introduce an unlabelled transition relation ($\rightarrowtail$). As mentioned above, the evaluation of expressions and the assignment to the relevant store of variables is described by rules (EVAL$_{\mathrm{I,L,E}}$). Rule (AWAIT) has also been removed from agent-level rules, as guards may now also refer to environmental variables.

Rule (MSG) simply states that the actions related to stigmergic communications only affect the system and leave the environment unchanged. Finally, rule (PAR$_{\mathrm{E}}$), which is commutative, states that the parallel composition of two systems affects the environment in an interleaved fashion. The rule symmetrical to (PAR$_{\mathrm{E}}$) is omitted.

3.4 Examples

In this section we specify several examples of multi-agent systems. We use a machine-readable specification language that extends LAbS with constructs that allow to compactly declare the composition of a system and its initial state. We call this language LAbS$^+$, to avoid confusion with the core process algebra LAbS. We now provide an informal description of the language:

TABLE 3.10: Semantics of agents in a situated system.

$$\frac{P \xmapsto{x \leftarrow e} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \boldsymbol{\mathcal{K}}[\![e]\!], Zp\rangle}\ (\textsc{attr})$$

$$\frac{P \xmapsto{x \leftsquigarrow e} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \boldsymbol{\mathcal{K}}[\![e]\!], Zp\rangle}\ (\textsc{lstig})$$

$$\frac{P \xmapsto{x \Leftarrow e} P' \quad Zc = Zp = \emptyset}{\langle I, L, P, Zc, Zp\rangle \xrightarrow{e \triangleright x} \langle I, L, P', Zc \cup \boldsymbol{\mathcal{K}}[\![e]\!], Zp\rangle}\ (\textsc{env})$$

TABLE 3.11: Semantics of situated systems.

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \boldsymbol{\mathcal{E}}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_I}{(E, a) \rightarrowtail (E, \langle I[x \mapsto v], L, P, Zc, Zp\rangle)}\ (\textsc{eval}_\mathrm{I})$$

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \boldsymbol{\mathcal{E}}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_L}{(E, a) \rightarrowtail (E, \langle I, L \oplus (x, v, tod()), P, Zc, Zp \cup \{x\}\rangle)}\ (\textsc{eval}_\mathrm{L})$$

$$\frac{a \xrightarrow{e \triangleright x} \langle I, L, P, Zc, Zp\rangle \quad \boldsymbol{\mathcal{E}}_2[\![e]\!](E, I, L) = v \neq \bot \quad x \in \mathcal{K}_E}{(E, a) \rightarrowtail (E[x \mapsto v], \langle I, L, P, Zc, Zp\rangle)}\ (\textsc{eval}_\mathrm{E})$$

$$\frac{(E, \langle I, L, P, Zc, Zp\rangle) \rightarrowtail (E', \langle I', L', P', Zc', Zp'\rangle) \quad E, I, L \models g}{(E, \langle I, L, g \to P, Zc, Zp\rangle) \rightarrowtail (E', \langle I', L', P', Zc' \cup \boldsymbol{\mathcal{K}}[\![g]\!], Zp'\rangle)}\ (\textsc{await})$$

$$\frac{S \xrightarrow{\mu(I,x,v,t)} S'}{(E, S) \rightarrowtail (E, S')}\ (\textsc{msg}) \qquad \frac{(E, S) \rightarrowtail (E', S')}{(E, S \| T) \rightarrowtail (E', S' \| T)}\ (\textsc{par}_\mathrm{E})$$

Section 3.5 contains a more thorough definition. Table 3.12 shows the LAbS$^+$ syntax for process terms, and compares it to the LAbS one introduced in Table 3.2. A LAbS$^+$ specification consists of the following elements:

- A *global section* that contains the declaration of environment variables, specifies the composition of the system, and possibly the declaration of external parameters that can be set by the user when performing analysis on the system. A newly declared variable can be initialised via an assignment or left undefined by using the keyword `undef`. In addition to deterministic assignment, a nondeterministic assignment from either a range or a set of values is possible.

TABLE 3.12: Comparison between LAbS and LAbS$^+$ syntaxes for process terms.

$\surd$	$x \leftarrow e$	$x \leftsquigarrow e$	$x \Leftarrow e$	$P;Q$	$P+Q$	$P \mid Q$	$g \rightarrow P$
`Skip`	`x <- e`	`x <~ e`	`x <-- e`	`P ; Q`	`P ++ Q`	`P || Q`	`g -> P`

- A (possibly empty) set of *stigmergy declarations*, each containing a link predicate and a sequence of variables. All stigmergic interactions over variables of a given stigmergy are constrained by their link predicate. It might be convenient to group different stigmergic variables that form composite data. A typical example would be a position within an n-dimensional space. To do that, the user can define a *tuple* by declaring a sequence of comma-separated stigmergic variables. A tuple is considered as a single stigmergy entry: the stigmergic values from the same tuple are always propagated and confirmed together.
- A non-empty set of *agent declarations*, one for each different *kind* of agent. Each declaration contains the attributes of the agent and its behavioural specifications. The list of virtual stigmergies that the agent manipulates is explicitly declared in this section too.
- The list of the *properties* of interest for the specified system. A property has a temporal *modality* followed by a *predicate*. The modality is either `always` or `eventually`, corresponding to the LTL operators $\Box$ and $\Diamond$, respectively. The predicate ranges over the attributes and the stigmergic variables of the agents, and over the environment variables. A basic predicate is a comparison between two arithmetic expressions over said variables. Composite predicates can be obtained using the usual Boolean operators. Existential or universal quantification over the sets of agents of any given kind is also allowed within the predicate.

LAbS$^+$ supports one-dimensional *arrays*-.[1] Formally, we treat an array of length n as a collection of n variables; specifications that allow an out-of-bound array access are invalid. The elements of a stigmergic array are assumed to be tupled together. This language also introduces several extensions to the syntax of arithmetic expressions. The read-only variable `id` evaluates to a natural number strictly smaller than the number of agents in the system, and the evaluated value is unique for each agent. An expression may also contain calls to the following built-in functions: `max(e1, e2)` and `min(e1, e2)` (respectively, the maximum and the minimum between the values obtained by computing expressions `e1`, `e2`), and `abs(e)` (the absolute value of expression `e`).

3.4.1 Dining philosophers

The specification of Figure 3.1a describes the *dining philosophers* scenario (Example 2.1). The system is parameterised in a value `_n` (line 2), which determines the number of agents in the

[1]Higher-dimensional arrays are not currently supported, but may be added in a future revision of the language.

system (line 4: in LAbS$^+$, the `spawn` keyword introduces the composition of the system, i.e., the number of agents that it contains). The environment features an array `forks` whose elements are all initialised to 0 (line 3). Each element of the array models a fork: a value 0 means that the fork is available, while a value 1 means that it is currently held by one of the agents. The (recursive) behaviour of the philosophers is specified at lines 9–20. Each agent repeatedly tries to acquire two forks, by checking and updating the elements `id` and `(id+1)%_n` of the array `forks`. After acquiring both forks, the agent releases them and starts over. Each agent maintains an internal variable `status`, initially set to 0, which describes its current situation (line 8). When `status` is set to either 0, 1, or 2, it denotes the number of forks currently held by the agent. When `status` is set to 3, it means that the agent has just released one fork and is going to release the other one during its next action. Lastly, invariant `NoDeadlock` (lines 24–26) states that the system should never reach a state where all agents are waiting for the second fork.

3.4.2 Leader election

Figure 3.1b contains a simple *leader election* system (Example 2.2): all the agents repeatedly update a stigmergic variable `leader` with their own `id`, but they stop doing so as soon as they detect that the variable already stores a value lower than their identifier. Thus, they eventually agree to choose the agent with `id = 0` as their leader.

Lines 1–4 specify that `_n` is an external parameter, and that the system is composed of `_n` agents of kind `Node`. Lines 6–9 define a stigmergy `Election` containing a single variable `leader`. Its `link` predicate is simply `true`, so any two agents may communicate at any time. The stigmergic variable `leader` is initially set to the value of `_n`. The definition of `Node` agents states that they can access the `Election` stigmergy (line 12). Their behaviour (lines 13–16) simply tells them to repeatedly set `leader` to their own `id` as long as that variable contains a greater value. Finally, property `LeaderIs0` (lines 20–22) specifies that the system should eventually reach a state where all `Node` agents agree on a value of `0` for variable `leader`.

3.4.3 Flocking

We now describe two systems of flocking agents that aim at replicating the *Boids* rules described in Example 2.3 by means of appropriate virtual stigmergies.

A simple model of flocking behaviour. As a first step, we describe how to specify the *alignment* mechanism. The LAbS$^+$ specification of Table 3.2 models a flock of `_n` agents distributed on a square grid, or *arena*, of `_size` × `_size` locations. `_n` and `_size` are both external parameters: the third parameter `_delta` encodes the *visibility range* of the agents. The position

```
system {
  extern = _n
  environment = fork[_n]: 0
  spawn = Phil: _n
}

agent Phil {
  interface = status: 0
  Behaviour =
    fork[id] = 0 ->
      fork[id] <-- 1;
      status <- 1;
      fork[(id+1) % _n] = 0 ->
        fork[(id+1) % _n] <-- 1;
        status <- 2;
        fork[(id+1) % _n] <-- 0;
        status <- 3;
        fork[id] <-- 0;
        status <- 0;
        Behaviour
}

check {
  NoDeadlock =
    always exists Phil p,
      status of p != 1
}
```

(A) Dining philosophers.

```
system {
  extern = _n
  spawn = Node: _n
}

stigmergy Election {
  link = true
  leader: _n
}

agent Node {
  stigmergies = Election
  Behaviour =
    leader > id ->
      leader <~ id;
      Behaviour
}

check {
  LeaderIs0 =
    finally forall Node a,
      leader of a = 0
}
```

(B) Leader election.

FIGURE 3.1: Two example systems in LAbS$^+$.

of each agent is stored as a pair of attributes `x`, `y`, and may be initialised to any location in the arena (line 16). Its direction is instead recorded by the stigmergic tuple $\langle$`dirX`, `dirY`$\rangle$, with both elements initially set to either 1 or -1 (line 11). The `Behaviour` process (line 18) states that every agent simply repeatedly moves on the arena one step at a time. The `Move` process (line 19) describes a single step, which amounts to a sum between the position attributes and the direction variables. Note that the arena wraps around: for instance, an agent at location $(x,$ `_size` $-\ 1)$ that moves in direction $(1, 1)$ will reach $(x + 1, 0)$. This explains the modulo operators in `Move`.

Due to their behavioural specification, agents must repeatedly access the values of $\langle$`dirX`, `dirY`$\rangle$: therefore, they will also repeatedly send confirmation messages to verify that these values are up-to-date. The link predicate at lines 7–9 encodes that the Euclidean distance between the sender and the receiver should be at most `_delta`. Thus, whenever two agents happen to be close enough, they may agree on the direction with the most recent timestamp (see Figure 3.3 for a graphical representation). Property `Consensus` (lines 23–25) states that all the agents should eventually move in the same direction.

Adding the *cohesion* mechanism. Figure 3.4 shows a slightly more complex system, where `Bird` agents implement both an *alignment* and a *cohesion* mechanism. The global section and the `Alignment` stigmergy declaration are omitted, as they are exactly the same as those of Figure 3.2. To achieve cohesion, the agents maintain some information about the *group* of agents

```
system {
  extern = _n, _size, _delta
  spawn = Bird: _n
}

stigmergy Alignment {
  link =
    ((x of 1 - x of 2) * (x of 1 - x of 2)) +
    ((y of 1 - y of 2) * (y of 1 - y of 2)) <= _delta * _delta

  dirX, dirY: [-1, 1], [-1, 1]

}

agent Bird {
  interface = x: 0.._size; y: 0.._size
  stigmergies = Alignment
  Behaviour = Move; Behaviour
  Move = x, y <- (x + dirX) % _size, (y + dirY) % _size
}

check {
  Consensus =
    finally forall Bird b1, forall Bird b2,
      dirX of b1 = dirX of b2 and dirY of b1 = dirY of b2
}
```

FIGURE 3.2: A *Boids*-like system with only the *alignment* rule.

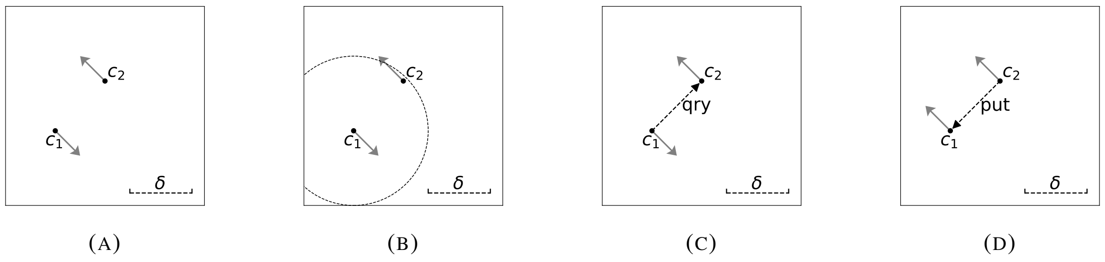


FIGURE 3.3: Two agents c_1, c_2 agree on a direction of movement.

they belong to. This information is stored as a tuple within the `Cohesion` stigmergy (lines 3–7). Each group has a *leader* and zero or more *followers*. The leader registers its `id` and position in the variables `leader`, `posX`, and `posY`. The last variable in the tuple, `count`, is a counter that is repeatedly updated by the followers. In the initial state, each boid is the leader of its own group, and `count` is set to 1. Thanks to the link predicate (line 4), a boid can decide to join a larger group than the one it is actually part of. When this happens, the new follower increases the group counter and moves towards its new leader, if they are far apart (Figure 3.5). These actions are specified by the `Attract` process (lines 16–30), which is repeatedly performed by each agent after each movement they make. Lastly, property `OneLeader` (lines 34–36) specifies that all the agents should eventually have the same leader.

```
system { ... }

stigmergy Cohesion {
  link = (count of 1 >= count of 2)

  leader, posX, posY, count: id, -1, -1, 1
}

stigmergy Alignment { ... }

agent Bird {
  interface = x: 0.._size; y: 0.._size
  stigmergies = Cohesion; Alignment
  Behaviour = Move; Attract; Behaviour
  Move = x, y <- (x + dirX) % _size, (y + dirY) % _size
  Attract = (
    (leader = id -> posX, posY <~ x, y)
    ++
    (leader != id ->
      count <~ count + 1; (
        (abs(x - posX) > _delta -> dirX <~ (posX-x) / abs(posX-x))
        ++
        (abs(x - posX) <= _delta -> Skip)
      ); (
        (abs(y - posY) > _delta -> dirY <~ (posY-y) / abs(posY-y))
        ++
        (abs(y - posY) <= _delta -> Skip)
      )
    )
  )
}

check {
  OneLeader =
    finally forall Bird b1, forall Bird b2,
      leader of b1 = leader of b2
}
```

FIGURE 3.4: A *Boids*-like system with *alignment* and *cohesion* rules.

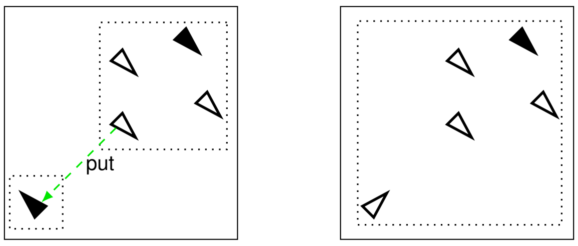


FIGURE 3.5: The *cohesion* rule in action. Each dotted box contains a group of agents. Filled shapes denote leaders, while the followers are outlined. One of the followers of the larger group propagates the *cohesion* tuple $(leader, count, pos_x, pos_y)$ to the lonely leader in the bottom left, which then becomes a follower and points toward its new leader.

3.4.4 Line formation

The specification in Figure 3.6 calls back to Example 2.4: it describes a system of `_n` robots on a horizontal segment of length `_size`. They must move so that their distance from each other is eventually not less than `_range`. Each robot starts from a nondeterministic position along the segment, stored in the `pos` attribute. We only assume that no agent starts on one of the endpoints (i.e., with a value of either 0 or `_size` $- 1$ in `pos`). To achieve their goal, all agents repeatedly

```
1   system {
2     extern = _range, _n, _size
3     spawn = Robot: _n
4   }
5
6   stigmergy Left {
7     link =
8       pos of 1 - pos of 2 >= 0 and
9       pos of 1 - pos of 2 <= _range
10
11    idLeft: undef
12  }
13
14  stigmergy Right {
15    link =
16      pos of 2 - pos of 1 > 0 and
17      pos of 2 - pos of 1 <= _range
18
19    idRight: undef
20  }
21
22  agent Robot {
23    interface = pos: 1.._size
24    stigmergies = Left; Right
25    Behaviour =
26      idLeft, idRight <~ id, id;
27      (
28        (idLeft != id ->
29            (pos > 0 -> pos <- pos - 1)
30            ++
31            (pos = 0 -> pos <- pos + 1))
32        ++
33        (idRight != id ->
34            (pos < _size - 1 -> pos <- pos + 1)
35            ++
36            (pos = _size - 1 -> pos <- pos - 1))
37        ++
38        (idRight = id and idLeft = id -> Skip)
39      ); Behaviour
40
41  }
42
43  check {
44    Distancing =
45      finally forall Robot a, forall Robot b,
46        id of a = id of b or
47        abs(pos of a - pos of b) >= _range
48  }
```

FIGURE 3.6: A system of line-forming agents.

write their `id` on two stigmergic variables `idLeft`, `idRight` (line 26). These variables reside in stigmergies `Left` and `Right`, respectively (lines 6–20). The link predicates of `Left` and `Right` are similar, but inverted: the predicate of `Left` states that the receiver must be to the left of the sender (i.e., its position must have a lower value), while for that of `Right` the sender must be to the left of the receiver. In both cases, the distance between the two must be at most `_range`. Thus, when an agent notices that `idLeft` is different from its own `id`, it makes a step to the right (and vice versa), because the value must have been set by another robot that is too close. However, if the robot already is at one of the endpoints of the segment, it moves towards the other robot instead (lines 28–38). The `Distancing` property defined at lines 44–47 states that eventually all robots are at no less than `_range` steps from each other.

3.4.5 Majority protocols

We now introduce two majority protocols (see Example 2.5) taken from the population protocols literature, and show how to encode them with LAbS$^+$. The overall structure of these examples can be adapted to other population protocols, potentially with a higher number of states and transitions.

Approximate majority. The system of Figure 3.7 models an approximate majority population protocol [13]. Each agent has an initial opinion, encoded in attribute `state` as either 0 (N) or 1 (Y): external parameters `_no` and `_yes` represent the number of agents with initial opinion set to 0 or 1, respectively (line 2). In this protocol, an agents that meets another with a different opinion assumes a *blank* opinion, encoded as the value 2. An agent with a blank opinion, in turn, will imitate the opinion of the next agent it meets.

To model this protocol, we exploit the environment as a communication medium. Namely, an agent may *initiate* an interaction by writing its own `id` and `state` into the environment variables `initiator` and `message` (line 6). Alternatively (lines 8–13), it may *respond* to another agent by reading these values and update its own state accordingly. Assuming that `_no` $>$ `_yes`, a safety property of interest is that the system never reaches a state where all agents agree on Y. This property is encoded as `NoYConsensus` (lines 29–31)

A correct majority protocol. Figure 3.8 describes a 4-state protocol where all agents are initially either in state Y or in state N. When a Y-agent[2] meets a N-agent, they change their states to y and n, respectively. A n-agent changes its state to y upon meeting either a Y- or a y-agent. On the other hand, a y-agent only changes its state to n upon meeting a N-agent. This difference acts as a tie-breaker: a system that is initially tied will eventually reach a consensus on y. It can be proved that this protocol successfully computes the majority. To be more precise, if the initial number of Y-agents is greater than or equal to the number of N-agents, the system eventually contains only Y- and y-agents. Otherwise, it stabilises to a configuration that contains only N's and n's [16].

This specification is slightly more complex than the previous one, as interactions may affect the state of both the initiator and the responder. We encode this by letting the responder store its state within an environment variable: the initiator will check this variable and update its own state accordingly. To preserve the semantics of population protocols and avoid unwanted interactions, we introduce a shared variable `lock` that every agent has to check before performing an action. Our language does not require specific primitives for locks, thanks to the possibility of performing compound assignments and the atomicity of guarded processes. Throughout the specification,

[2]For simplicity, we use the term *x-agent* to refer to an agent which is in state x

```
system {
  extern = _yes, _no
  environment = initiator: undef; message: undef
  spawn = Yes: _yes, No: _no
  Protocol = (
    (state != 2 -> initiator, message <-- id, state)
    ++
    (initiator != id ->
      (message = 1 and state = 2 -> state <- 1) ++
      (message = 1 and state = 0 -> state <- 2) ++
      (message = 0 and state = 1 -> state <- 2) ++
      (message = 0 and state = 2 -> state <- 0) ++
      (message = state -> Skip)
    )
  ); Protocol
}

agent Yes {
  interface = state: 1
  Behaviour = Protocol
}

agent No {
  interface = state: 0
  Behaviour = Protocol
}

check {
  NoYConsensus =
    always exists Yes y, exists No n,
    state of y != 1 or state of n != 1
}
```

FIGURE 3.7: A population protocol for approximate majority.

states are encoded as $0 = N$, $1 = Y$, $2 = n$, $3 = y$. When `lock` is set to 0, agents can only initiate a transition by writing its own `id` and state to the `initiator` and `message` variables, and setting `lock` to 1 (lines 10–12). This value signals the other agents that a transition has been initiated and that they are free to respond (lines 13–25). In three cases, the responder only needs to update its own state and reset the shared variables to allow a new transition to take place (lines 20–22). In the case of a YN-transition, instead, the responder also stores N in the environment variable `responder` and sets `lock` to 2 (lines 14–16). At this point, the initiator is the only agent that can perform an action: namely, it sets its own state to y and increases *lock* to 3 (lines 6–7). This value signals the responder that the transition has been fully performed, so it can safely reset the environment variables to enable new transitions (line 17).

3.5 Syntax of the LAbS$^+$ specification language

This section describes the LAbS$^+$ specification language in detail.

```
system {
  extern = _yes, _no
  environment = initiator: -1; message: -1; responder: -1; lock: 0
  spawn = Yes: _yes, No: _no
  Protocol = (
      (initiator = id and lock = 2 ->
        (state = 1 and responder = 0 -> state <- 3; lock <-- 3)
      ) ++
      (initiator != id ->
          (lock = 0 ->
            (state != 2 -> initiator, message, lock <-- id, state, 1)
          ) ++
          (lock = 1 ->
            (message = 1 and state = 0 ->
              lock, responder <-- 2, state;
              state <- 2;
              (lock = 3) -> initiator, responder, lock <-- -1, -1, 0
            ) ++
            ((
                (message = 1 and state = 2 -> state <- 3) ++
                (message = 0 and state = 3 -> state <- 2) ++
                (message = 3 and state = 2 -> state <- 3)
              );
              initiator, lock <-- -1, 0
            )
        ))); Protocol
}

agent Yes {
  interface = state: 1

  Behaviour = Protocol
}

agent No {
  interface = state: 0

  Behaviour = Protocol
}

check {
  NoYConsensus =
    always exists Yes y, exists No n,
    (state of y = 0 or state of y = 2)
    or
    (state of n = 0 or state of n = 2)
}
```

FIGURE 3.8: A correct population protocol to compute the majority opinion.

3.5.1 Basic syntactic elements

Variable names, identifiers, literals. *Variable names* identify local, stigmergic, and environment variables. They are formed by a lowercase letter, possibly followed by a sequence of letters, numbers, or underscores. *Keywords*, listed in Table 3.13, may not be used as variable names. *Identifiers* are formed by an uppercase letter and zero or more letters, numbers, or underscores. The string `Skip` may not be used as an identifier, as it denotes the $\surd$ action. A *numeric literal* is a sequence of digits, optionally preceded by a sign. We use $\mathbb{V}$, $\mathbb{ID}$, and $\mathbb{NUM}$ to denote the sets of variable names, identifiers, and numeric literals, respectively.

TABLE 3.13: LAbS$^+$ keywords.

`abs`	`extern`	`link`	`stigmergies`
`agent`	`false`	`max`	`stigmergy`
`always`	`finally`	`min`	`system`
`and`	`forall`	`of`	`true`
`environment`	`id`	`or`	`undef`
`exists`	`interface`	`spawn`	

BNF conventions. The syntax of LAbS$^+$ is defined in BNF notation [18]. We use *italics* and `typewriter` fonts for nonterminal and terminal symbols, respectively. Furthermore, we use the following notation to keep our syntax definitions compact:

- $\langle x \rangle?$ means that term x is optional.
- $\langle x \rangle*$ means that term x may occur zero or more times.
- $\langle x \rangle+$ means that term x may occur one or more times: it is equivalent to $x\langle x \rangle*$.
- $\langle x_1, \ldots \rangle$ denotes either a single occurrence of x, or multiple occurrences separated by commas: it is equivalent to $x\langle \texttt{,}\, x \rangle*$.
- $\langle x_1; \ldots \rangle$ has the same meaning as $\langle x_1, \ldots \rangle$, except that occurrences of x are separated by semicolons.

References, expressions, processes. A *reference* is a variable name, possibly followed by an *index* between square brackets. It is forbidden to put an index in a reference to a non-array variable. On the contrary, references to array variables must always feature an index. The set of references is denoted by $\mathit{ref}^{\mathbb{V}}$. Arithmetic and Boolean *expressions* over references are respectively denoted by $\mathit{expr}^{\mathbb{V}}$ and $\mathit{bexpr}^{\mathbb{V}}$. The syntax of references and expressions is shown in Table 3.14a. An arithmetic expression may contain the binary operators `+` (sum), `*` (multiplication), `/` (integer division), and `%` (modulo), and the unary operator `abs` (absolute value). The `-` token denotes both the binary subtraction and the unary negation operators. LAbS$^+$ provides two additional arithmetic functions, `max` and `min`, that respectively evaluate to the maximum and the minimum of their arguments. A Boolean expression may be either a literal (`true` or `false`), a logical negation of another expression (denoted by `!`), a conjunction or disjunction of two sub-expressions, or a comparison between two arithmetic expressions. Finally, Table 3.14b shows the LAbS$^+$ syntax for *processes*. The precedence of operators, from highest to lowest, is the following: (1) guarding; (2) sequential composition; (3) nondeterministic choice; (4) parallel composition. Compound assignments (see Section 3.2.4) are supported, provided that

TABLE 3.14: LAbS⁺ syntax (1).

(A) References and expressions.

$$
\begin{array}{rcl}
\mathit{ref}^{\mathbb{V}} & ::= & \mathbb{V}\ \langle\ \texttt{[}\ \mathit{expr}^{\mathbb{V}}\ \texttt{]}\ \rangle? \\
\mathit{expr}^{\mathbb{V}} & ::= & \mathit{ref}^{\mathbb{V}} \mid \mathbb{NUM} \\
 & \mid & \mathit{expr}^{\mathbb{V}}\ \mathit{op}\ \mathit{expr}^{\mathbb{V}} \\
 & \mid & \texttt{-}\mathit{expr}^{\mathbb{V}} \\
 & \mid & \texttt{abs}\,\texttt{(}\mathit{expr}^{\mathbb{V}}\texttt{)} \\
 & \mid & \mathit{fn}\,\texttt{(}\mathit{expr}^{\mathbb{V}}\texttt{,}\ \mathit{expr}^{\mathbb{V}}\texttt{)} \\
 & \mid & \texttt{(}\ \mathit{expr}^{\mathbb{V}}\ \texttt{)} \\
\mathit{bexpr}^{\mathbb{V}} & ::= & \texttt{true} \\
 & \mid & \texttt{false} \\
 & \mid & \texttt{!}\ \mathit{bexpr}^{\mathbb{V}} \\
 & \mid & \mathit{bexpr}^{\mathbb{V}}\ \mathit{bop}\ \mathit{bexpr}^{\mathbb{V}} \\
 & \mid & \mathit{expr}^{\mathbb{V}}\ \mathit{cmp}\ \mathit{expr}^{\mathbb{V}} \\
 & \mid & \texttt{(}\ \mathit{bexpr}^{\mathbb{V}}\ \texttt{)} \\
\mathit{op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
\mathit{cmp} & ::= & \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{!=} \\
\mathit{fn} & ::= & \texttt{max} \mid \texttt{min} \\
\mathit{bop} & ::= & \texttt{and} \mid \texttt{or}
\end{array}
$$

(B) Processes, actions, and variable declarations.

$$
\begin{array}{rcl}
\mathit{proc} & ::= & \mathit{act} \\
 & \mid & \mathit{proc}\ \texttt{;}\ \mathit{proc} \\
 & \mid & \mathit{proc}\ \texttt{++}\ \mathit{proc} \\
 & \mid & \mathit{proc}\ \texttt{||}\ \mathit{proc} \\
 & \mid & \mathit{bexpr}^{\mathbb{V}}\ \texttt{->}\ \mathit{proc} \\
 & \mid & \texttt{(}\ \mathit{proc}\ \texttt{)} \\
 & \mid & \mathbb{ID} \\
\mathit{act} & ::= & \texttt{Skip} \\
 & \mid & \langle \mathit{ref}_1^{\mathbb{V}}, \ldots \rangle\ \mathit{asgn}\ \langle \mathit{expr}_1^{\mathbb{V}}, \ldots \rangle \\
\mathit{asgn} & ::= & \texttt{<-} \mid \texttt{<\~{}} \mid \texttt{<--} \\
 & & \\
\mathit{decl} & ::= & \mathbb{V}\langle\ \texttt{[}\ \mathbb{NUM}\ \texttt{]}\ \rangle?\ \texttt{:}\ \mathit{init} \\
\mathit{init} & ::= & \texttt{undef} \\
 & \mid & \mathbb{NUM} \\
 & \mid & \texttt{\{}\ \langle \mathbb{NUM}_1, \ldots \rangle\ \texttt{\}} \\
 & \mid & \mathbb{NUM}\ \texttt{..}\ \mathbb{NUM}
\end{array}
$$

(a) all referenced variables match the assignment operator, and (b) the number of references on the left-hand side matches the number of expressions on the right-hand side.

Variable declaration and initialisation. A variable declaration *decl* is composed of a variable name, an optional numeric literal enclosed in square brackets, and an *initialiser*, which specifies the possible values of the variable in the initial state of the system. If a numeric literal `[n]` is provided, the variable is an *array* of length n and the initialiser applies to all its elements. The length must be strictly positive.

The syntax of *decl* is shown in Table 3.14b. The `undef` initialiser leaves the variable in an undefined state ($\perp$). If the initialiser is a single numeric literal, the variable is initialised to the corresponding value. A *set* initialiser is a sequence of comma-separated numeric literals between curly braces. A variable with a set initialiser may initially assume any of the provided values. Lastly, the *range* initialiser is written `m..n`, where `m` and `n` are numeric literals with $\texttt{m} < \texttt{n}$, and corresponds to the set initialiser `{m, m+1, ..., n-1}`. Notice that the upper bound `n` is not included in the range.

3.5.2 Syntax of a LAbS⁺ specification

As stated informally in Section 3.4, a specification has the following structure: a global section, zero or more stigmergy declarations, one or more agent declarations, and a property section

TABLE 3.15: LAbS⁺ syntax (2).

(A) A LAbS⁺ specification.

$$\mathit{spec} ::= \mathit{global}\ \langle\ \mathit{stigmergy}\ \rangle{*}\ \langle\ \mathit{agent}\ \rangle{+}\ \mathit{check}$$

(B) Global section.

$$\begin{aligned}
\mathit{global} ::=\ & \mathtt{system}\ \{ \\
& \langle\ \mathtt{extern} = \langle \mathit{extern}_1, \ldots \rangle\ \rangle? \\
& \langle\ \mathtt{environment} = \langle \mathit{decl}_1; \ldots \rangle\ \rangle? \\
& \mathtt{spawn} = \langle \mathit{spawn}_1, \ldots \rangle \\
& \langle \mathbb{ID} = \mathit{proc}\ \rangle{*} \\
& \} \\
\mathit{extern} ::=\ & \_\mathbb{V} \\
\mathit{spawn} ::=\ & \mathbb{ID}\ \mathtt{:}\ \mathbb{NUM}
\end{aligned}$$

(C) Stigmergy declaration.

$$\begin{aligned}
\mathit{stigmergy} ::=\ & \mathtt{stigmergy}\ \mathbb{ID}\ \{ \\
& \mathtt{link} = \mathit{bexpr}^{\mathbb{V}\times\{1,2\}} \\
& \langle\ \mathit{tupleDecl}\ \rangle{+} \\
& \} \\
\mathit{tupleDecl} ::=\ & \langle \mathbb{V}_1, \ldots \rangle\ \mathtt{:}\ \langle \mathit{init}_1, \ldots \rangle \\
\mathit{ref}^{\mathbb{V}\times\{1,2\}} ::=\ & \mathbb{V}\ \langle\ \mathtt{[}\ \mathit{expr}^{\mathbb{V}\times\{1,2\}}\ \mathtt{]}\ \rangle?\ \mathtt{of\ 1} \\
\mid\ & \mathbb{V}\ \langle\ \mathtt{[}\ \mathit{expr}^{\mathbb{V}\times\{1,2\}}\ \mathtt{]}\ \rangle?\ \mathtt{of\ 2}
\end{aligned}$$

(Table 3.15a).

Global section. The global section (Table 3.15b) is enclosed in braces and preceded by the `system` keyword. It contains an optional `extern` section where external parameter names are defined. A parameter name is a variable name preceded by an underscore. Once defined, external parameters may be used throughout the specification, in any place where a numeric literal ($\mathbb{NUM}$) is expected. The global section may also contain an `environment` section with declarations of shared variables, and it must have a `spawn` section that specifies the composition of the system. Additionally, it may contain zero or more global *process definitions*, binding identifiers to process terms. Process constants that occur in these process terms must refer to other processes defined within this section.

Stigmergy declarations. A stigmergy declaration (Table 3.15c) is enclosed in braces, and preceded by the keyword `stigmergy` and the name of the stigmergy. To encode link predicates, we need to introduce *link-references*, denoted as $\mathit{ref}^{\mathbb{V}\times\{1,2\}}$. These references are decorated with either `of 1` or `of 2` to denote that they must be evaluated against the sender's and the receiver's knowledge, respectively. As an example, the link-reference `x of 1` corresponds to the syntax x_s as defined in Section 3.2.2. Once we introduce link-references, we can easily define (arithmetic and Boolean) *link-expressions*: they are defined just as in Table 3.14a, but feature link-references instead of plain ones. Thus, a link predicate is just a Boolean link-expression, preceded by the `link` keyword and an equal sign. Furthermore, a link-reference to an array variable may feature an arithmetic link-expression as an index. After the link predicate, there may be one or more declarations of stigmergic variables. Tuples (see Section 3.2.4) are defined by comma-separated sequences of variables, followed by a matching number of initialisers.

Agent declarations. An agent declaration describes the local variables and behaviour of a *kind* of agent (Table 3.16a). As usual, it is enclosed in braces and preceded by the keyword `agent`,

TABLE 3.16: LAbS$^+$ syntax (3).

(A) Agent declarations.

$$\begin{aligned}
\mathit{agent} ::= &\ \texttt{agent}\ \mathbb{ID}\ \texttt{\{}\\
&\langle\ \texttt{interface} = \langle \mathit{decl}_1; \ldots \rangle\ \rangle ?\\
&\langle\ \texttt{stigmergies} = \langle \mathbb{ID}_1; \ldots \rangle\ \rangle ?\\
&\langle\ \mathbb{ID} = \mathit{proc}\ \rangle *\\
&\texttt{\}}
\end{aligned}$$

(B) Property section.

$$\begin{aligned}
\mathit{check} &::= \texttt{check \{}\\
&\ \langle\ \mathbb{ID} = \mathit{property}\ \rangle *\\
&\ \texttt{\}}\\
\mathit{property} &::= \mathit{modality}\ \langle\ \mathit{quantifier}\ \texttt{,}\ \rangle *\ \mathit{bexpr}^{\mathbb{V}\times\mathbb{V}}\\
\mathit{modality} &::= \texttt{finally} \mid \texttt{always}\\
\mathit{quantifier} &::= \texttt{exists}\ \mathbb{ID}\ \ \mathbb{V} \mid \texttt{forall}\ \mathbb{ID}\ \ \mathbb{V}\\
\mathit{ref}^{\mathbb{V}\times\mathbb{V}} &::= \mathbb{V}\ \langle\ \texttt{[}\ \mathit{expr}^{\mathbb{V}\times\mathbb{V}}\ \texttt{]}\ \rangle ?\ \texttt{of}\ \mathbb{V}
\end{aligned}$$

plus the name of the kind. It may contain one or more local variable declarations (preceded by the `interface` keyword), and a sequence of stigmergy identifiers (preceded by `stigmergies`), to indicate which stigmergic variables are maintained in the agent's local stigmergy. Then, the agent declaration contains one or more named process definitions. One of these processes must have the identifier `Behaviour`, and will be identified as the one encoding the behaviour of this kind of agent. Process constants that occur within these process terms may refer to other, locally-defined processes, or to global definitions.

Property section. The property section (Table 3.16b) is introduced by the `check` keyword and enclosed in braces. It contains zero or more property definitions. A property is composed by a temporal *modality* (`finally` or `always`), zero or more *quantifiers* with trailing commas, and a state predicate. A state predicate is a Boolean expression where all references are decorated with names. We denote the set of such references by $\mathit{ref}^{\mathbb{V}\times\mathbb{V}}$. Each name must appear in one of the quantifiers of the property: for instance, `forall Agent a, x of a = 0` is a valid fragment, while `forall Agent a, x of b = 0` is not.

3.6 Summary

In this chapter, we have introduced LAbS, a core language to describe agent behaviours, and LAbS$^+$, a machine-readable specification formalism that extends LAbS. The key feature of the language is a distributed, decentralised data structure to model inter-agent communication, or knowledge propagation. This data structure is based on the concept of virtual stigmergy. After some preliminary definitions (Section 3.1), Section 3.2 introduced the basics of the language and its operational semantics. In Section 3.3, we described how LAbS also allows to model the external environment, i.e. a shared data repository accessible by the different agents. In Section 3.4, we gave an informal description of LAbS$^+$ and showed how it can be used to model a selection of multi-agent systems. Lastly, Section 3.5 provided a formal description of the LAbS$^+$ syntax.

Chapter 4

Sequential emulations

In this chapter, we explain how to reduce property checking of distributed systems to verification of sequential imperative programs. We call this technique *sequential emulation*. Our approach is rather general, in that it can be used to translate from any domain-specific language defined with a structural operational semantics (SOS) to any imperative language with arrays and loops. Any verification technique for the target language that supports standard instrumentation constructs for automated analysis (i.e., nondeterministic initialisation, assertions, assumptions) can then simply be used as a black box. As an immediate advantage, this approach may leverage a comprehensive range of readily available techniques for the analysis of sequential imperative programs. Furthermore, it can easily integrate new techniques as soon as they become available. Our methodology is shown in the diagram below (Figure 4.1).

$$\begin{tikzcd}
\begin{array}{c}\text{Concurrent}\\ \text{Formal}\\ \text{Specifications}\\ \mathbb{S}\end{array} \arrow[r] & \begin{array}{c}\text{Triple structure}\\ \mathbb{T}\end{array} \arrow[r] & \begin{array}{c}\text{Sequential}\\ \text{Imperative}\\ \text{Program}\\ \mathbb{P}\end{array} \arrow[r] & \begin{array}{c}\text{Program}\\ \text{Verification}\end{array} \\
\begin{array}{c}\text{Property}\\ \phi\end{array} \arrow[urr, bend right] & & &
\end{tikzcd}$$

FIGURE 4.1: Overview of our encoding procedure.

Essentially, from a formal specification of a distributed system $\mathbb{S}$ and a temporal property ϕ we generate a program $\mathbb{P}$ in some imperative language. Depending on the property, we perform a certain verification task on $\mathbb{P}$ which will determine whether ϕ holds in $\mathbb{S}$.

Let us first clarify how we interpret a temporal property ϕ over a system $\mathbb{S}$. The SOS of the input language associates to $\mathbb{S}$ a labelled transition system $\mathcal{S} \triangleq \langle S, \Lambda, \rightarrow \rangle$. Let AP be the set of atomic propositions that appear in ϕ, and assume that we can always define a subset $I \subseteq S$ of *initial states*, as well as a *labelling function* L that associates to each state of $\mathcal{S}$ a subset of *AP*. [1]

[1]These assumptions are intentionally abstract. The atomic propositions that may appear in ϕ, their interpretation over the states of $\mathcal{S}$, and the set of initial states depend on the semantics of the input specification language.

Then, the 4-ple $K \triangleq \langle S, I, L, \rightarrow \rangle$ is a Kripke structure that models $\mathbb{S}$: therefore, the question of whether ϕ holds in $\mathbb{S}$ is equivalent to asking whether $K \models \phi$.

Our procedure currently supports basic *invariants* as well as simple *emergent* properties, corresponding respectively to LTL formulas of the form $\Box\psi$ and $\Diamond\psi$, where ψ is a predicate over atomic propositions. Its key ingredient is a two-phase encoding from distributed systems to imperative programs. In the first step of the encoding (Sect. 4.1), we generate from $\mathbb{S}$ a *triple structure* $\mathbb{T}$. The triple structure acts as an intermediate representation and compactly represents the behaviour of the input system. Such representation consists of a set of basic elements, or *transition triples*, corresponding to the individual actions of the agents as they occur in their behavioural specifications. We then introduce a mechanism to *enable* or *disable* these triples, so that $\mathbb{T}$ correctly reproduces the behaviour of $\mathbb{S}$. For instance, if we have to encode a sequential composition $a.b$ of two actions, we want $\mathbb{T}$ to capture that the second action b can only be enabled after a has been consumed. To do so, we equip $\mathbb{T}$ with a *program counter* to keep track of the current execution point, and assigning a unique *identifier* to a and b, (e.g., 1 and 2, respectively). Then, we initialise the program counter of $\mathbb{T}$ to 1 to enable action a; we set it to 2 right after executing a to enable b, and to 0 right after consuming b, so that a or b cannot be executed again. We can generalise this reasoning to other composition operators, e.g., choice and parallel composition. In general, we can express the possible execution flows allowed by $\mathbb{S}$ in terms of *symbolic expressions* over the values of the program counter right before and right after consuming an action. Such *entry and exit conditions* will capture all the feasible flows of actions in $\mathbb{S}$ by appropriately enabling or disabling the corresponding triples of $\mathbb{T}$. Unlike an explicit LTS encoding, this representation does not need a full enumeration of the states and transitions of $\mathbb{S}$, and thus retains the compactness of the original specification.

In the second step (Sect. 4.2), we obtain an *emulation program* $\mathbb{P}$ from $\mathbb{T}$. The global variables of $\mathbb{P}$ represent the state of $\mathbb{S}$. For each triple in $\mathbb{T}$, we generate an *emulation function* in $\mathbb{P}$ that mimics the corresponding action of $\mathbb{S}$. To do so, the function manipulates the global variables of $\mathbb{P}$ as prescribed by the SOS rules for that action. The emulation function also enforces the entry condition of the triple by means of an assumption over program counter variables; similarly, it encodes the exit condition through a (possibly nondeterministic) assignment to these variables.

This approach is *semantics-based*: to encode a language, one has to devise the appropriate global variables and emulation functions so as to encode the SOS of the source language. If the semantics of the input language specifies additional transitions that $\mathbb{S}$ may perform, we introduce additional emulation functions to represent them.

We then complete $\mathbb{P}$ with its main function, that models the evolution of $\mathbb{S}$ by repeatedly calling the emulation functions. Finally, we instrument $\mathbb{P}$ for property checking. In particular, depending on ϕ, we reduce the problem of checking whether $\mathbb{S} \models \phi$ to either reachability in, or termination of, $\mathbb{P}$.

Our procedure only requires a one-time manual effort that has not been automated yet. Namely, one has to render the semantics of the actions of the source language as code fragments that have to be expanded within the emulation functions stubs in the target program. Once these fragments have been provided, the procedure does not require any user interaction. To have a fully-automated semantic-based synthesis of program verifiers, we plan to support machine-readable rule formats, such as MSOS [184].

4.1 From Formal Specifications to Symbolically Linked Triples

We first transform $\mathbb{S}$ into a *triple structure* $\mathbb{T}$ by encoding the actions of the former into triples of the latter. Intuitively, each triple of $\mathbb{T}$ symbolically represents transitions in the LTS of the encoded process. The execution of $\mathbb{S}$ is modelled by keeping track of which transitions may be performed at any given time. This is achieved by equipping the triple structure with a *program counter* and by guarding each triple with an appropriate predicate over it.

Definition 4.1. A *program counter* $pc = \langle pc_0, pc_1, \ldots, pc_l \rangle$ is a vector of integers.

Definition 4.2. A *transition triple* (or *triple*, for short) consists of an *entry condition*, an *action*, and an *exit condition* (see Definitions 4.3, 4.4, and 4.5), and is denoted as

$$t = \langle \rhd(t), \mu(t), \lhd(t) \rangle.$$

Definition 4.3. The *entry condition* $\rhd(t)$ is a predicate over the elements of a program counter. More specifically, it is always in the form of a conjunction of clauses $\rhd_0 \wedge \rhd_1 \wedge \cdots \wedge \rhd_l$, where the i-th clause predicates over pc_i. Given a program counter pc, we write $pc \models \rhd$ to denote that the elements of pc are a valid assignment for $\rhd$.

Definition 4.4. The *action* $\mu(t)$ is either an action in the language of the specification $\mathbb{S}$, or a distinguished *null action* λ. We will use the null action for triples that do not directly encode the specification, but are needed for our encoding to behave correctly.

Definition 4.5. The *exit condition* $\lhd(t)$ is also a predicate over the elements of pc. We write $\lhd_i = (\cdot)$ to denote that pc_i is unconstrained in an exit condition $\lhd$.

Definition 4.6. A *symbolically linked triple structure* (SLTS; *triple structure*, for short) is a pair $\mathbb{T} = \langle T, pc \rangle$, where T is a set of transition triples and pc is a program counter. With a slight abuse of notation, we may write $t \in \mathbb{T}$ whenever a triple t is a member of the set of triples of $\mathbb{T}$.

Definition 4.7. A triple $t \in T$ is *enabled* in a triple structure $\langle T, pc \rangle$ iff. $pc \models \rhd(t)$. In general, any number of triples may be enabled for any given value of pc.

Definition 4.8. The *evolution condition* of a triple structure is the least relation induced by the following inference rule:

$$\frac{t \in T \quad pc \models \rhd(t) \quad pc' \models \lhd(t) \quad \lhd_i(t) = (\cdot) \Rightarrow pc'_i = pc_i}{\langle T, pc\rangle \xrightarrow{\mu(t)} \langle T, pc'\rangle}$$

We call each element $\langle \mathbb{T}, \mu(t), \mathbb{T}'\rangle$ in the evolution condition an *evolution* of $\mathbb{T}$ into $\mathbb{T}'$ via t. Whenever such an evolution exists, we say that $\mathbb{T}$ *may evolve* to $\mathbb{T}'$ via t.

The evolution condition has four premises. The first two state that T contains an enabled triple t; the last two define the new value of the program counter, and therefore the set of triples that will be enabled, after the triple structure has evolved. Intuitively, the third premise says that the new program counter pc' must satisfy exit condition $\lhd(t)$. The fourth additionally states that every component pc'_i that was unconstrained by $\lhd(t)$ should have the same value as before the transition. Therefore, the execution of two triples with the same exit condition will not necessarily yield the same sets of enabled triples.

Definition 4.9 (Symbolic links)**.** Let t an enabled triple in a triple structure $\mathbb{T}$, and assume that $\mathbb{T}$ may evolve to $\mathbb{T}'$ via t. For each enabled triple $t' \in \mathbb{T}'$, we say that t is *symbolically linked* to t'.

We say that a triple t is *symbolically linked* to another triple t' iff. t is enabled in $\mathbb{T}$ and there exists an evolution of $\mathbb{T}$ into a $\mathbb{T}'$ such that t' is enabled in $\mathbb{T}'$. Notice that multiple triples may be enabled at the same time, and that the update operation may return multiple feasible values for the program counter. Thus, each triple may be linked to several other ones.

Intuitively, we can construct a triple structure that mimics the LTS of any given process by giving appropriate entry and exit conditions to each triple. We will formalise an encoding function $[\![\cdot]\!]$ that maps process terms to triple structures. We assume that each elementary sub-expression μ of the encoded process term is given a unique *identifier*, i.e., a positive number $id(\mu)$. Notice that, even though two sub-expressions may be identical, their identifiers are still distinct. For instance, if the encoded process contain several occurrences of the same action, we will reserve a separate identifier for each occurrence. The encoding function considers three process composition operators: binary sequential composition ($P; Q$); n-ary nondeterministic choice ($\Sigma_i P_i$); and n-ary parallel composition ($\Pi_i P_i$). We treat action prefixing (denoted $a.P$ in CCS [180]) as a special case of sequential composition. Processes within a parallel composition may perform two-party synchronisation according to a (language-specific) *synchronisation algebra* [246]. The encoded process may also contain *process constants*: we assume that these constants always refer to the process itself. We also restrict the use of recursion, by disallowing recursion within parallel composition (as in $P \triangleq a.0 \mid b.P$), as well as unguarded recursion (as in $P \triangleq P + a$).

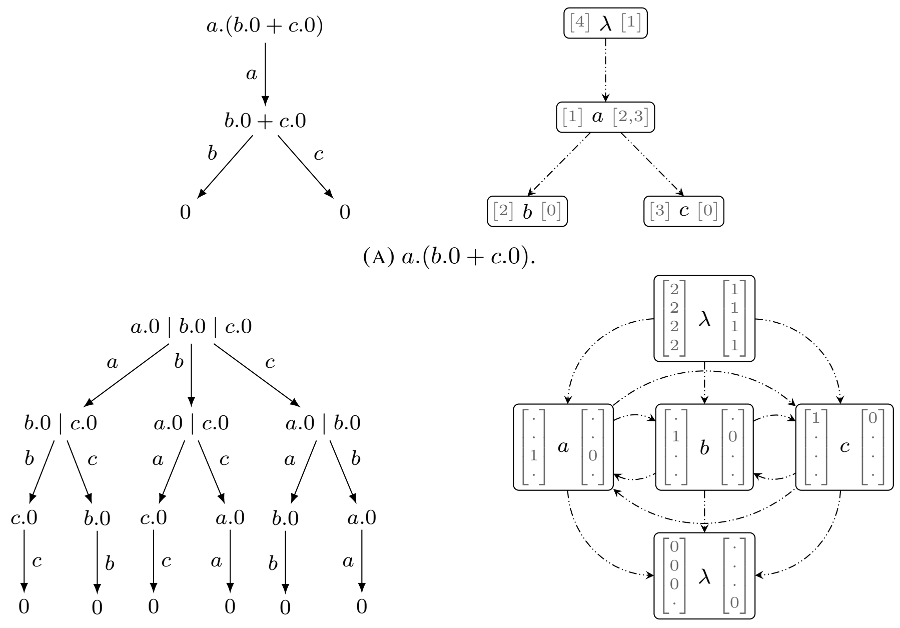
Elementary examples. Before we formalise the encoding procedure, let us consider a couple of elementary processes expressed in CCS, their LTSs as defined by the semantics of the language, and the corresponding triple structures . (Figures 4.2a and 4.2b). In both examples, triples are represented by boxes containing their entry condition, action, and exit condition, with edges between two triples to denote that they are symbolically linked. To make our triple structure diagrams more readable, we use the following graphical conventions:

- Each entry condition is represented as a vector of length n. In each component i of the vector ($i = 0, 1, \dots$), we write k if the entry condition contains a clause $pc_i = k$, or $\cdot$ if the component is unconstrained.
- We represent exit conditions likewise, but allow multiple feasible values for each component (denoting a disjunction). For instance, after executing the a-triple of Fig. 4.2a, pc_0 is nondeterministically assigned to either 2 or 3. Note that multiple triples may be enabled at the same time, and that each triple may be symbolically linked to many triples; conversely, many triples may be linked to the same triple.

Notice that both examples feature a triple with an entry condition $pc = pc^\star$ and a null action (λ). We call them *start triples* ($t^\star$). A start triple does not correspond to any concrete action in $\mathbb{S}$, but is always guaranteed to be executed at the very beginning. We ensure this by initialising the program counter to a value $pc^\star$ that only enables the start triple (which is given a unique entry condition). In turn, the exit condition $\lhd(t^\star)$ sets up the program counter so as to only enable those triples corresponding to *initial* actions of the encoded process.

In Fig. 4.2a we consider the LTS (left) and triple structure (right) of a process $a.(b.0 + c.0)$. In this example, the program counter contains a single component pc_0. Since the process must necessarily perform a as its first action, the start triple $t^\star$ is symbolically linked with the a-triple only. In turn, the exit condition of the a-triple may enable either the b-triple or the c-triple, mimicking the semantics of the choice operator $+$. The other two triples instead update pc_0 to 0 to denote that the process terminates after performing either of them.

Fig. 4.2b shows how a parallel process $\Pi \triangleq a.0 \mid b.0 \mid c.0$ is encoded by a triple structure with four triples and a program counter of length 4. The first component of the program counter, pc_0, tracks the execution of the overall process Π (the *parent*), while each of the other component $pc_{1,2,3}$ tracks one of the sub-processes within the parallel composition (the *children* of Π). Process Π may perform any permutation of actions a, b, and c. Therefore, the exit condition of $t^\star$ enables all their corresponding triples. We also track the termination of Π by adding a *join triple* t_{join} that has entry condition $pc_1 = pc_2 = pc_3 = 0$ and no action. This entry condition is only satisfied when all children of Π have terminated. If this is the case, the triple sets the element pc_0 to 0 so as to signal that Π itself has terminated as well. This addition may not seem necessary at this



(A) $a.(b.0 + c.0)$.

(B) $a.0 \mid b.0 \mid c.0$.

FIGURE 4.2: LTSs and triple structures for two simple CCS processes.

point, but is needed when the parallel process is sequentially composed with another process ($\Pi; Q$). In this case, the join triple updates the program counter so that the Q process may start upon termination of Π.

Notice that each μ-triple ($\mu = a, b, c$) symbolically represents all μ-transitions of the LTS of Π. The vectorial program counter keeps track of all interleavings within the parallel composition without explicitly representing them. This retains compactness.

Encoding sequentiality and choices. Let us start by considering processes with no parallel composition. To encode these processes, a program counter with a single component pc_0 is sufficient. An action μ is encoded as a triple that has entry condition $pc_0 = id(\mu)$ and action μ. The exit condition of this μ-triple depends on the actions that may follow μ in the encoded process. If μ is followed by a generic process term P, we can define a function $\lhd [P]_0$ returning an exit condition over pc_0. This exit condition only enables the correct triples generated from P, i.e., those corresponding to actions that may directly follow μ. Thus, we call $\lhd [P]_0$ the *enabler* of P. To model recursion, we simply compute the enabler of the entire process and use it as the exit condition of the triples preceding the recursive call. For instance, consider a process $P \triangleq a.P$. The exit condition of the a-triple is $\lhd [P]_0 = \lhd [a.P]_0$. In general, we define the enabler of a sequential composition $P; Q$ to be the enabler of P: intuitively, the triples generated from Q cannot be enabled until P has terminated. Since we treat action prefixing as a specific case of sequential composition, we apply this definition to obtain $\lhd [a.P]_0 = \lhd [a]_0$.

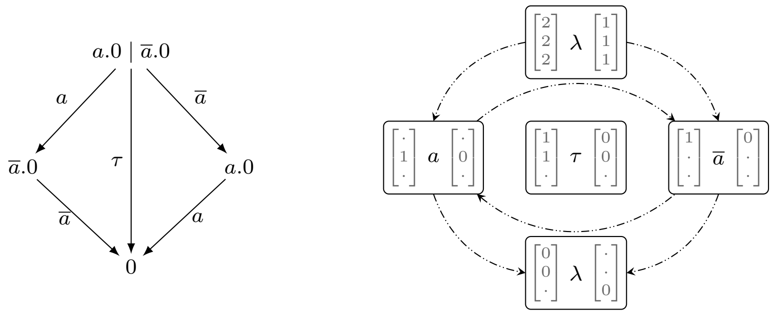


FIGURE 4.3: LTS and triple structure for a process with synchronisation.

Therefore, the a-triple enables itself after being performed, and the recursive nature of P is encoded properly.

To illustrate the procedure in detail, let us see how the process $a.(b.0 + c.0)$ is encoded as the triple structure shown in Fig. 4.2a. Let us assume that $id(a) = 1$, $id(b) = 2$, $id(c) = 3$, and let us start by encoding the $b.0$ sub-term. To do so, we encode b and 0 separately and then use the enabler of 0 as the exit condition of b. The encoding $[\![b]\!]$ is a single triple with entry condition $\rhd[b]_0 \triangleq pc_0 = 2$; on the other hand, 0 is the deadlocked process and thus its encoding is the empty set. We define its enabler to be $\lhd\,[0]_0 \triangleq pc_0 := 0$, which we use as the exit condition of the b-triple. We encode the sub-term $c.0$ likewise. Finally, we encode the sequential composition of a with the choice $b.0 + c.0$. First, we construct an a-triple with entry condition $pc_0 = 1$. Then, we compute the enabler of the choice term as the disjunction of the enablers of its sub-terms $(pc_0 := 2 \vee pc_0 := 3)$. Informally, this exit condition will set pc_0 to either 2 or 3.

Encoding of a parallel process. Let us now consider processes that contain a parallel composition operator, such as the one shown in Fig. 4.2b. To encode such a process, we give additional, unique identifiers $id(\Pi_i P_i)$, $id(P_i)$ to all parallel composition terms and their sub-terms. Each child process within a parallel composition $\Pi \triangleq \Pi_i\ P_i$ may evolve independently of the others. To model that, we encode each child process as a separate set of triples, whose entry and exit conditions consider separate elements $pc_{id(P_i)}$ of the program counter. The enabler of Π is the sequence of the enablers of its children: this encodes the fact that all children are enabled from the start. Furthermore, we add to the sequence another assignment $pc_k := id(\Pi)$. Intuitively, by setting pc_k to such a value, we guarantee that only triples from within Π may be performed until Π terminates.

This encoding captures the *interleaving* of the children processes, which progress by alternating the execution of their actions. However, many process algebras also allow parallel processes to *synchronise* on pairs of actions. A general way to represent this behaviour is by means of a *synchronisation algebra* σ (see Section 2.3.3). For instance, in CCS we have $\sigma(a, b) = \tau$ iff. $b = \overline{a}$, and $\sigma(a, b) = \bot$ otherwise. Furthermore, $\sigma(a, *) = a$ for any action a, meaning that a process may always interleave its actions with others. Our approach to model two-party synchronisation, then, is to first compute the interleaving triple structure, and then insert a transition for each pair of actions that may synchronise. We illustrate this approach on the simple CCS process $a.0 \mid \overline{a}.0$ (Fig. 4.3). Its triple structure is obtained by first computing the interleaving triple structure, resulting in one a-triple t_a and one $\overline{a}$-triple $t_{\overline{a}}$. Then, we introduce a third triple with entry condition $\triangleright(t_a) \wedge \triangleright(t_{\overline{a}})$, action $\sigma(a, \overline{a}) = \tau$, and exit condition $\triangleleft(t_a), \triangleleft(t_{\overline{a}})$. By definition, whenever t_a and $t_{\overline{a}}$ are enabled, so is this newly created τ-triple. If the τ-triple is performed, it applies $\triangleleft(t_a)$ to the current program counter and then applies $\triangleleft(t_{\overline{a}})$ to the result, therefore setting all components of pc to 0.

Formal definitions. We formally define our *translation* from a behaviour S to a set of triples via a function $[\![P]\!]_k^{\triangleleft}$ (Table 4.1), where P is a syntactic fragment of S, k is a program counter index, and $\triangleleft$ is an exit condition. Both the deadlocked process 0 and the recursive invocation K translate to the empty set. The translation $[\![\mu]\!]_k^{\triangleleft}$ of a single action μ only contains one triple. This triple has an entry condition checking that pc_k matches the identifier of μ, action μ, and exit condition $\triangleleft$. To translate a sequential composition $P; Q$, we translate the two processes separately and use the enabler of Q as the exit condition parameter when translating P. The translation of a choice is the union of the translations of its terms. Finally, translating the parallel composition of n process terms P_i requires translating each P_i with a different program counter index, namely $id(P_i)$. Then, a synchronisation triple is added for each pair of triples corresponding to synchronising actions. Lastly, the join triple must be added, so that, when all child processes have terminated, the exit condition $\triangleleft$ of the parent may be applied. Finally, Table 4.2 contains a formal definition of the enabler function which has been informally described thus far. Both definitions are given by induction on the structure of syntactic fragments P.

Definition 4.10 (Encoding of a process)**.** Given a process S, let $T = [\![S]\!]_0^{pc_0=0}$ and let $pc^\star$ be a program counter that does not enable any triple in T. Then, the *encoding* of S is the triple structure $\mathbb{T} = \langle T \cup \{t^\star\}, pc^\star \rangle$, where $t^\star \triangleq \langle pc = pc^\star, \bot, \triangleleft\, [S]_0 \rangle$.

System-level encoding. We consider a *system* $\mathbb{S}$ to be a composition of concurrent processes (i.e., the agents), which we assume to be different from each other and possibly recursive. We also assume that agents may synchronise on the pairs of actions specified in a synchronisation algebra $\sigma_{\mathbb{S}}$.

Definition 4.11 (Encoding of a system)**.** Let $\mathbb{S} = \{S_1, \ldots, S_n\}$. Let $\mathbb{T}_i = \langle T_i, pc^{i\star} \rangle$ the encoding of S_i. Then the *encoding* of $\mathbb{S}$ is a triple structure $\langle T \cup T_\sigma, pc^\star \rangle$ where:

- T is the union of all T_i, with appropriate adjustments to the entry and exit conditions of triples to avoid spurious symbolic links;

TABLE 4.1: Definition of the translation function $[\![\cdot]\!]$.

$$
\begin{aligned}
[\![0]\!]_k^{\lhd} &\triangleq \emptyset\\
[\![K]\!]_k^{\lhd} &\triangleq \emptyset\\
[\![\mu]\!]_k^{\lhd} &\triangleq \{\langle pc_k = id(\mu),\, \mu,\, \lhd\rangle\}\\
[\![P;Q]\!]_k^{\lhd} &\triangleq [\![P]\!]_k^{\lhd[Q]_k} \cup [\![Q]\!]_k^{\lhd}\\
[\![\Sigma_i P_i]\!]_k^{\lhd} &\triangleq \textstyle\bigcup_i [\![P_i]\!]_k^{\lhd}\\
[\![\Pi_i P_i]\!]_k^{\lhd} &\triangleq \textstyle\bigcup_i T_i \cup \bigcup_{i\neq j} \{[\![t_i, t_j]\!]_\sigma \,|\, t_i \in T_i, t_j \in T_j, \sigma(\mu(t_i), \mu(t_j)) \neq \bot\} \cup t_{join}\\
\text{where} \quad T_i &= [\![P_i]\!]_{id(P_i)}^{(pc_{id(P_i)}=0)}\\
[\![t, t']\!]_\sigma &= \big\langle \rhd(t) \wedge \rhd(t'), \sigma(\mu(t), \mu(t')), \big(\lhd(t), \lhd(t')\big)\big\rangle\\
t_{join} &= \Big\langle \textstyle\bigwedge_i pc_{id(P_i)} = 0, \bot, \lhd\Big\rangle
\end{aligned}
$$

TABLE 4.2: Definition of the enabler function $\lhd [\cdot]_k$.

$$
\begin{aligned}
\lhd [0]_k &\triangleq (pc_k = 0)\\
\lhd [K]_k &\triangleq \lhd [P]_k \text{ iff. } K \triangleq P\\
\lhd [\mu]_k &\triangleq (pc_k = id(\mu))\\
\lhd [P;Q]_k &\triangleq \lhd [P]_k\\
\lhd [\Sigma_i P_i]_k &\triangleq \textstyle\bigvee_i \lhd [P_i]_k\\
\lhd [\Pi_i P_i]_k &\triangleq (pc_k = id(\Pi_i P_i)) \wedge \textstyle\bigwedge_i \lhd [P_i]_{id(P_i)}
\end{aligned}
$$

- T_σ contains a triple for each pair of triples $t_i \in T_i, t_j \in T_j$ such that $i \neq j$ and the actions of the two triples may synchronise according to $\sigma_{\mathbb{S}}$;
- $pc^\star$ is the concatenation of all $pc^{i\star}$.

Intuitively, when we construct T we need to adjust entry and exit conditions of each T_i so that they refer to the correct components of the concatenated program counter.

To illustrate the system encoding procedure, we will describe and encode a simplified *two-phase commit* (2PC) scenario [112]. A two-phase commit protocol involves a number of *workers*, which must collectively decide whether to commit or rollback a transaction by interacting through a *coordinator*. In the first phase, the coordinator asks the workers to cast a vote. Each worker may either agree or disagree on committing the transaction. In the second phase, if all workers have agreed, the coordinator tells them to commit the transaction; otherwise, the coordinator sends them a rollback request. In any case, the workers send an acknowledgement message back to the coordinator and finalise the transaction. We consider a system composed of one coordinator and one worker, which act according to the following CCS processes:

$$\text{COORD} \triangleq vote.\left(\overline{agree}.commit.\overline{commit}.ok.0 + \overline{disagree}.rollback.\overline{rollback}.nok.0\right) \quad (4.1)$$

$$\text{WORKER} \triangleq \overline{vote}.agree.\left(\overline{commit}.commit.\text{WORKER} + \overline{rollback}.rollback.\text{WORKER}\right) \quad (4.2)$$

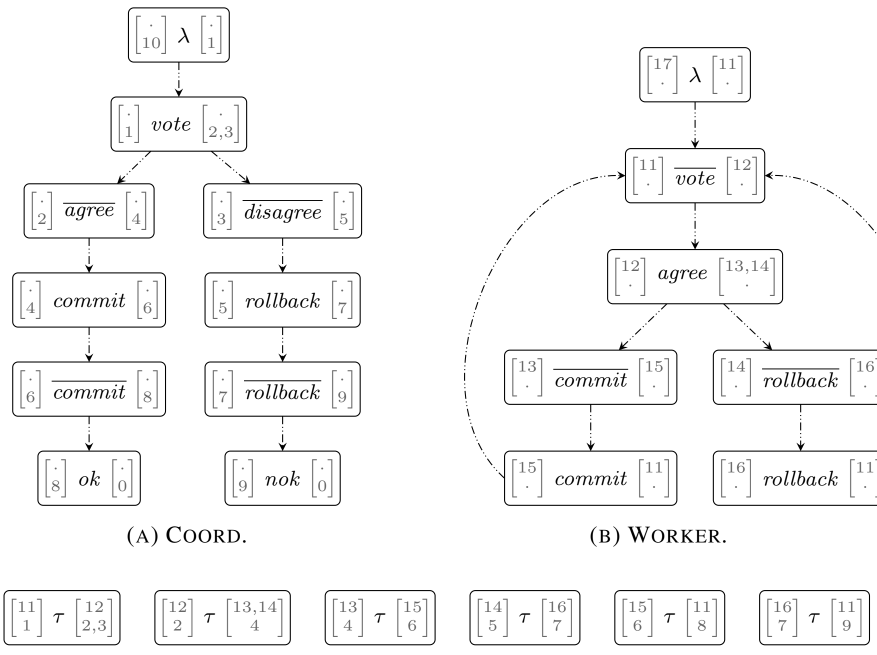
The set of triples that encode this system is shown in Fig. 4.4. It includes the individual triple structure of each agent, namely COORD (Fig. 4.4a) and WORKER (Fig. 4.4b), and an additional set of τ-triples, one for each pair of complementary actions (Fig. 4.4c). The program counter of the whole system has two components, respectively tracking the evolution of WORKER and COORD.

Since WORKER always agrees to commit, we would expect the *nok* action (denoting that a transaction has been rolled back) to be unreachable. However, CCS processes are not *forced* to synchronise: they might simply perform their actions independently from each other. As a consequence, this system (and its triple structure) admits several traces leading to *nok*: one of them, namely $\langle \tau, \overline{disagree}, agree, \tau, \overline{rollback}, nok \rangle$, is graphically represented in Fig. 4.5. Each diagram in the figure represents the triple structure of Fig. 4.4: black circles denote enabled triples.

Language-specific interaction rules. In some process algebras there is a strict difference between terms that may freely occur within the structure of a process, and terms that may only appear at the top level. These operators typically describe additional rules related to the interaction between agents. For instance, in our encoding of CCS we assume that the *restriction* operator $P \backslash \Theta$ (where Θ is a set of actions) may only be used on the top-level process. This does not lead to any loss of generality for our encoding: a CCS process with any number of restriction sub-terms may always be expressed as a process with a single restriction, by means of a suitable α-renaming to avoid unwanted name capture. Encoding a restricted system $\mathbb{S} \setminus \Theta$ is straightforward: we simply encode $\mathbb{S}$, then remove any triple whose action is (equal or complementary to) a member of Θ.

To show the effect of restriction, let us consider a variant of the 2PC example where we restrict the system by $\Theta \triangleq \{agree, disagree, commit, rollback\}$. This restriction forces the coordinator to synchronise with the worker on all actions within Θ: since WORKER is never willing to *disagree*, the action *nok* becomes unreachable. The triple structure of the restricted 2PC system is obtained from the one of Fig. 4.4 by removing all triples except the ones encoding *ok*, *nok*, and the synchronisations.

Supporting multi-party synchronisation. Several process algebras, such as CSP and LOTOS, feature *multi-party synchronisation* between parallel processes. As an example, let us consider



(A) COORD. (B) WORKER. (C) Synchronisation triples.

FIGURE 4.4: Triple structure generated from the specifications of the two-phase commit (2PC) example.

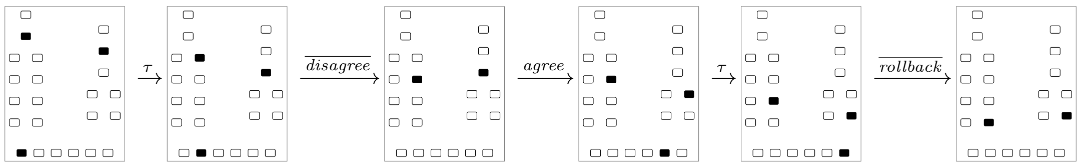


FIGURE 4.5: Graphical representation of a possible execution of the 2PC example.

Hoare's parallel composition operator $P \mid[L]\mid Q$, where L is a set of actions. Intuitively, P and Q may interleave the execution of all actions that are not in L, and they are forced to synchronise on those in L. Whenever they do synchronise on an action $a \in L$, the parallel composition performs an a-move. This allows an external process to synchronise with the a action again. For instance, a process $a \mid[\{a\}]\mid a \mid[\{a\}]\mid a$ may perform a single a-move and become $\surd$. This behaviour is formalised by the following semantic rules:

$$\frac{P \xrightarrow{\mu} P' \quad \mu \notin L}{P \mid[L]\mid Q \xrightarrow{\mu} P' \mid[L]\mid Q} \qquad \frac{Q \xrightarrow{\mu} Q' \quad \mu \notin L}{P \mid[L]\mid Q \xrightarrow{\mu} P \mid[L]\mid Q'} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad a \in L}{P \mid[L]\mid Q \xrightarrow{a} P' \mid[L]\mid Q'}$$

A synchronisation algebra σ such that $\sigma(a, a) = a$ and $\sigma(a, *) = \bot$ for all $a \in L$ can fully capture Hoare's operator $\cdot \mid[L]\mid \cdot$. However, the encoding function of Table 4.1 only accounts for operators such as Milner's, where $\sigma(a, *) = a$ for all actions. Algorithm 1 sketches a possible approach to generate the set of triples for a process P defined as n processes composed together

```
input  : A parallel composition $P_1 \,|[L]|\, P_2 \,|[L]|\, \cdots \,|[L]|\, P_n$, and a synchronisation algebra $\sigma$
         that captures $\cdot \,|[L]|\, \cdot$
output : A set of transition triples $T$
1  $T_i \leftarrow [\![P_i]\!]^{(pc_{id(P_i)}=0)}_{id(P_i)}, i = 1, \ldots, n$
2  $T \leftarrow \{t \in T_1 \mid \exists \alpha \neq *.\sigma(\mu(t), \alpha) \neq \bot\}$
3  foreach $i = 2, \ldots, n$ do
4      $I \leftarrow \emptyset, R \leftarrow \emptyset$
5      foreach $(t, t') \in T \times T_i.\sigma(\mu(t), \mu(t')) \neq \bot$ do
6          $I \leftarrow I \cup [\![t, t']\!]_\sigma$
7          $R \leftarrow R \cup t$
8      end
9      foreach $t \in T$ do
10         if $\forall \alpha.\sigma(\mu(t), \alpha) \neq \bot.\forall t' \in T_i.\mu(t') \neq \alpha$ then
11             $R \leftarrow R \cup t$
12         end
13     end
14     $T \leftarrow (T \setminus R) \cup I$
15 end
16 foreach $t \in \{T_1 \cup \cdots \cup T_n\}.\sigma(\mu(t), *) \neq \bot$ do
17     $T \leftarrow T \cup t$
18 end
19 $T \leftarrow T \cup t_{join}$
```

Algorithm 1: Encoding Hoare's parallel operator as a set of transition triples.

with Hoare's operator: $P \triangleq P_1 \,|[L]|\, \ldots \,|[L]|\, P_n$. In the rest of this paragraph we will also use the shorter notation $P \triangleq \big|[L]\big|_{i=1}^{n} P_i$.

First, a set of triples T_i is generated for each process P_i (line 1). Then, we accumulate a set T of triples that initially contains all triples in T_1 whose may synchronise with some other action α (line 2). Then, for each T_i, $i = 2, \ldots, n$, we accumulate two set of triples which will be either inserted into T (I) or removed from it (R). Both are initially empty (line–4). For each pair of triples (t, t') in $(T \times T_i)$ such that their actions may synchronise, we add a synchronisation triple to I (line 6) and add the triple t to R (line 7). We also add to R all triples $t \in T$ such that $\mu(t)$ cannot synchronise with any triple in T_i (lines 9–12). Intuitively, we need to remove such triples because Hoare's parallel operator requires *all* processes to synchronise on the same action. If T_i contains no triple with such an action, it means that P_i is not willing to perform that action. Therefore, a triple $t \in T$ that cannot synchronise with any triple in T_i encodes a transition that cannot happen, and thus must be removed. At the end of each iteration of the outer loop, we update T by removing all triples that also belong to R, and then adding the triples from I (line–14) . Once all T_i have been processed, we also add to T those triples of $T_1, \ldots, T_n$ whose actions are not forced to synchronise (lines 16–18), and finally add the join triple (line 19).

Having described how Algorithm 1 works, we now state two lemmas concerning its time complexity and its correctness when computing the SLTS for a process $P \triangleq \big|[L]\big|_{i=1}^{n} P_i$. Intuitively, Lemma 4.12 states that Algorithm 1 runs in polynomial time, while Lemma 4.13 shows that each triple within the output of the algorithm represents either an action that does not belong to L, or one that is in L and upon which all processes are willing to synchronise. Thus, T correctly emulates the behaviour of Hoare's parallel operator.

Lemma 4.12 (Time complexity of Algorithm 1)**.** *Let $P \triangleq \big|[L]\big|_{i=1}^{n} P_i$, and let σ be a synchronisation algebra that captures $\cdot \,|[L]|\, \cdot$. Assume that each P_i contains at most m actions, and that no two actions within each P_i may synchronise. Then, Algorithm 1 with input $\langle P, \sigma \rangle$ terminates in time $O(n \cdot m^2)$.*

Proof. The lack of synchronisation within each P_i implies that each triple structure T_i contains no synchronisation triples. Thus, each T_i contains at most $O(m)$ triples and may be generated in time $O(m)$. Therefore, both line 1 and the loop at lines 16–18 run in time $O(n \cdot m)$. Let us now focus on the loop at lines 3–15. The inner loop at lines 5–8 features a Cartesian product between two triple structures T, T_i. As said above, T_i contains $O(m)$ triples. We claim that T also contains $O(m)$, for all iterations of the outer loop. In fact, T is initialised to a subset of T_1, so it contains $O(m)$ triples before the first iteration of the outer loop (line 2). At each iteration, T is updated by adding and removing triples that belong to sets I and R, respectively (line 14). Notice, however, that I and R are empty at the beginning of each iteration of the outer loop (line 4), and that every time time a triple is added to I one is also added to R (lines 6–7). Therefore, T always contains $O(m)$ elements. The inner loop, then, must check $O(m) \cdot O(m)$ pairs of triples, and thus runs in time $O(m^2)$. Therefore, the outer loop runs in time $O(n \cdot m^2)$ and dominates the overall running time of the algorithm. □

Lemma 4.13 (Correctness of Algorithm 1)**.** *Let $P \triangleq \big|[L]\big|_{i=1}^{n} P_i$, σ a synchronisation algebra that captures $\cdot \,|[L]|\, \cdot$, and T the triple structure obtained by running Algorithm 1 on input $\langle P, \sigma \rangle$. Additionally, let $T_1, \ldots, T_n$ the triple structures generated after executing line 1 of Algorithm 1. Then, a triple t belongs to T if and only if one of the following holds:*

1. *t belongs to some T_i and $\sigma(\mu(t), *) \neq \bot$;*
2. *t is the join triple;*
3. *There exist $t_1 \in T_1$, $t_2 \in T_2$, …, $t_n \in T_n$ such that $t = [\![\ldots[\![[\![t_1, t_2]\!]_\sigma, \ldots]\!]_\sigma, t_n]\!]_\sigma$.*

Proof. All triples that satisfy (1) or (2) are added to T at lines 16–19. To show that (3) is a sufficient condition for t being in T, let us consider how triples are added to T throughout the algorithm. Initially, T contains those triples of T_1 whose actions may synchronise according to L (line 2). Whenever we find within some T_i a triple t' that may synchronise with some triple $t \in T$, we replace t with the synchronisation triple $[\![t, t']\!]_\sigma$ (lines 5–8). Furthermore, if for some

$t \in T$, some T_i contains no such triple, we remove t from T (lines 9–12). Therefore, when the loop at lines 3–15 ends, only those triples that satisfy (2) are in T.

Since no other triples are added to T, a triple that satisfies neither (1), nor (2), nor (3) cannot be in T when the algorithm terminates. $\square$

4.2 From Symbolically Linked Triples to Imperative Programs

We now describe how to generate the *emulation program* $\mathbb{P}$ for the system $\mathbb{S}$, relying on $\mathbb{T}$ as an intermediate representation for the individual behaviours of the agents. The structure of $\mathbb{P}$ is shown in Fig. 4.6, where `N` denotes the total number of agents in $\mathbb{S}$, `pc[N]` the program counter, and `agent`, `action` respectively the identifier of the agent and of the action of $\mathbb{S}$ currently being emulated. Separate *emulation functions* in $\mathbb{P}$ encode the transition triples obtained by the procedure from Section 4.1. The program in the figure represents a simplified version of the encoding for 2PC system described in the previous section and whose STLS is shown in Fig. 4.4. For instance, function `vote` emulates the triple $vote$ in Fig. 4.4a. The `assume` statement at the beginning implements the entry condition of the triple (see e.g. line 4). The nondeterministic assignment and the `assume` statement at the end implement the exit condition. For conciseness we do not report the other emulation functions. When dealing with more sophisticated SOS rules, it may be necessary to define additional global variables to represent other information about the state of $\mathbb{S}$. The emulation functions can manipulate such global variables following the operational semantics of the specification language. Notice that, if multiple agents have the same behaviour, we only need to encode that behaviour into emulations functions once. Then, we may use a global variable `id` within each function to determine which agent is performing the corresponding action (not shown in the program). Thus, we can concisely represent systems that contain multiple agents with the same behaviour. In principle, such an encoding could also support the creation of additional agents during the evolution of the system by using dynamic arrays. However, for simplicity we assume that the number of agents is fixed.

In the rest of the encoding of Figure 4.6, we embed the scheduler (lines 28–41) along with a few ancillary functions. The scheduler initialises the global state of the program by invoking function `init()` (line 29), that also initialises the program counters of the agents according to the exit conditions of their start triples. This is equivalent to implementing each start triple as an emulation function and initialising the program counters according to $pc^\star$.

At each iteration of the main loop, the scheduler nondeterministically picks an agent to simulate, say `agent` $= i$, by invoking `next()` (lines 22–26). Then, it simulates an action of S_i by invoking the corresponding emulation function. Any of the emulation functions can be selected

```
int pc[N], agent, action;

vote() {
     assume(pc[agent] = 1);
     action := vote;
     pc[agent] := *;
     assume(pc[agent] = 2 ∨ pc[agent] = 3);
}

...       // Other emulation functions

init() {
     pc[1] := 11;  // Worker
     pc[0] := 1;  // Coord
}

check() {
     if(¬always) error;
     if(eventually) exit;
}

next() {
     if (fair) agent := (agent + 1) % N;
     else agent := *;
     assume(agent < N);
}

main() {
     init();

     while (true) {
          next();

          choice := *;
          if (choice = 1) vote();
          if (choice = 2) agree();
          ...

          check();
     }
}
```

FIGURE 4.6: Emulation program for the 2PC system.

at each iteration of the scheduler. However, the assumptions on the program counter of `agent` will prune away unfeasible executions by enforcing the entry conditions.

The nondeterministic choice of the agent to simulate models the interleaving of the behavioural processes of the agents. In addition, the scheduler may nondeterministically attempt to invoke a system-level emulation function (i.e., synchronisation in CCS). This linearises the concurrent execution of agent-level transitions ($S_1, \ldots, S_n$) and of system-level transitions and yields different advantages. First, it models the interleaving compactly: thanks to symbolic expressions and nondeterministic updates to the program counters, it can represent an exponential number of feasible executions. Second, it removes concurrency and therefore allows to use program analysis techniques that only support sequential programs. Third, it allows to model scheduling variations by simply restricting the nondeterminism over the interleaving of agents, i.e., by overriding the `next()` function. Currently, we provide a completely nondeterministic scheduler as well as a round-robin one, depending on the flag `fair`.

A function `check()` encodes the property ϕ to verify (lines 17–20). The code sample shows both types of encodings; however, in practice only one property at a time is encoded. In the case of an invariant property, we simply add the formula to the program as an assertion, which is checked at every step of the scheduler. For instance, if we want to instrument the program of Fig. 4.6 to verify whether the *fail* action is unreachable in the 2PC system, we simply replace the body of function `check()` with the statement `if (action =` *fail*`)` **`error;`**. To deal with emergent properties, we use the state formula as a termination condition for the whole program. Then, we can perform termination analysis on the generated program. Since the program can only terminate when ϕ holds, verifying that the program unconditionally terminates is equivalent to verifying that ϕ holds in $\mathbb{S}$. In our example program, we can check whether an *ok* action is always reached in the 2PC system with the statement `if (action =` *ok*`)` **`exit;`**. If the source language defines agents with individual states, we can easily support properties over them, possibly containing existential and universal quantifiers over the agents. To do so, we first apply quantifier elimination and then encode the resulting first-order formula into a predicate over the variables of $\mathbb{P}$ that encode the state of the agents.

Size of the emulation program. The encoding function of Sect. 4.1 generates at most one triple for each elementary action in the system (triples corresponding to restricted actions are removed), plus one triple for each pair of synchronising actions. The emulation program, then, will contain one emulation function for each of these triples. The size of the program counter for each agent is linear in the number of parallel processes in its behaviour.

Correctness sketch. We now sketch a correctness proof for our two-step encoding. Intuitively, such proof shows that, given a system $\mathbb{S}$ and an invariant property ϕ, an error state is reachable in the emulation program $\mathbb{P}$ generated from them if and only if $\mathbb{S}$ violates ϕ.

As shown in Figure 4.6, $\mathbb{P}$ is composed of a scheduler, and separate functions to emulate the transitions of $\mathbb{S}$. We assume that each emulation function correctly reproduces the semantics of its corresponding action by performing adequate assignments to the state variables of $\mathbb{P}$. At each iteration, the scheduler emulates a transition by calling one of these functions. The interleaving between agents is modelled by the nondeterministic assignment to the `agent` variable (lines 22–26). State variables are only altered within an emulation function, and the property check is performed right after each emulation step (line 39), so that a violation of ϕ in $\mathbb{S}$ will immediately cause an assertion failure in $\mathbb{P}$.

The problem, then, is to show that the flow of actions for the individual agents of $\mathbb{S}$ is preserved in $\mathbb{P}$. In particular, we wish $\mathbb{P}$ to reproduce any possible trace of $\mathbb{S}$ without introducing spurious executions. Intuitively, this is guaranteed in $\mathbb{P}$ by the entry guards and the exit assignments within the emulation functions: the entry guards restrict the possible emulation functions that can be invoked by the scheduler at the current emulation step; the exit assignments update the set of emulation functions from which the scheduler can nondeterministically pick at the next emulation step. For example, considering Figure 4.2a, right after executing action a, nondeterministically either b or c can be executed.

Lemma 4.14 (Completeness of Sequential Emulation)**.** *For each feasible execution f of $\mathbb{S}$, there exists a feasible execution g of $\mathbb{P}$ such that in g the emulation functions are invoked exactly in the same order as their corresponding actions in f.*

Proof. The proof is by induction on the length of the execution trace f of $\mathbb{S}$. In the following, we assume for simplicity that $\mathbb{S}$ is composed of a single agent.

In the `init()` function (lines 12–15), we initialise the program counter of $\mathbb{P}$ so that only the emulation functions for the *initial* actions of $\mathbb{S}$ are enabled, and thus only these functions can be executed in the first iteration of the scheduler (Section 4.2). Thus, the lemma holds for all traces of length 1.

Now, assuming that $\mathbb{P}$ correctly emulates $\mathbb{S}$ up to $n-1$ actions, let β be the n-th action for some trace of $\mathbb{S}$, and α the action immediately preceding β. We need to show that there exists an execution of $\mathbb{P}$ such that the $(n-1)$-th and n-th emulation steps invoke their emulation functions $\mathtt{f}_\alpha$ and $\mathtt{f}_\beta$, respectively. By the inductive hypothesis $\mathbb{P}$ emulates $\mathbb{S}$ up to $n-1$ actions, therefore there must exist an execution of $\mathbb{P}$ where $\mathtt{f}_\alpha$ is invoked at the $(n-1)$-th emulation step.

If β immediately follows α within a sequential composition in $\mathbb{S}$, the exit assignment of the program counter at the end of $\mathtt{f}_\alpha$ will match the guard at the beginning of $\mathtt{f}_\beta$, and thus $\mathtt{f}_\beta$ may be invoked right after emulating α.

If β and α occur as parallel actions in $\mathbb{S}$, then β might have been performed instead of α as the $(n-1)$-th action for some other trace of $\mathbb{S}$. In that case, by inductive hypothesis, $\mathtt{f}_\beta$ would have been invoked instead of $\mathtt{f}_\alpha$. This means that the guard for $\mathtt{f}_\beta$ is already satisfied at the $(n-1)$-th iteration of the scheduler. The exit assignments and entry guards within f_α and f_β in $\mathbb{P}$ work on different elements of the program counter, and thus cannot interfere with each other (Fig. 4.2b). Therefore, $\mathtt{f}_\beta$ can be executed right after $\mathtt{f}_\alpha$. □

Lemma 4.15 (Soundness of Sequential Emulation)**.** *For each feasible execution g of $\mathbb{P}$, there exists a feasible execution f of $\mathbb{S}$ such that the actions in f follow the same order in which the emulation functions are invoked in g.*

Proof. If g has length 1, then it is composed by a call to an emulation function. By construction of the intermediate representation $\mathbb{T}$, and of the start triple in particular, this function must necessarily correspond to an initial action of $\mathbb{S}$. Thus, all executions of length 1 satisfy the lemma.

Let us now assume by way of contradiction that there exists an execution of $\mathbb{P}$ where $\mathtt{f}_\gamma$ is invoked at the n-th emulation step right after $\mathtt{f}_\alpha$, but γ never follows α in any trace of $\mathbb{S}$. Assume, again, that $\mathbb{P}$ correctly emulates $\mathbb{S}$ up to the first $n-1$ actions (inductive hypothesis). If $\mathtt{f}_\gamma$ is called at the n-th iteration of the scheduler, then its guard must be satisfied. Two cases may apply: either the guard was also satisfied at the previous emulation step, or it was not.

In the first case, there exists another execution of $\mathbb{P}$ where $\mathtt{f}_\gamma$ is the $(n-1)$-th called function. By the inductive hypothesis, γ must be the $(n-1)$-th action for some trace of $\mathbb{S}$. If α and γ can both be the $(n-1)$-th action, either they are parallel actions or they occur within a nondeterministic choice in $\mathbb{S}$. If they are parallel actions, then we have found a contradiction: there must be a trace of $\mathbb{S}$ where γ follows α. On the other hand, if there is a nondeterministic choice between α and γ, then the exit assignment in $\mathtt{f}_\alpha$ is constructed so that the entry guard of $\mathtt{f}_\gamma$ can never be satisfied (Fig. 4.2a). Thus there cannot be any execution of $\mathbb{P}$ where $\mathtt{f}_\alpha$ is followed by $\mathtt{f}_\gamma$.

The second case implies that an exit assignment at the end of $\mathtt{f}_\alpha$ causes the guard of $\mathtt{f}_\gamma$ to be satisfied. To do that, the exit assignment of $\mathtt{f}_\alpha$ must take into account the guard of $\mathtt{f}_\gamma$. However, this only happens when α is the last action of some sub-process of $\mathbb{S}$, which in turn is sequentially composed with some other sub-process that has γ as a potential initial action. But then there must exist a trace where γ follows α. □

Theorem 4.16 (Correctness of Sequential Emulation of Invariant Properties)**.** *Assuming that ϕ is an invariant property, given the system specification $\mathbb{S}$ and the property ϕ, program $\mathbb{P}$ contains a reachable assertion failure if and only if ϕ does not hold in $\mathbb{S}$, i.e., there exists a feasible execution trace of $\mathbb{S}$ that violates ϕ.*

Proof. The theorem follows from Lemma 4.14 and Lemma 4.15. □

4.3 Summary

In this chapter, we have proposed a semantics-based technique that reduces property checking of distributed systems to verification of sequential programs. An encoding procedure translates the initial system under consideration and a property of interest into a sequential program, over which reachability or termination analysis can be performed. The analysis verdict attests whether the property of interest holds in the initial system. Section 4.1 presented the first step of our encoding procedure and introduced the concept of symbolically linked triple structures (SLTS), which we use as an intermediate representation between input specification and output programs. In Section 4.2, we described how an SLTS is translated into a sequential program, and we sketched a correctness proof of our encoding procedure.

Chapter 5

Prototype implementation

In this chapter we present SLiVER, a prototype tool for the analysis of multi-agent systems described in our specification language (Chapter 3) that relies on the encoding procedure introduced in Chapter 4. The source code of SLiVER is publicly available at the URL `https://github.com/labs-lang/sliver`, which also contains binary releases for the GNU/Linux operating system.

5.1 Usage

SLiVER may be invoked on a specification file `spec.labs` with the following command line:

```
sliver.py spec.labs [params] [--fair] [--backend b] [--steps s]
```

where square brackets denote optional parts of the command line. The `params` argument is a comma-separated sequence of `param=value` pairs. For instance, a valid value of `params` might be `n=5,size=10`. This argument must specify a value for every external parameter that appears in `spec.labs`. The `--fair` flag tells SLiVER to enforce round-robin scheduling of the agents (see Section 4.2). The user may select the desired analysis back end by adding `--backend b` to the command line. Currently, `b` must be either `cadp` [100], `cbmc` [59], `cseq` [138], or `esbmc` [97]. The default back end is CBMC. This choice was mostly driven by experience gained while developing both SLiVER itself and the LAbS$^+$ specifications shown in Section 3.4: namely, we found that bit-precise BMC with a shallow verification bound quickly discovered subtle bugs in the specifications, and also allowed us to detect errors in early prototypes of the encoding procedure itself.

Lastly, option `--steps s` tells SLiVER to verify the system up to s transitions. This is done by replacing the endless scheduler loop in the emulation program (Figure 4.6, line 31) with one

```
<initialization>
        fork[0] <-- 0
        fork[1] <-- 0
        fork[2] <-- 0
        fork[3] <-- 0
        fork[4] <-- 0
Phil 0: status <- 0
Phil 1: status <- 0
Phil 2: status <- 0
Phil 3: status <- 0
Phil 4: status <- 0
<end initialization>
Phil 0: fork[0] <-- 1
Phil 4: fork[4] <-- 1
Phil 4: status <- 1
Phil 3: fork[3] <-- 1
Phil 3: status <- 1
Phil 2: fork[2] <-- 1
Phil 2: status <- 1
Phil 1: fork[1] <-- 1
Phil 1: status <- 1
Phil 0: status <- 1
<property violated>
```

(A) A counterexample trace for property `NoDeadlock` of Listing 3.1a.

```
<initialization>
Node 0: leader <~ 3,0
Node 1: leader <~ 3,1
Node 2: leader <~ 3,2
<end initialization>
Node 0: leader <~ 0,3
Node 2: leader <~ 2,4
<Node 0: confirm 'leader'>
Node 1: leader <~ 0,3
<Node 0: end confirm 'leader'>
<Node 1: propagate 'leader'>
<Node 1: end propagate 'leader'>
<Node 0: propagate 'leader'>
<Node 0: end propagate 'leader'>
<Node 2: confirm 'leader'>
Node 0: leader <~ 2,4
Node 1: leader <~ 2,4
<Node 2: end confirm 'leader'>
<Node 0: propagate 'leader'>
<Node 0: end propagate 'leader'>
Node 0: leader <~ 0,8
<Node 0: propagate 'leader'>
Node 1: leader <~ 0,8
Node 2: leader <~ 0,8
<Node 0: end propagate 'leader'>
<property satisfied>
<deadlock>
```

(B) A simulation of the *leader election* system (Listing 3.1b).

FIGURE 5.1: Example of SLiVER outputs.

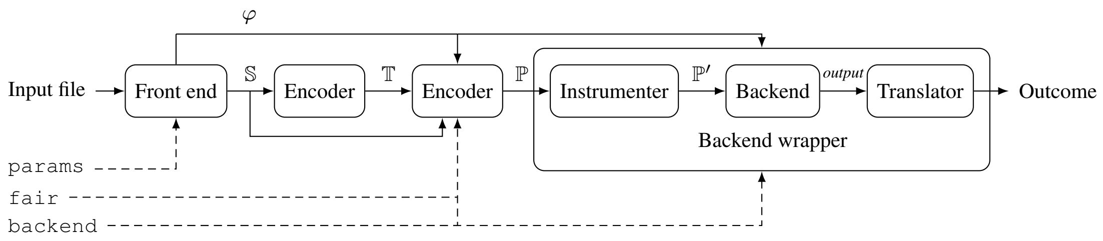
that exits after s iterations. If this option is omitted, or s is set to 0, SLiVER attempts to verify the full system.

The tool supports other flags, not shown above. If an invocation is enriched with `--verbose`, SLiVER will print the full output from the back end. The `--debug` flag enables the output of additional messages for diagnostic purposes. Finally, the `--show` flag forces SLiVER to print the emulation program and quit without performing any analysis.

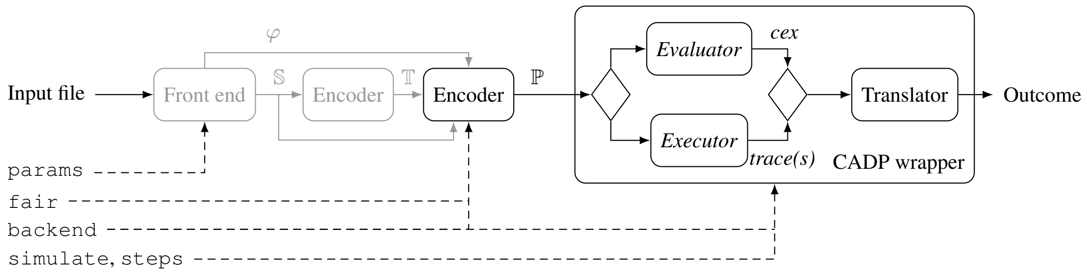
Example 5.1 (Verification of a LAbS$^+$ system)**.** Let us consider the *dining philosophers* system of Figure 3.1a. To verify whether a system of, say, 5 philosophers violates the invariant property `NoDeadlock`, we invoke SLiVER with the command line

```
sliver.py philosophers.labs n=5
```

The resulting counterexample is translated back into a LAbS$^+$-like syntax and shown to the user (Figure 5.1a). The output is divided into two parts: an initial section, delimited by `<initialization>...<end initialization>`, which describes the initial state of the system for the counterexample; and the actual trace, i.e., a sequence of LAbS$^+$ assignments that leads to a property violation.



(A) Encoding and analysis of C emulation programs.



(B) Encoding into LNT programs and analysis with CADP tools.

FIGURE 5.2: SLiVER analysis workflows.

5.2 Analysis workflow

In this section we describe how SLiVER analyses a given LAbS$^+$ specification. Roughly, the workflow involves performing the encoding procedure of Chapter 4 and then running an analysis backend on the resulting emulation program.

As of now, SLiVER may generate emulation programs in either the C programming language [141] or the LNT process calculus [103]. We chose C because of its relevance as a programming language and of the abundance of existing verification tools targeting it. We chose to also generate LNT programs for two reasons. We wanted to show that the procedure of Chapter 4 is general with respect to the language chosen for the emulation program, and we alo wanted to leverage the CADP analysis toolbox [100], which contains a wide range of tools for the simulation and verification of LNT programs and supports an expressive property language [170].

In this work, we have avoided encoding our specifications into formalisms that may be directly understood by a solver, such as the SMT-LIB input format [24]. This choice has several motivations. First, it allows us to experiment with a wide selection of analysis techniques provided by mature verification tools. Also, while SLiVER currently generates verification-oriented code, we envision that it may also generate executable code in the future. Thus, a behaviour could be first proved safe through formal verification, and then compiled to run on real agents. Having an encoding procedure that targets general-purpose languages makes this goal more reachable. The main drawback of this approach is that the tools which we use to verify the SLiVER-generated programs usually perform their own translation towards another format, which is then fed to a solver. If SLiVER directly performed its encoding towards such a format rather than using C (or

LNT) as an intermediate language, the resulting problem would likely be more compact and thus easier for the solver to analyse. However, this is not a fundamental limitation, as the architecture of our tool allows us to easily add a translation towards, e.g., SMT-LIB in the future.

C-based workflow. Figure 5.2a shows the workflow that SLiVER follows to encode and verify a specification as a C program. First, a front end parses and preprocesses a LAbS$^+$ input file (see Section 3.5). During the preprocessing step, external parameters in the input file are replaced with the values provided by the user via the `parameters` argument. The front end returns a *specification* $\mathbb{S}$ and a *property* φ.

Then, the two-step encoding described in Chapter 4 is performed. In the first step, the tool creates a triple structure for each `Behaviour` process defined in the specifications, and then merges them in a triple structure $\mathbb{T}$ that encodes the entire system. The second step of the encoding uses $\mathbb{T}$, along with φ and some information from $\mathbb{S}$ (such as the total number of agents to encode) to generate the emulation program $\mathbb{P}$. The `fair` flag specifies whether the emulation program should implement a round-robin scheduler (see Section 4.2).

Finally, the emulation program is passed to a backend wrapper. The wrapper first instruments the program for compatibility with the chosen backend (e.g., by using the correct syntax to model nondeterministic initialisation of variables, assertions, and assumptions). Then, it calls the backend analysis tool with appropriate settings. These settings may change according to the property φ to verify, as well as the values of `simulate` and `steps`. The wrapper also takes care of translating the output returned by the backend into a LAbS-like syntax before showing it to the user. This makes the output easier to understand and effectively hides the details of the emulation program from the end user.

LNT-based workflow. The workflow for LNT emulation programs is similar to the one used for C programs. The main difference is that it also supports *simulation* of the given system: if SLiVER is invoked with `--simulate n` (where `n` > 0) it will generate n simulation traces of the given system (provided that the chosen backend supports simulation). The length of each trace may be bounded to s by using the option `--steps s`.

The workflow makes use of two tools from the CADP toolbox, namely Evaluator and Executor. Evaluator, on the one hand, is a model checker that can evaluate properties expressed in the language MCL [170], a temporal logic based on the modal μ-calculus extended with regular action formulas. MCL also features value-passing constructs, but we currently do not use them. Executor, on the other hand, performs a bounded random exploration of the state space of a given program. Starting from the initial state, it repeatedly enumerates and then randomly chooses one of the transitions going out of the current state, until it has generated a sequence of the requested length. Explorations can be made reproducible by manually providing a seed for the internal pseudo-random number generator.

The LNT-based workflow is shown in Figure 5.2b, where all those components whose behaviour does not change with respect to the C-based workflow are dimmed. Of course, the second step of the encoder outputs a program written in LNT, rather than in C. Then, a CADP wrapper checks the value of the `simulate` option and launches the appropriate analysis tool. If `simulate` is omitted or set to 0, the wrapper invokes Evaluator to model-check $\mathbb{P}$. If a counterexample is found, a translation module converts it to a LAbS-like syntax and shows it to the user; otherwise, the user is notified that φ holds in $\mathbb{S}$. In case the user asked instead for one or more simulation traces, SLiVER calls Executor to generate them. Each trace is then translated back into LAbS$^+$ and shown to the user. Simulation traces will also contain information about an invariant property being violated or an emergent property becoming satisfied.

Example 5.2 (Simulation of a LAbS$^+$ system)**.** Now, assume that we want to simulate the evolution of the *leader election* system of Figure 3.1b. To do so, we invoke SLiVER with the following command line:

```
sliver.py leader.labs n=3 --backend cadp
        --simulate 1 --steps 100
```

This command tells SLiVER to generate an LNT emulation program from `spec.labs` and then pass it to the Executor tool provided by CADP. The program is instrumented so that SLiVER will print a message whenever property `LeaderIs0` is satisfied during the simulation. A possible output for this command is shown in Figure 5.1b. Again, the output describes first the initial state of the system, and then the steps of the simulation trace. Notice that all stigmergic assignments within the trace show both the new value and timestamp for the assigned variable. In the first steps of this trace, nodes 0 and 2 update `leader` to their respective ids. Then, node 0 sends a confirmation message for `leader`. It does so because it had to compute the guard `leader > id`. Node 1 picks up the message and updates its value of `leader` accordingly (lines 8–10). On the other hand, node 2 ignores the message, since its own value of `leader` has a higher timestamp. After a sequence of messaging rounds, during which node 0 sets `leader` to 2 (line 16), the same node updates yet again `leader` to 0 (line 21). Then, a propagation message from node 0 forces the other nodes to accept that value for `leader`, and property `LeaderIs0` becomes satisfied (line 26). At this point, the system reaches a deadlocked state since no agent is able to perform any action, and the trace ends (line 27).

Implementation details. The LAbS$^+$ parser and encoder are implemented in about 2500 lines of F#. The encoder generates emulation programs by relying on code templates which amount to

450 additional lines for each supported language (currently, C and LNT). The rest of SLiVER consists of roughly 1000 lines of Python. F# belongs to the ML family of strongly-typed functional programming languages [181], and thus provides advanced features such as (possibly recursive) algebraic data types, pattern matching, type inference, and higher-order functions [228]. These features were useful in the development of the LAbS$^+$ parser and the implementation of the encoding procedure. The Python layer of SLiVER accepts and validates user input, invokes the parser/encoder binary so as to generate the emulation program, feeds it to a verification back end, and performs the conterexample translation (if a counterexample is found). We chose to implement these facilities in Python due to its dynamic nature and its extensive standard library, which allow for rapid development.

5.3 Handling language-specific features

We now discuss how SLiVER encodes some features that are specific to LAbS and LAbS$^+$, and therefore were not covered by the general description of the encoding procedure.

Stigmergic interaction. According to the semantics of LAbS, an agent may send a propagation or confirmation message whenever it has at least one element in its sets of pending messages Zp and Zc. We encode this behaviour by inserting two additional emulation functions, `propagate` and `confirm`, into the emulation program. These functions nondeterministically pick up an agent with a pending message and emulate the stigmergic message-passing transitions formalised by the rules of Table 3.8. At each iteration, the scheduler may decide to call one of these functions (assuming that there is at least one agent with a pending message), or another emulation function instead. This models the asynchronous nature of stigmergic interaction.

Guards. As described in Section 3.2, LAbS allows to define a guarded process $g \rightarrow P$ that behaves as P on the condition that predicate g holds. We encode this construct by decorating emulation functions with additional assumption statements that model the guards in the input specifications. Consider for instance the *leader election* system specified by Figure 3.1b. Agents in that system follow a guarded recursive behaviour: they can assign their `id` to variable `leader`, but only if the `id` of the current `leader` is greater than theirs (Figure 5.3a). The triple structure for this behaviour has a start triple, plus another triple that encodes action `leader` $\leftsquigarrow$ `id`. SLiVER encodes the latter by means of the emulation function shown in Figure 5.3b. There, `L[i][0]` stores the value of the stigmergic variable *leader* for the `i`-th agent, and `Ltstamp[i][0]` contains its associated timestamp. Note that the first assumption statement in the function (line 2) compares the active agent's value of *leader* with its own *id*. If this assumption holds, then the emulation function may proceed by first evaluating the entry condition (line 3),

```
Behaviour =
  leader > id ->
    leader <~ id;
    Behaviour
```

(A) LAbS$^+$ specification.

```
1   void emulation_fn() {
2     assume(L[id][0] > id);
3     assume(pc[id][0] == 1);
4
5     // Implement rule LSTIG (Table 3.7)
6     assume(Zc = Zp = ∅);
7     L[id][0] = id;
8     Ltstamp[id][0] = tod();
9     Zp[id][0] = 1;
10
11    pc[id][0] = 1;
12  }
```

(B) Guarded emulation function for `leader` $\leftsquigarrow$ `id`.

FIGURE 5.3: Encoding LAbS$^+$ guards in C.

then by performing action `leader` $\leftsquigarrow$ `id` (lines 7–9), and finally by updating the program counter (line 11).

To properly decorate emulation functions, SLiVER performs a separate static analysis on each agent's behaviour within the input specifications. The result of this analysis is a mapping γ from elementary actions to predicates. Then, whenever we have to encode a triple t into an emulation function, SLiVER adds an assumption statement `assume`$(\gamma(\mu(t)))$ at the beginning of the function body. The static analysis may be formalised as a function Γ (5.1) that recursively accumulates guards while visiting the structure of a LAbS process.

$$\begin{aligned}
\Gamma\,[\mu]^g &\triangleq \langle \mu, g\rangle \\
\Gamma\,\big[g' \to P\big]^g &\triangleq \Gamma\,[P]^{g\wedge g'} \\
\Gamma\,[P;Q]^g &\triangleq \Gamma\,[P]^g \cup \Gamma\,[Q]^{\mathit{true}} \\
\Gamma\,[\textstyle\sum_i P_i]^g &\triangleq \textstyle\bigcup_i \Gamma\,[P_i]^g \\
\Gamma\,[\textstyle\prod_i P_i]^g &\triangleq \textstyle\bigcup_i \Gamma\,[P_i]^{g\vee\exists j.run(P_j)}
\end{aligned} \tag{5.1}$$

When recursion reaches an elementary action μ, the latter is mapped to the currently accumulated guard g. When we encounter a guarded process $g' \to P$, we accumulate g' into the current guard. The other rules state that guards only affect the first process in a sequential composition $P;Q$, and distribute over the choice operator. Finally, let us consider a guarded parallel process $g \to P$, where $P = P_1 \mid \ldots \mid P_n$. We cannot simply distribute g over its child processes, because g only needs to be satisfied once to enable all of the child processes of P. Rather, we define a predicate $run(P_i)$ which holds if and only if the child process P_i has started its execution by performing the first transition. This predicate may be simply implemented as a check on the program counter. Then, we can recur on each child process by using $g \vee \exists j.run(P_j)$ as accumulated guard. This disables the guard g on the actions of P_i, as soon as one of its siblings P_j performs an action.

Quantified properties. As described in Section 3.5.2, a LAbS$^+$ property may contain universal or existential quantifiers over the agents. SLiVER currently applies quantifier elimination [230] to obtain a quantifier-free predicate, which is then encoded in the emulation program. In principle, this approach may be applied to formulas with any combination of existential and universal quantifiers. However, in practice it produces inconveniently long predicates whenever alternate quantifiers are involved. Therefore, SLiVER currently limits its support to alternation-free properties.

5.4 Experimental evaluation

We evaluate our approach to the verification of multi-agent systems by using SLiVER to encode a selection of LAbS$^+$ specifications from Section 3.4 as C or LNT programs, and then attempting to verify them with a selection of existing tools.

5.4.1 Benchmark description and experimental setup

We now describe our selection of benchmark verification tasks. First, we list the programs under analysis and their corresponding LAbS$^+$ specifications. Then, we informally describe the properties of interest and provide their encoding in the LAbS$^+$ property language.

- `approx-a` and `approx-b` encode the *approximate majority* system of Figure 3.7 with parameters ($\_\mathtt{yes} = 1, \_\mathtt{no} = 2$) and ($\_\mathtt{yes} = 2, \_\mathtt{no} = 3$), respectively.
- `boids` encodes the system of flocking agents of Figure 3.4, with parameters ($\_\mathtt{birds} = 3, \_\mathtt{size} = 5, \_\mathtt{delta} = 5$). For this system we assume round-robin scheduling.
- `flock` encodes the simpler system of flocking agents shown in Figure 3.2, with parameters ($\_\mathtt{birds} = 3, \_\mathtt{size} = 5, \_\mathtt{delta} = 5$). For this system we assume round-robin scheduling.
- `formation` encodes the line-forming system of Figure 3.6, instantiated with parameters ($\_\mathtt{size} = 10, \_\mathtt{n} = 3, \_\mathtt{range} = 2$). For this system we assume round-robin scheduling.
- `maj` encodes the *majority protocol* of Figure 3.8 with parameters ($\_\mathtt{yes} = 1, \_\mathtt{no} = 2$).

The *invariance* experiments consist of the following verification tasks:

- For `approx-a` and `approx-b`, we verify that the agents never reach a consensus on the minority opinion. This is equivalent to checking that there is always some agent that does *not* agree on that opinion. Thus, we use the following LAbS$^+$ property:

  ```
  always exists Yes y, exists No n,
  state of y != 1 or state of n != 1,
  ```

  where variable `state` represents the opinion of an agent, and value 1 encodes the minority opinion.

- For `maj`, we similarly want to verify that a consensus on the minority opinion is unreachable. Since this opinion is encoded as two separate values of the `state` variable, we have to use a slightly different LAbS$^+$ property:

  ```
  always exists Yes y, exists No n,
  (state of y = 0 or state of y = 2) or
  (state of n = 0 or state of n = 2),
  ```

  which informally means that there is always at least one agent that follows the majority opinion.

- For `formation`, we check that no robot ever reaches a position outside of the range $[0, \texttt{\_size} - 1]$. This is encoded by a predicate on their `pos` attribute:

  ```
  always forall Robot r,
  pos of r >= 0 and pos of r < _size.
  ```

The *emergence* experiments are the following:

- For `formation`, we check that all robots eventually are no closer than `_range` from each other:

  ```
  finally forall Robot a, forall Robot b,
  id of a = id of b or
  abs(pos of a - pos of b) >= _range.
  ```

  We add the `id of a = id of b` clause to bypass the check whenever `a` and `b` refer to the same robot.

- For `flock`, we check that all agents eventually move in the same direction:

  ```
  finally forall Bird b1, forall Bird b2,
  dirX of b1 = dirX of b2 and
  dirY of b1 = dirY of b2.
  ```

- For `boids`, we check that all agents eventually select the same leader:

```
finally forall Bird b1, forall Bird b2,
leader of b1 = leader of b2.
```

- For `maj`, we check that all agents eventually agree on the majority opinion:

```
finally forall Yes y, forall No n,
(state of y = 0 or state of y = 2) and
(state of n = 0 or state of n = 2).
```

The techniques and tools used in the experimental evaluation have been described in Sections 2.4.1 and 2.4.2. Specifically, we used the following releases:

- 2LS [55]: version 0.7.0;
- CADP [100]: version 2020-d;
- CBMC [59]: version 5.4 for the invariance benchmark and a competition release, submitted to SV-COMP 2019, for emergence benchmarks;
- CPAchecker [28]: version 1.8;
- ESBMC [97]: version 5.1.0 for BMC experiments and version 6.4.0 for k-induction experiments;
- Seahorn [117]: version 0.1.0-rc3-7cba65c9;
- SMACK [205]: version 2.4.0;
- Symbiotic [54]: version 6.0.3;
- Ultimate Automizer [121]: competition release, submitted to SV-COMP 2020;
- VVT [116]: competition release, submitted to SV-COMP 2016.

In a preliminary experimental phase, we also tried other tools, namely: AProve [106] (which performs termination analysis based on symbolic execution), HiFrog [9] (SMT-based function summarisation), IKOS [45] (abstract interpretation), and Kratos [56] (lazy predicate abstraction). However, these tools did not produce any conclusive results on our programs and thus we did not consider them for the final selection of verification tasks. It may be interesting to investigate whether future versions of these tools would be able to provide conclusive verdicts for our emulation programs.

By choosing different configurations for some of these tools, we end up with 48 and 16 verification tasks for the invariance and emergence benchmarks, respectively. All the experiments were performed on a dedicated 64-bit GNU/Linux workstation with kernel 4.9.95, equipped with 128GB of physical memory and a dual 3.10GHz Xeon E5-2687W 8-core processor. We set a time limit of 12 hours and a memory limit of 32 GB for all experiments.

TABLE 5.1: Results of the *invariance* verification tasks for C emulation programs. –[a]: Timeout (12 hours). –[b]: Inconclusive analysis reported by the tool. –[c]: Out of memory (32 GB). –*: The tool requires an array-free encoding.

Techniques	Systems			
	`formation`	`approx-a`	`approx-b`	`maj`
Symbolic execution (Symbiotic)	0.01 ✓	206.83 ✗	60.47 ✗	–[a]
Bit-precise BMC (CBMC)	–[a]	0.01 ✗	0.01 ✗	–[a]
Word-level BMC (ESBMC)	–[a]	0.08 ✗	0.07 ✗	–[a]
Word-level BMC (SMACK)	–[a]	0.67 ✗	1.83 ✗	–[a]
Explicit-value analysis (CPAchecker)*	–[c]	337.08[b]	–[c]	1.03 ✓
Predicate abstraction+CEGAR (CPAchecker)*	0.34 ✓	0.08 ✗	0.03 ✗	–[c]
Automata+CEGAR (Automizer)	5.98 ✓	1.17 ✗	45.7 ✗	–[a]
k-induction (CPAchecker)*	–[c]	0.17 ✗	0.01 ✗	–[c]
k-induction (2LS)*	0.01 ✓	0.12 ✗	0.05 ✗	–[a]
k-induction (ESBMC)	0.01 ✓	0.01 ✗	0.05 ✗	0.01 ✓
PDR (Seahorn)	0.3 ✓	0.03 ✗	0.5 ✗	4.67 ✓
PDR (VVT)	0.03 ✓	0.01 ✗	0.01 ✗	0.01 ✓
	✓	✗	✗	✓

5.4.2 Experimental results: C emulation programs

Tables 5.1 and 5.2 report our experimental results in invariance (reachability) and emergence (termination) analysis of C emulation programs. In both tables, the top row and the leftmost column refer to the considered multi-agent system and the program analysis technique (along with the specific implementation), respectively. For each tool, we cite the specific implementation and, when possible, the analysis technique that it implements. As pointed out in Section 2.4.2, some of these tools (marked by an asterisk in both tables) have little to no support for C arrays: in order to use them, we provided an equivalent, array-free version of the programs under verification. The bottom of the table reports the verdict we were able to draw by only inspecting the experimental results, without exploiting any previous knowledge of the benchmarks. In the internal cells of the two tables, we report the partial verdicts along with the decision time (in minutes) for each tool and system. Conclusive results are marked with ✓ or ✗ to respectively denote that the property under analysis was successfully verified or violated. Superscripts provide further details on the inconclusive experiments.

Main insights. By looking at the separate columns of both Tables 5.1 and 5.2, we observe that for each of the systems under consideration at least one tool is able to generate a conclusive verification verdict, and that the conclusive verdicts for a given system are always consistent. Therefore, we can confidently draw a verdict for every system. Interestingly, our verdicts do confirm all the known results from the literature: specifically, for `approx-a`, `approx-b`, and maj we can confirm the findings in [16, 37]. Even more interestingly, we manage to successfully verify an emergent property for boids. It is worth observing that this requires analysis of the flock's behaviour up to an unbounded number of steps. Besides simulation [190], successful analysis of this system was previously limited to the bounded case [83]. We are not aware of previous attempts of unbounded verification of this system.

Invariance benchmarks. As shown in Table 5.1, symbolic execution [143] gives correct results but seems to have issues with performance. The analysis of both approx-a and approx-b takes three and one hour, respectively, and the tool is not able to verify maj due to timeout.

Bounded model checking [31] is consistently quick in detecting property violations, with either SAT or SMT decision procedures, and all the considered tools for this category were able to generate precise violation witnesses. However, this technique alone cannot provide conclusive results in the absence of property violations, such as formation and maj. For these systems, we repeatedly increased the verification bound until timing out, to double-check the consistency with the other approaches.

Abstraction-based analysers [30, 110, 121] seem to complement these limitations, as they can successfully determine the safety for formation and maj in half of the cases. In contrast, the results on the unsafe instances (confidently claimed as such via bounded analysis) are sometimes inconclusive. Analysis procedures based on CEGAR loops [62] correctly identify the approx programs as unsafe and the formation program as safe, but they run into memory or time limits issues when considering the more complex maj program. This confirms once again that under- and over-approximation are orthogonal to each other.

Interestingly, we observe the superiority of inductive techniques, i.e., k-induction[1] [219] and property directed reachability [44], over the other approaches we considered. These techniques exhibit outstanding performances with consistent verdicts; produce precise witnesses for violated properties, with comparable performances to the fastest bounded model checker; and are competitive, if not superior, to abstraction-based tools on safe systems.

Emergence benchmarks. In Table 5.2 we observe overall less data points as well as a smaller portion of conclusive verdicts with respect to Table 5.1. In fact, the presence of arrays and non-linear operations appears to be a major hindrance for the tools that we have considered for termination analysis. Nevertheless, we do manage to find at least one conclusive verdict for each verification task except maj. In particular, 2LS can confirm the termination of flock when using the *equalities* abstract domain. We also consider a competition release of CBMC, which

[1]As implemented by ESBMC 6, which also infers invariants through interval analysis. As shown in Table 5.1, the other implementations that we have considered (CPAchecker and 2LS) sometimes report inconclusive results on safe systems.

TABLE 5.2: Results of the *emergence* verification tasks for C emulation programs. $–^a$: Timeout (12 hours). $–^b$: Inconclusive analysis reported by the tool. $–^*$: The tool requires an array-free encoding.

	Systems			
Techniques	`formation`	`flock`	`boids`	`maj`
Symbolic execution (Symbiotic)	488.40^b	$–^a$	$–^a$	338.67^b
BMC+completeness threshold (CBMC)	214.32 ✓	243.84 ✓	49.15 ✓	$–^a$
Summarization+intervals (2LS)*	0.08^b	0.03^b	32.15^b	0.01^b
Summarization+equalities (2LS)*	319.40^b	107.72 ✓	$–^a$	0.38^b
	✓	✓	✓	

TABLE 5.3: Experimental results for LNT emulation programs. $–^a$: Out of memory (32 GB).

(A) Invariance benchmarks.

`formation`	`approx-a`	`approx-b`	`maj`
$–^a$	0.03 ✕	15.1 ✕	0.12 ✓

(B) Emergence benchmarks.

`formation`	`flock`	`boids`	`maj`
$–^a$	$–^a$	$–^a$	0.13 ✓

performs termination analysis by repeatedly unrolling all loops with an increasing bound and using BMC to check unwinding assertions until a completeness threshold is found [63]. This approach can verify the termination for all systems under verification except `maj`.

5.4.3 Experimental results: LNT emulation programs

In order to assess the capabilities of our encoding into LNT programs, we generated an LNT emulation program for each system in our benchmark selection. We then used the Evaluator explicit-state model checker to perform all verification tasks. The results (Tables 5.3a and 5.3b) show that this approach appears to be competitive with C-based back ends on systems without stigmergic interaction, namely `approx-a`, `approx-b`, and `maj`: in these cases, the model checker is able to provide verdicts that are always consistent with those presented in Section 5.4.2. Interestingly, Evaluator is also able to prove an emergent property on `maj`, unlike C-based tools. On the other hand, systems that contain one or more stigmergic variables are not as amenable to verification, with Evaluator hitting the memory limit (32GB) on all tasks. In this case, the symbolic techniques implemented by C-based back ends seem to have an advantage. This is likely a consequence of the large state space of these systems, which in turn is due to the asynchronous nature of stigmergic interaction. Our sequential emulation program encodes asynchrony (as well as concurrency) through non-deterministic assignments, which may not be a good fit for the enumeration-based approach of Evaluator.

5.5 Summary

In this chapter, we have described SLiVER, our implementation of the encoding procedure described in Chapter 4 for LAbS$^+$ specifications. The tool implements the encoding procedure described in Chapter 4, and thus shows that this procedure can be fully automated. The tool also proves that our intermediate representation allows us to generate emulation program in a variety of languages. In Section 5.1, we described how to invoke SLiVER from the command line to verify or simulate a LAbS$^+$ system. Section 5.2 showed the workflows used by SLiVER to generate and analyse C and LNT programs, while Section 5.3 discussed how we supported some language features that are specific to LAbS$^+$. Lastly, in Section 5.4, we showed that our tool can be effectively used to verify both *invariant* and *emergent* properties of multi-agent systems, by relying on a selection of mature program verification tools.

Chapter 6

Conclusions

In this work, we have pointed out that the multi-agent system paradigm is appropriate for modelling many scenarios of interest across several research areas.

Modelling an existing system may allow researchers to test hypotheses and gain a deeper insight into their systems of interest. At the same time, agent-based models may be used as an artefact to design new systems and analyse their correctness before they are built and deployed. In any case, the complexity of the scenarios of interest is ever-increasing. First, the interactions between agents may give rise to unexpected, even surprising consequences even though each agent may have a very simple behaviour. Furthermore, many interesting features of these systems are precisely an emergent result of this apparently chaotic evolution. These observations rule out informal reasoning for all but the simplest systems, and call for innovative *languages* and *tools* to specify and formally analyse complex multi-agent systems.

To address these needs, we have introduced a formal language built around the concept of virtual stigmergies. This form of asynchronous, indirect communication may be the key to concisely model many complex scenarios of interest. Specifically, we have provided a core process algebra, LAbS that formally specifies how agents perform their actions and communicate (via either stigmergic or shared variables). Then, we have introduced a specification language, LAbS$^+$, that allows to fully describe a multi-agent system. Besides the behavioural specifications, written in a machine-readable version of LAbS, this language allows to specify other important information, such as the initial state of the system, its composition, and the properties it should satisfy.

We have then described a semantics-based encoding procedure that translates specifications of concurrent systems into sequential imperative programs. The procedure relies on an intermediate representation where a *process* (i.e., the behaviour of an agent within the system) is split into its elementary actions, while its structure is preserved by symbolic links between the actions. This representation is then used to construct the sequential program. We have shown that relevant collective properties of a given concurrent system may be verified by (i) encoding the system specifications as a sequential program through this procedure, (ii) decorating this program with statements encoding the properties of interest, and (iii) running a verification tool developed for the language used to write the program.

To prove effectiveness of our approach, we developed SLiVER, a prototype implementation of our encoding technique for the LAbS$^+$ language. We used SLiVER to generate a collection of sequential programs that encode several real-world multi-agent systems in either the C or the LNT language, and showed that the generated programs are amenable to formal verification by means of multiple state-of-the-art verification tools. We are not claiming that our results represent an actual breakthrough in formal verification of multi-agent systems. In fact, verifying systems larger than those featured in our experimental evaluation is still a challenging task. However, our approach is definitely competitive with tailored end-to-end platforms for MAS modelling and analysis. Namely, while some of these platforms may provide more advanced features such as parameterised verification [37, 149], they either restrict the capabilities of the agents and of the overall system under analysis (e.g., by disallowing value-passing), or limit the range of supported properties, or bind themselves to a specific verification technique. Our approach does not have these restrictions: in fact, it may support specification languages featuring complex interactions, such as those of attribute-based stigmergic communication. Furthermore, the ease with which we can naturally support new verification techniques leads us to believe that sequential emulation, together with ever-improving back ends, may have long-term relevance. Finally, the ability to adapt the encoding procedure to different domain-specific languages may allow scholars from different research disciplines to formally verify their agent-based models.

Future directions

Obviously, our work could be extended along many different directions. Below we just sketch some research ideas that in our opinion deserve further attention.

Language. The specification language could be extended in several ways. For instance, we could add a behavioural primitive allowing an agent to *spawn* new ones, so as to describe open-ended systems. With respect to stigmergic interactions, we have described a mechanism where values are timestamped and agents always prefer the value with a higher timestamp. However, one may think of different mechanisms, e.g., letting agents prefer the minimum (or maximum) value, which may be better suited for some scenarios. It would be interesting to explore such alternatives and find a way to include them within a LAbS$^+$ specification. We could also introduce language constructs to enable reuse and modularity of the specifications, such as a macro system, parameterised process invocation, or the capability to include code from other files. Most of these new functionalities may be added without revising the formal semantics of the language. They would enable us to develop standard libraries of behaviours and patterns (e.g., for link predicates) commonly observed in the multi-agent literature, and would in general simplify the specification of complex systems.

Encoding. The correctness proof sketched in Section 4.2 may be formalised as a behavioural equivalence between the LTS of the input system (as given by its semantics) and the evolution condition of its corresponding triple structure. Currently, our encoding requires a one-time manual effort: one has to write a "template" code block for each action in the input language. Such a template is then used to construct emulation functions. It is reasonable to expect that this effort could be mechanised as well (at least for a chosen target language): given a machine-readable SOS of the action, a tool could automatically generate templates for the corresponding emulation functions. While the second step of our procedure currently generates sequential programs, it could be adapted so as to generate concurrent programs instead. Broadly speaking, such a program would instantiate a sub-process for each agent and have them evolve concurrently: this might be beneficial when efficient verification techniques may be applied on the generated concurrent program. In the case of LNT, compositional verification [102, 156, 157] might be one such technique. It would also be interesting to investigate other fairness guarantees besides plain round-robin scheduling, and how they may be enforced in both sequential and concurrent programs. Understanding whether the proposed encoding may support verification of more expressive properties (e.g., arbitrary LTL formulas) would be another research direction worth pursuing.

Implementation. The SLiVER tool may be improved in several ways so as to efficiently verify large multi-agent systems. We could investigate parallel and distributed implementations for some of the verification techniques reviewed in Section 2.4.1, such as bounded model checking [137], property directed reachability [166], and explicit-state model checking [101]. Moreover, to deal with even larger systems, we could exploit the simulation capabilities of SLiVER to implement statistical model checking techniques [218].

Bibliography

[1] Y. Abd Alrahman, R. De Nicola, and G. Garbi. GoAt: Attribute-based interaction in Google Go. In T. Margaria and B. Steffen, editors, *8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 11246 of *LNCS*, pages 288–303. Springer, 2018. doi:10.1007/978-3-030-03424-5_19.

[2] Y. Abd Alrahman, R. De Nicola, and M. Loreti. A calculus for collective-adaptive systems and its behavioural theory. *Information and Computation*, 268, 2019. doi:10.1016/j.ic.2019.104457.

[3] G. Agha and C. Hewitt. *Actors: A conceptual foundation for concurrent object-oriented programming*. MIT Press, 1987.

[4] P. E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In K. D. Forbus and H. E. Shrobe, editors, *6th National Conference on Artificial Intelligence (AAAI)*, pages 268–272. Morgan Kaufmann, 1987.

[5] P. E. Agre and D. Chapman. What are plans for? *Robotics and Autonomous Systems*, 6 (1-2):17–34, 1990. doi:10.1016/S0921-8890(05)80026-0.

[6] S. B. Akers Jr. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6): 509–516, 1978. doi:10.1109/TC.1978.1675141.

[7] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2): 123–154, 1984. doi:10.1016/0004-3702(84)90008-0.

[8] J. F. Allen and J. A. G. M. Koomen. Planning using a temporal world model. In A. Bundy, editor, *8th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 741–747. William Kaufmann, 1983.

[9] L. Alt, S. Asadi, H. Chockler, K. Even-Mendoza, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. HiFrog: SMT-based function summarization for software verification. In A. Legay and T. Margaria, editors, *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10206 of *LNCS*, pages 207–213. Springer, 2017. doi:10.1007/978-3-662-54580-5_12.

[10] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 340–351. Springer, 1997. doi:10.1007/3-540-63166-6_34.

[11] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126 (1):3–30, 1994. doi:10.1016/0304-3975(94)90266-6.

[12] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. In S. Chaudhuri and S. Kutten, editors, *23rd Symposium on Principles of Distributed Computing (PODC)*, pages 290–299. ACM, 2004. doi:10.1145/1011767.1011810.

[13] D. Angluin, J. Aspnes, and D. Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008. doi:10.1007/s00446-008-0059-z.

[14] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In A. Valmari, editor, *13th International SPIN Workshop (SPIN)*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006. doi:10.1007/11691617_9.

[15] J. Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010. doi:10.1145/1810891.1810910.

[16] J. Aspnes and E. Ruppert. An introduction to population protocols. In B. Garbinato, H. Miranda, and L. E. T. Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer, 2009. doi:10.1007/978-3-540-89707-1_5.

[17] R. L. Axtell, J. M. Epstein, J. S. Dean, G. J. Gumerman, A. C. Swedlund, J. Harburger, S. Chakravarty, R. Hammond, J. Parker, and M. Parker. Population growth and collapse in a multiagent model of the Kayenta Anasazi in Long House Valley. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7275–7279, 2002. doi:10.1073/pnas.092080799.

[18] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *1st International Conference on Information Processing*, pages 125–131. UNESCO, 1959.

[19] A. Bakst, K. von Gleissenthall, R. G. Kici, and R. Jhala. Verifying distributed programs via canonical sequentialization. *PACMPL*, 1(OOPSLA):110:1–110:27, 2017. doi:10.1145/3133934.

[20] M. Ballerini, N. Cabibbo, R. Candelier, A. Cavagna, E. Cisbani, I. Giardina, V. Lecomte, A. Orlandi, G. Parisi, A. Procaccini, M. Viale, and V. Zdravkovic. Interaction ruling animal

collective behavior depends on topological rather than metric distance: Evidence from a field study. *Proceedings of the National Academy of Sciences*, 105(4):1232–1237, 2008. doi:10.1073/pnas.0711437105.

[21] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286 (5439):509–512, 1999. doi:10.1126/science.286.5439.509.

[22] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *4th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005. doi:10.1007/11804192_17.

[23] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-825.

[24] C. W. Barrett, L. M. de Moura, S. Ranise, A. Stump, and C. Tinelli. The SMT-LIB initiative and the rise of SMT - (HVC 2010 award talk). In S. Barner, I. G. Harris, D. Kroening, and O. Raz, editors, *6th International Haifa Verification Conference (HVC)*, volume 6504 of *LNCS*, page 3. Springer, 2010. doi:10.1007/978-3-642-19583-9_2.

[25] J. A. Bergstra and J. Tucker. Top-down design and the algebra of communicating processes. *Science of Computer Programming*, 5(2):171–199, 1985. doi:10.1016/0167-6423(85)90010-3.

[26] J. A. Bergstra, J. W. Klop, and J. V. Tucker. Algebraic tools for system construction. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, E. Clarke, and D. Kozen, editors, *Workshop on Logics of Programs*, volume 164 of *LNCS*, pages 34–44. Springer, 1983. ISBN 978-3-540-12896-0 978-3-540-38775-6. doi:10.1007/3-540-12896-4_353.

[27] D. Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 11429 of *LNCS*, pages 133–155. Springer, 2019. doi:10.1007/978-3-030-17502-3_9.

[28] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16.

[29] D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In T. Margaria and B. Steffen, editors, *7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 9952 of *LNCS*, pages 195–211. Springer, 2016. doi:10.1007/978-3-319-47166-2_14.

[30] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In V. Cortellessa and D. Varró, editors, *16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 7793 of *LNCS*, pages 146–162. Springer, 2013. doi:10.1007/978-3-642-37057-1_11.

[31] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0_14.

[32] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003. doi:10.1016/S0065-2458(03)58003-2.

[33] J. Birgmeier, A. R. Bradley, and G. Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In A. Biere and R. Bloem, editors, *26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 831–848. Springer, 2014. doi:10.1007/978-3-319-08867-9_55.

[34] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th Symposium on Operating System Principles (SOSP)*, volume 21, pages 123–138. ACM, 1987. doi:10.1145/41457.37515.

[35] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993. doi:10.1145/163298.163303.

[36] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. B. Jackson, editors, *22nd International Conference on Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010. doi:10.1007/978-3-642-14295-6_31.

[37] M. Blondin, J. Esparza, and S. Jaax. Peregrine: A tool for the analysis of population protocols. In H. Chockler and G. Weissenbacher, editors, *30th International Conference on Computer Aided Verification (CAV)*, volume 10981 of *LNCS*, pages 604–611. Springer, 2018. doi:10.1007/978-3-319-96145-3_34.

[38] E. Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002. doi:10.1073/pnas.082080899.

[39] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm intelligence - From natural to artificial systems*. Studies in the Sciences of Complexity. Oxford University Press, 1999. ISBN 978-0-19-513159-8.

[40] R. H. Bordini, J. F. Hübner, and R. Vieira. Jason and the golden fleece of agent-oriented programming. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 3–37. Springer, 2005.

[41] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.

[42] R. H. Bordini, J. F. Hbner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007. ISBN 978-0-470-06184-8. doi:10.1002/9780470061848.

[43] J. C. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3, pages 721–756. North-Holland, 2007. doi:10.1016/s1570-2464(07)80015-2.

[44] A. R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

[45] G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In D. Giannakopoulou and G. Salaün, editors, *12th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 8702 of *LNCS*, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.

[46] M. Bratman. *Intention, plans, and practical reason*. Harvard University Press, 1987. ISBN 978-0-674-45818-5.

[47] R. A. Brooks. Intelligence without reason. In *12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 569–595. Morgan Kaufmann, 1991. ISBN 1-55860-160-0.

[48] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1-3):139–159, 1991. doi:10.1016/0004-3702(91)90053-M.

[49] E. Buchanan, A. Pomfret, and J. Timmis. Dynamic task partitioning for foraging robot swarms. In M. Dorigo, M. Birattari, X. Li, M. López-Ibáñez, K. Ohkura, C. Pinciroli, and

T. Stützle, editors, *10th International Conference on Swarm Intelligence (ANTS)*, volume 9882 of *LNCS*, pages 113–124. Springer, 2016. ISBN 978-3-319-44426-0 978-3-319-44427-7. doi:10.1007/978-3-319-44427-7_10.

[50] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In T. Vojnar and L. Zhang, editors, *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 11428 of *LNCS*, pages 21–39. Springer, 2019. doi:10.1007/978-3-030-17465-1_2.

[51] V. Buravlev, R. De Nicola, and C. A. Mezzina. Tuple spaces implementations and their efficiency. In *18th International Conference on Coordination Models and Languages (COORDINATION)*, volume 9686 of *LNCS*, pages 51–66. Springer, 2016. ISBN 978-3-319-39518-0. doi:10.1007/978-3-319-39519-7_4.

[52] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Symposium on Logic in Computer Science (LICS)*, pages 428–439. IEEE, 1990. doi:10.1109/LICS.1990.113767.

[53] B. F. Caviness, J. R. Johnson, B. Buchberger, and G. E. Collins, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Texts and Monographs in Symbolic Computation. Springer, 1998. doi:10.1007/978-3-7091-9459-1.

[54] M. Chalupa, T. Jasek, L. Tomovic, M. Hruska, V. Soková, P. Ayaziová, J. Strejcek, and T. Vojnar. Symbiotic 7: Integration of Predator and more - (competition contribution). In A. Biere and D. Parker, editors, *26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12079 of *LNCS*, pages 413–417. Springer, 2020. doi:10.1007/978-3-030-45237-7_31.

[55] H.-Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising interprocedural bit-precise termination proofs. In M. B. Cohen, L. Grunske, and M. Whalen, editors, *30th International Conference on Automated Software Engineering (ASE)*, pages 53–64. IEEE, 2015. doi:10.1109/ASE.2015.10.

[56] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - A software model checker for SystemC. In G. Gopalakrishnan and S. Qadeer, editors, *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 310–316. Springer, 2011. doi:10.1007/978-3-642-22110-1_24.

[57] F. Ciocchetta and J. Hillston. Bio-PEPA: An extension of the process algebra PEPA for biochemical networks. *Electronic Notes in Theoretical Computer Science*, 194(3): 103–117, 2008. doi:10.1016/j.entcs.2007.12.008.

[58] F. Ciocchetta and J. Hillston. Bio-PEPA for epidemiological models. *Electronic Notes in Theoretical Computer Science*, 261:43–69, 2010. doi:10.1016/j.entcs.2010.01.005.

[59] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 168–176. Springer, 2004. ISBN 1045389020279. doi:10.1007/978-3-540-24730-2_15.

[60] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logics of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1981. doi:10.1007/BFb0025774.

[61] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998. doi:10.1007/BFb0028741.

[62] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643.

[63] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In B. Steffen and G. Levi, editors, *5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*, pages 85–96. Springer, 2004. doi:10.1007/978-3-540-24622-0_9.

[64] E. M. Clarke, W. Klieber, M. Novácek, and P. Zuliani. Model checking and the state explosion problem. In B. Meyer and M. Nordio, editors, *LASER International Summer School 2011*, volume 7682 of *LNCS*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.

[65] E. M. Clarke, T. A. Henzinger, and H. Veith. Introduction to model checking. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer, 2018. doi:10.1007/978-3-319-10575-8_1.

[66] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. A. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 394–397. Springer, 31. doi:10.1007/3-540-61474-5_87.

[67] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *European Conference on Artificial Life (ECAL)*, pages 134–142. Elsevier, 1991.

[68] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In P. V. Hentenryck, editor, *4th International Symposium on Static Analysis (SAS)*, volume 1302 of *LNCS*, pages 388–394. Springer, 1997. doi:10.1007/BFb0032759.

[69] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *2nd International Symposium on Programming*, pages 106–130. Dunod, 1976.

[70] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[71] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992. doi:10.1093/logcom/2.4.511.

[72] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957. doi:10.2307/2963594.

[73] O.-J. Dahl and K. Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966. doi:10.1145/365813.365819.

[74] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer, 2005.

[75] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In M. Falaschi and E. Albert, editors, *17th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 91–102. ACM, 2015. doi:10.1145/2790449.2790529.

[76] R. De Nicola. Behavioral equivalences. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 120–127. Springer, 2011. doi:10.1007/978-0-387-09766-4_517.

[77] R. De Nicola. Process algebras. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1624–1636. Springer, 2011. doi:10.1007/978-0-387-09766-4_450.

[78] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):7:1–7:29, 2014. doi:10.1145/2619998.

[79] R. De Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, and A. Vandin. The SCEL language: Design, implementation, verification. In M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, editors, *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*, volume 8998 of *LNCS*, pages 3–71. Springer, 2015. ISBN 978-3-319-16309-3 978-3-319-16310-9. doi:10.1007/978-3-319-16310-9_1.

[80] R. De Nicola, L. Di Stefano, and O. Inverso. Toward formal models and languages for verifiable multi-robot systems. *Frontiers in Robotics and AI*, 5:94, 2018. doi:10.3389/frobt.2018.00094.

[81] R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: Empowering Erlang with attribute-based communication. *Science of Computer Programming*, 168:71–93, 2018. doi:10.1016/j.scico.2018.08.006.

[82] R. De Nicola, T. Duong, and M. Loreti. ABEL - A domain specific framework for programming with attribute-based communication. In H. R. Nielson and E. Tuosto, editors, *21st International Conference on Coordination Models and Languages (COORDINATION)*, volume 11533 of *LNCS*, pages 111–128. Springer, 2019. doi:10.1007/978-3-030-22397-7_7.

[83] R. De Nicola, L. Di Stefano, and O. Inverso. Multi-agent systems with virtual stigmergy. *Science of Computer Programming*, 187:102345, 2020. doi:10.1016/j.scico.2019.102345.

[84] J. L. Deneubourg, S. Aron, S. Goss, J. M. Pasteels, and G. Duerinck. Random behaviour, amplification processes and number of participants: How they contribute to the foraging properties of ants. *Physica D*, 22(1):176–186, 1986. doi:10.1016/0167-2789(86)90239-3.

[85] L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In L. Padgham, D. C. Parkes, J. P. Müller, and S. Parsons, editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 3, pages 1303–1306. IFAAMAS, 2008.

[86] C. Detrain and J.-L. Deneubourg. Scavenging by Pheidole pallidula: A key for understanding decision-making systems in ants. *Animal Behaviour*, 53(3):537–547, 1997. doi:10.1006/anbe.1996.0305.

[87] L. Di Stefano, F. Lang, and W. Serwe. Combining SLiVER with CADP to analyze multi-agent systems. In S. Bliudze and L. Bocchi, editors, *22nd International Conference on Coordination Models and Languages (COORDINATION)*, volume 12134 of *LNCS*, pages 370–385. Springer, 2020. doi:10.1007/978-3-030-50029-0_23.

[88] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971. doi:10.1007/BF00289519.

[89] B. Dimsdale and H. M. Markowitz. A Description of the SIMSCRIPT Language. *IBM Systems Journal*, 3(1):57–67, 1964. doi:10.1147/sj.31.0057.

[90] G. Echeverria, S. Lemaignan, A. Degroote, S. Lacroix, M. Karg, P. Koch, C. Lesire, and S. Stinckwich. Simulating complex robotic scenarios with MORSE. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz,

C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, editors, *3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, volume 7628 of *LNCS*, pages 197–208. Springer, 2012. doi:10.1007/978-3-642-34327-8_20.

[91] A. M. El-Sayed, P. Scarborough, L. Seemann, and S. Galea. Social network analysis and agent-based modeling in social epidemiology. *Epidemiologic Perspectives & Innovations*, 9(1):1, 2012. doi:10.1186/1742-5573-9-1.

[92] J. M. Epstein and R. Axtell. *Growing artificial societies: Social science from the bottom up*. MIT Press, 1996. ISBN 978-0-262-05053-1.

[93] J. D. Farmer and D. Foley. The economy needs agent-based modelling. *Nature*, 460(7256): 685–686, 2009. doi:10.1038/460685a.

[94] R. D. Fennell and V. R. Lesser. Parallelism in artificial intelligence problem solving: A case study of Hearsay II. *IEEE Transactions on Computers*, 26(2):98–111, 1977. doi:10.1109/TC.1977.5009289.

[95] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971. doi:10.1016/0004-3702(71)90010-5.

[96] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

[97] M. Y. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. ESBMC 5.0: An industrial-strength C model checker. In M. Huchard, C. Kästner, and G. Fraser, editors, *33rd International Conference on Automated Software Engineering (ASE)*, pages 888–891. ACM, 2018. doi:10.1145/3238147.3240481.

[98] H. Garavel. Revisiting sequential composition in process calculi. *Journal of Logical and Algebraic Methods in Programming*, 84(6):742–762, 2015. doi:10.1016/j.jlamp.2015.08.001.

[99] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In P. A. Abdulla and K. R. M. Leino, editors, *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011. doi:10.1007/978-3-642-19835-9_33.

[100] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *Software Tools for Technology Transfer*, 15(2):89–107, 2013. doi:10.1007/s10009-012-0244-z.

[101] H. Garavel, R. Mateescu, and W. Serwe. Large-scale distributed verification using CADP: Beyond clusters to grids. *Electronic Notes in Theoretical Computer Science*, 296:145–161, 2013. doi:10.1016/j.entcs.2013.07.010.

[102] H. Garavel, F. Lang, and R. Mateescu. Compositional verification of asynchronous concurrent systems using CADP. *Acta Informatica*, 52(4-5):337–392, 2015. doi:10.1007/s00236-015-0226-1.

[103] H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In J.-P. Katoen, R. Langerak, and A. Rensink, editors, *ModelEd, TestEd, TrustEd*, volume 10500 of *LNCS*, pages 3–26. Springer, 2017. doi:10.1007/978-3-319-68270-9_1.

[104] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, 1982. doi:10.1109/TC.1982.1675885.

[105] R. Gerth, D. A. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *15th International Symposium on Protocol Specification, Testing and Verification (PSTV)*, pages 3–18. Chapman & Hall, 1995.

[106] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

[107] P. Godefroid. *Partial-order methods for the verification of concurrent systems - An approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer, 1996. ISBN 3-540-60761-7. doi:10.1007/3-540-60761-7.

[108] M. Goldsmith and I. Zakiuddin. Critical systems validation and verification with CSP and FDR. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *International Workshop on Current Trends in Applied Formal Method (FM-Trends)*, volume 1641 of *LNCS*, pages 243–250. Springer, 1998. doi:10.1007/3-540-48257-1_15.

[109] V. Goranko and A. Rumberg. Temporal logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2020.

[110] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997. doi:10.1007/3-540-63166-6_10.

[111] P.-P. Grassé. 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. *Insectes Sociaux*, 6(1):41–80, 1959. doi:10.1007/BF02223791.

[112] J. Gray. Notes on data base operating systems. In M. J. Flynn, J. Gray, A. K. Jones, K. Lagally, H. Opderbeck, G. J. Popek, B. Randell, J. H. Saltzer, and H.-R. Wiehle, editors, *Operating Systems, An Advanced Course*, volume 60 of *LNCS*, pages 393–481. Springer, 1978. doi:10.1007/3-540-08755-9_9.

[113] V. Grimm and S. F. Railsback. Agent-based models in ecology: Patterns and alternative theories of adaptive behaviour. In F. C. Billari, T. Fent, A. Prskawetz, and J. Scheffran, editors, *Agent-Based Computational Modelling: Applications in Demography, Social, Economic and Environmental Sciences*, Contributions to Economics, pages 139–152. Physica-Verlag, 2006. ISBN 978-3-7908-1721-8. doi:10.1007/3-7908-1721-X_7.

[114] V. Grimm, E. Revilla, U. Berger, F. Jeltsch, W. M. Mooij, S. F. Railsback, H.-H. Thulke, J. Weiner, T. Wiegand, and D. L. DeAngelis. Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. *Science*, 310(5750):987–991, 2005. doi:10.1126/science.1116681.

[115] J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems*. MIT Press, 2014. ISBN 978-0-262-02771-7.

[116] H. Günther, A. Laarman, and G. Weissenbacher. Vienna Verification Tool: IC3 for parallel software - (competition contribution). In M. Chechik and J.-F. Raskin, editors, *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *LNCS*, pages 954–957. Springer, 2016. doi:10.1007/978-3-662-49674-9_69.

[117] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In D. Kroening and C. S. Pasareanu, editors, *27th International Conference on Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015. doi:10.1007/978-3-319-21690-4_20.

[118] J. Harrison. Theorem proving for verification (invited tutorial). In A. Gupta and S. Malik, editors, *20th International Conference on Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 11–18. Springer, 2008. doi:10.1007/978-3-540-70545-1_4.

[119] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000. doi:10.1007/s100090050043.

[120] M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In D. Van Hung and M. Ogawa, editors, *11th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 8172 of *LNCS*, pages 365–380. Springer, 2013. doi:10.1007/978-3-319-02444-8_26.

[121] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In N. Sharygina and H. Veith, editors, *25th International Conference on Computer aided verification (CAV)*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013. doi:10.1007/978-3-642-39799-8_2.

[122] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. Ultimate Automizer and the search for perfect interpolants - (competition contribution). In D. Beyer and M. Huisman, editors, *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10806 of *LNCS*, pages 447–451. Springer, 2018. doi:10.1007/978-3-319-89963-3_30.

[123] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *7th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 85 of *LNCS*, pages 299–309. Springer, 1980. doi:10.1007/3-540-10003-2_79.

[124] J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018. doi:10.1016/j.jlamp.2018.02.004.

[125] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In J. Launchbury and J. C. Mitchell, editors, *29th Symposium on Principles of Programming Languages (POPL)*, pages 58–70. ACM, 2002. doi:10.1145/503272.503279.

[126] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245. Morgan Kaufmann, 1973.

[127] F. Heylighen. Stigmergy as a universal coordination mechanism I: Definition and components. *Cognitive Systems Research*, 38:4–13, 2016. doi:10.1016/j.cogsys.2015.12.002.

[128] D. Hiebeler. The Swarm Simulation System and individual-based modeling. In *Decision Support 2001: Advanced Technology for Natural Resource Management*, 1994.

[129] J. Hillston, J. Pitt, M. Wirsing, and F. Zambonelli. Collective adaptive systems: Qualitative and quantitative modelling and analysis (Dagstuhl seminar 14512). *Dagstuhl Reports*, 4 (12):68–113, 2014. doi:10.4230/DagRep.4.12.68.

[130] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999. doi:10.1023/A:1010084620690.

[131] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971. doi:10.1007/BFb0059696.

[132] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5.

[133] K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012. doi:10.1007/978-3-642-31612-8_13.

[134] J. H. Holland. Complex adaptive systems. *Daedalus*, 121(1):17–30, 1992.

[135] S. M. Imam and V. Sarkar. Savina - An actor benchmark suite: Enabling empirical evaluation of actor libraries. In E. G. Boix, P. Haller, A. Ricci, and C. Varela, editors, *4th International Workshop on Programming based on Actors Agents & Decentralized Control (AGERE)*, pages 67–80. ACM, 2014. doi:10.1145/2687357.2687368.

[136] M. E. Inchiosa and M. T. Parker. Overcoming design and development challenges in agent-based modeling using ASCAPE. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7304–7308, 2002. doi:10.1073/pnas.082081199.

[137] O. Inverso and C. Trubiani. Parallel and distributed bounded model checking of multi-threaded programs. In R. Gupta and X. Shen, editors, *25th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 202–216. ACM, 2020. doi:10.1145/3332466.3374529.

[138] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In M. B. Cohen, L. Grunske, and M. Whalen, editors, *30th International Conference on Automated Software Engineering (ASE)*, pages 807–812. IEEE, 2015. doi:10.1109/ASE.2015.108.

[139] F. Iozzi, F. Trusiano, M. Chinazzi, F. C. Billari, E. Zagheni, S. Merler, M. Ajelli, E. Del Fava, and P. Manfredi. Little Italy: An agent-based approach to the estimation of contact patterns- fitting predicted matrices to serological data. *PLoS Computational Biology*, 6(12), 2010. doi:10.1371/journal.pcbi.1001021.

[140] ISO/IEC. Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, ISO, 1989.

[141] ISO/IEC. Programming languages- C. International Standard 9899:1999(E), ISO, 1999.

[142] H. Kaul and Y. Ventikos. Investigating biocomplexity through the agent-based paradigm. *Briefings in Bioinformatics*, 16(1):137–152, 2015. doi:10.1093/bib/bbt077.

[143] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394, 1976. doi:10.1145/360248.360252.

[144] N. P. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2149–2154. IEEE, 2004. doi:10.1109/IROS.2004.1389727.

[145] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In A. Biere and R. Bloem, editors, *26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 17–34. Springer, 2014. doi:10.1007/978-3-319-08867-9_2.

[146] K. Konolige. A first order formalization of knowledge and action for a multi-agent planning system. *Machine Intelligence*, 10, 1982.

[147] K. Konolige and N. J. Nilsson. Multiple-agent planning systems. In R. Balzer, editor, *1st National Conference on Artificial Intelligence (AAAI)*, pages 138–142. AAAI Press/MIT Press, 1980.

[148] A. Koubâa, M.-F. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. M. Shakshuki. COROS: A multi-agent software architecture for cooperative and autonomous service robots. In A. Koubâa and J. R. M. de Dios, editors, *Cooperative Robots and Sensor Networks 2015*, volume 604 of *Studies in Computational Intelligence*, pages 3–30. Springer, 2015. doi:10.1007/978-3-319-18299-5_1.

[149] P. Kouvaros and A. Lomuscio. A counter abstraction technique for the verification of robot swarms. In B. Bonet and S. Koenig, editors, *29th Conference on Artificial Intelligence (AAAI)*, pages 2081–2088. AAAI, 2015.

[150] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27: 333–354, 1983. doi:10.1016/0304-3975(82)90125-6.

[151] J. Lächele, A. Franchi, H. H. Bülthoff, and P. Robuffo Giordano. SwarmSimX: Real-time simulation environment for multi-robot systems. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, editors, *3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, volume 7628 of *LNCS*, pages 375–387. Springer, 2012. doi:10.1007/978-3-642-34327-8_34.

[152] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987. doi:10.1016/0004-3702(87)90050-6.

[153] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

[154] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

[155] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978. doi:10.1016/0376-5075(78)90045-4.

[156] F. Lang, R. Mateescu, and F. Mazzanti. Compositional verification of concurrent systems by combining bisimulations. In M. H. ter Beek, A. McIver, and J. N. Oliveira, editors, *3rd World Congress on Formal Methods (FM)*, volume 11800 of *LNCS*, pages 196–213. Springer, 2019. doi:10.1007/978-3-030-30942-8_13.

[157] F. Lang, R. Mateescu, and F. Mazzanti. Sharp congruences adequate with temporal logics combining weak and strong modalities. In A. Biere and D. Parker, editors, *26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12079 of *LNCS*, pages 57–76, Dublin, Ireland, 2020. Springer. doi:10.1007/978-3-030-45237-7_4.

[158] C. G. Langton. Studying artificial life with cellular automata. *Physica D*, 22(1):120–149, 1986. doi:10.1016/0167-2789(86)90237-X.

[159] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

[160] T. Lev-Ami, R. Manevich, and S. Sagiv. TVLA: A system for generating abstract interpreters. In R. Jacquart, editor, *18th World Computer Congress, topical sessions (WCC)*, volume 156 of *IFIP-AICT*, pages 367–375. Kluwer/Springer, 2004. doi:10.1007/978-1-4020-8157-6_28.

[161] M. Li, Z. Cai, X. Yi, Z. Wang, Y. Wang, Y. Zhang, and X. Yang. ALLIANCE-ROS: A software architecture on ROS for fault-tolerant cooperative multi-robot systems. In R. Booth and M.-L. Zhang, editors, *14th Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, volume 9810 of *LNCS*, pages 233–242. Springer, 2016. doi:10.1007/978-3-319-42911-3_19.

[162] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: An open-source model checker for the verification of multi-agent systems. *Software Tools for Technology Transfer*, 19(1):9–30, 2017. doi:10.1007/s10009-015-0378-x.

[163] M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with CARMA and its tools. In M. Bernardo, R. De Nicola, and J. Hillston, editors, *16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), Advanced Lectures*, volume 9700 of *LNCS*, pages 83–119. Springer, 2016. doi:10.1007/978-3-319-34096-8_4.

[164] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. C. Balan. MASON: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005. doi:10.1177/0037549705058073.

[165] J. Malhotra, R. M. Shapiro, S. A. Smolka, and A. Giacalone. Winston: A tool for hierarchical design and simulation of concurrent systems. In C. Rattray, editor, *Workshop on Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 140–152. Springer, 1988. doi:10.1007/978-1-4471-3534-0_7.

[166] M. Marescotti, A. Gurfinkel, A. E. J. Hyvärinen, and N. Sharygina. Designing parallel PDR. In D. Stewart and G. Weissenbacher, editors, *17th international Conference on Formal Methods in Computer Aided Design (FMCAD)*, pages 156–163. IEEE, 2017. doi:10.23919/FMCAD.2017.8102254.

[167] M. Massink, D. Latella, A. Bracciali, and J. Hillston. Modelling non-linear crowd dynamics in Bio-PEPA. In D. Giannakopoulou and F. Orejas, editors, *14th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 6603 of *LNCS*, pages 96–110. Springer, 2011. doi:10.1007/978-3-642-19811-3_8.

[168] M. Massink, M. Brambilla, D. Latella, M. Dorigo, and M. Birattari. On the use of Bio-PEPA for modelling and analysing collective behaviours in swarm robotics. *Swarm Intelligence*, 7(2-3):201–228, 2013. doi:10.1007/s11721-013-0079-6.

[169] M. J. Matarić. Designing emergent behaviors: From local interactions to collective intelligence. In J.-A. Meyer, H. L. Roitblat, and S. W. Wilson, editors, *2nd International Conference on Simulation of Adaptive Behavior (SAB)*, pages 432–441. MIT Press, 1993.

[170] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *15th International Symposium on Formal Methods (FM)*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008. doi:10.1007/978-3-540-68237-0_12.

[171] T. Mazur and G. Lowe. CSP-based counter abstraction for systems with node identifiers. *Science of Computer Programming*, 81:3–52, 2014. doi:10.1016/j.scico.2013.03.018.

[172] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4): 308–320, 1976. doi:10.1109/TSE.1976.233837.

[173] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989. doi:10.1145/76380.76382.

[174] C. McCaig, R. Norman, and C. Shankland. Process algebra models of population dynamics. In K. Horimoto, G. Regensburger, M. Rosenkranz, and H. Yoshida, editors, *3rd International Conference on Algebraic Biology (AB)*, volume 5147 of *LNCS*, pages 139–155. Springer, 2008. doi:10.1007/978-3-540-85101-1_11.

[175] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[176] B. McDaniel. Issues in distributed artificial intelligence. In *1st International Conference on Data Engineering (ICDE)*, pages 293–297. IEEE, 1984. doi:10.1109/ICDE.1984.7271285.

[177] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002. doi:10.1007/3-540-45657-0_19.

[178] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006. doi:10.1007/11817963_14.

[179] J.-J. C. Meyer, J. Broersen, and A. Herzig. BDI logics. In H. van Ditmarsch, J. Y. Halpern, W. van der Hoek, and B. Kooi, editors, *Handbook of Epistemic Logic*. College Publications, 2015. ISBN 978-1-84890-158-2.

[180] R. Milner. *A calculus of communicating systems*, volume 92 of *LNCS*. Springer, 1980. ISBN 3-540-10235-3. doi:10.1007/3-540-10235-3.

[181] R. Milner, M. Tofte, and R. Harper. *Definition of standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

[182] M. Mirolli, L. Tummolini, and C. Castelfranchi. Stigmergic cues and their uses in coordination: An evolutionary approach. In A. M. Uhrmacher and D. Weyns, editors, *Multi-Agent Systems - Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 243–265. CRC Press / Taylor & Francis, 2009. doi:10.1201/9781420070248.ch8.

[183] M. Mobilia, A. Petersen, and S. Redner. On the role of zealotry in the voter model. *Journal of Statistical Mechanics: Theory and Experiment*, 2007(08):P08029, 2007. doi:10.1088/1742-5468/2007/08/P08029.

[184] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004. doi:10.1016/j.jlap.2004.03.008.

[185] D. E. Nadales Agut. *A compositional interchange format for hybrid systems: Design and implementation*. PhD Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2012.

[186] A. Newell and H. A. Simon. GPS, a program that simulates human thought. In H. Billing, editor, *Lernende Automaten*, pages 109–124. Oldenbourg, 1961. doi:10.1016/B978-1-4832-1446-7.50040-6.

[187] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In C. Artho, A. Legay, and D. Peled, editors, *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 9938 of *LNCS*, pages 174–191, 2016. doi:10.1007/978-3-319-46520-3_12.

[188] N. J. Nilsson. Logic and artificial intelligence. *Artificial Intelligence*, 47(1-3):31–56, 1991. doi:10.1016/0004-3702(91)90049-P.

[189] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *15th International Workshop on Computer Science Logic (CSL)*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

[190] R. Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on Automatic Control*, 51(3):401–420, 2006. doi:10.1109/TAC.2005.864190.

[191] D. Olner. *An agent-based modelling approach to spatial economic theory*. PhD Thesis, University of Leeds, Leeds, UK, 2014.

[192] D. Olner, A. J. Evans, and A. J. Heppenstall. An agent model of urban economics: Digging into emergence. *Computers, Environment and Urban Systems*, 54:414–427, 2015. doi:10.1016/j.compenvurbsys.2014.12.003.

[193] R. Ostreiher. Is mobbing altruistic or selfish behaviour? *Animal Behaviour*, 66(1):145–149, 2003. doi:10.1006/anbe.2003.2165.

[194] H. V. D. Parunak. "Go to the ant": Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75(0):69–101, 1997. doi:10.1023/A:1018980001403.

[195] A. Philippou, M. Toro, and M. Antonaki. Simulation and verification in a process calculus for spatially-explicit ecological models. *Scientific Annals of Computer Science*, 23(1): 119–167, 2013. doi:10.7561/SACS.2013.1.119.

[196] C. Pinciroli and G. Beltrame. Buzz: An extensible programming language for heterogeneous swarm robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3794–3800. IEEE, 2016. ISBN 978-1-5090-3762-9. doi:10.1109/IROS.2016.7759558.

[197] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo. ARGoS: A modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012. doi:10.1007/S11721-012-0072-5.

[198] C. Pinciroli, A. Lee-Brown, and G. Beltrame. A tuple space for data sharing in robot swarms. In J. Suzuki, T. Nakano, and H. Hess, editors, *9th International Conference on Bio-Inspired Information and Communications Technologies (BICT)*, pages 287–294. ICST/ACM, 2015.

[199] L. Pitonakova, R. Crowder, and S. Bullock. Information flow principles for plasticity in foraging robot swarms. *Swarm Intelligence*, 10(1):33–63, 2016. doi:10.1007/s11721-016-0118-1.

[200] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[201] A. Pnueli. The temporal logic of programs. In *18th Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE, 1977. doi:10.1109/SFCS.1977.32.

[202] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 107–122. Springer, 2002. doi:10.1007/3-540-45657-0_9.

[203] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In W. Pugh and C. Chambers, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 14–24. ACM, 2004. doi:10.1145/996841.996845.

[204] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng. ROS: An open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[205] Z. Rakamaric and M. Emmi. SMACK: Decoupling Source Language Details from Verifier Implementations. In A. Biere and R. Bloem, editors, *26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 106–113. Springer, 2014. doi:10.1007/978-3-319-08867-9_7.

[206] A. S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. V. de Velde and J. W. Perram, editors, *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996. doi:10.1007/BFb0031845.

[207] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *1st International Conference on Multiagent Systems (ICMAS)*, pages 312–319. MIT Press, 1995.

[208] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In M. C. Stone, editor, *14th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 25–34. ACM, 1987. doi:10.1145/37402.37406.

[209] A. Riesco. Model checking parameterized by the semantics in Maude. In J. P. Gallagher and M. Sulzmann, editors, *14th International Symposium on Functional and Logic Programming (FLOPS)*, volume 10818 of *LNCS*, pages 198–213. Springer, 2018. doi:10.1007/978-3-319-90686-7_13.

[210] E. Rohmer, S. P. N. Singh, and M. Freese. V-REP: A versatile and scalable robot simulation framework. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1321–1326. IEEE, 2013. doi:10.1109/IROS.2013.6696520.

[211] R. J. Ross, R. W. Collier, and G. M. P. O'Hare. AF-APL - Bridging principles and practice in agent oriented languages. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *2nd International Workshop on Programming Multi-Agent Systems (ProMAS)*, volume 3346 of *LNCS*, pages 66–88. Springer, 2004. doi:10.1007/978-3-540-32260-3_4.

[212] S. E. Russell, H. R. Jordan, G. M. P. O'Hare, and R. W. Collier. Agent Factory: A framework for prototyping logic-based AOP languages. In F. Klügl and S. Ossowski, editors, *9th German Conference on Multiagent System Technologies (MATES)*, volume 6973 of *LNCS*, pages 125–136. Springer, 2011. doi:10.1007/978-3-642-24603-6_13.

[213] S. J. Russell and P. Norvig. *Artificial intelligence - A modern approach*. Pearson Education, 3rd edition, 2010. ISBN 978-0-13-207148-2.

[214] E. Sahin. Swarm robotics: From sources of inspiration to domains of application. In E. Sahin and W. M. Spears, editors, *1st International Workshop on Swarm Robotics*, volume 3342 of *LNCS*, pages 10–20. Springer, 2004. doi:10.1007/978-3-540-30552-1_2.

[215] D. Sangiorgi and D. Walker. *The pi-calculus - A theory of mobile processes*. Cambridge University Press, 2001. ISBN 978-0-521-78177-0.

[216] A. Scheidler, A. Brutschy, E. Ferrante, and M. Dorigo. The k-unanimity rule for self-organized decision-making in swarms of robots. *IEEE Transactions on Cybernetics*, 46 (5):1175–1188, 2016. doi:10.1109/TCYB.2015.2429118.

[217] M. Schlosser. Agency. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2019.

[218] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In R. Alur and D. A. Peled, editors, *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004. doi:10.1007/978-3-540-27813-9_16.

[219] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. H. Jr. and S. D. Johnson, editors, *3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000. doi:10.1007/3-540-40922-X_8.

[220] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993. doi:10.1016/0004-3702(93)90034-9.

[221] M. Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4(3): 237–257, 1998. doi:10.1162/106454698568576.

[222] E. Sklar. NetLogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311, 2007. doi:10.1162/artl.2007.13.3.303.

[223] H. Skubch, M. Wagner, R. Reichle, and K. Geihs. A modelling language for cooperative plans in highly dynamic domains. *Mechatronics*, 21(2):423–433, 2011. doi:10.1016/j.mechatronics.2010.10.006.

[224] A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu. Semantics-based program verifiers for all languages. In E. Visser and Y. Smaragdakis, editors, *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 74–91. ACM, 2016. doi:10.1145/2983990.2984027.

[225] J. E. Stiglitz and M. Gallegati. Heterogeneous interacting agent models for understanding monetary economies. *Eastern Economic Journal*, 37(1):6–12, 2011. doi:10.1057/eej.2010.33.

[226] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33–65, 2017. doi:10.1007/s10817-016-9389-x.

[227] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

[228] D. Syme. The early history of F#. *Proceedings of the ACM on Programming Languages*, 4(HOPL):75:1–75:58, 2020. doi:10.1145/3386325.

[229] J. Sztipanovits. Composition of cyber-physical systems. In *14th International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 3–6. IEEE, 2007. ISBN 0769527728. doi:10.1109/ECBS.2007.25.

[230] A. Tarski. A decision method for elementary algebra and geometry. Technical Report R-109, RAND Corporation, 1951. Reprinted in [53].

[231] L. Tesfatsion. Agent-based computational economics: Growing economies from the bottom up. *Artificial Life*, 8(1):55–82, 2002. doi:10.1162/106454602753694765.

[232] L. Tesfatsion and K. Judd. *Handbook of computational economics*, volume 2 of *Handbooks in Economics*. Elsevier, 2006. ISBN 978-0-444-51253-6.

[233] G. Theraulaz and E. Bonabeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, 1999. doi:10.1162/106454699568700.

[234] C. M. N. Tofts. A synchronous calculus of relative frequency. In J. C. M. Baeten and J. W. Klop, editors, *International Conference on Concurrency Theory (CONCUR)*, volume 458 of *LNCS*, pages 467–480. Springer, 1990. doi:10.1007/BFb0039078.

[235] C. M. N. Tofts. Process algebra as modelling. *Electronic Notes in Theoretical Computer Science*, 162:323–326, 2006. doi:10.1016/j.entcs.2005.12.114.

[236] J. Toner and Y. Tu. Flocks, herds, and schools: A quantitative theory of flocking. *Physical Review E*, 58(4):4828–4858, 1998. doi:10.1103/PhysRevE.58.4828.

[237] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000. doi:10.1145/352029.352035.

[238] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Symposium on Logic in Computer Science (LICS)*, pages 332–344. IEEE, 1986.

[239] R. Vaughan. Massively multi-robot simulation in Stage. *Swarm Intelligence*, 2(2-4): 189–208, 2008. doi:10.1007/s11721-008-0014-4.

[240] G. Vidal. Symbolic execution as a basis for termination analysis. *Science of Computer Programming*, 102:142–157, 2015. doi:10.1016/j.scico.2015.01.007.

[241] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984. doi:10.1109/TSE.1984.5010248.

[242] D. Weyns, M. Schumacher, A. Ricci, M. Viroli, and T. Holvoet. Environments in multiagent systems. *Knowledge Engineering Review*, 20(2):127–141, 2005. doi:10.1017/S0269888905000457.

[243] U. Wilensky. Modeling nature's emergent patterns with multi-agent languages. In *Euro-Logo*, 2001.

[244] G. Wilson and S. Shpall. Action. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2016.

[245] M. Winikoff. Assurance of agent systems: What role should formal verification play? In *Specification and Verification of Multi-Agent Systems*. Springer, 2010. ISBN 978-1-4419-6983-5 978-1-4419-6984-2. doi:10.1007/978-1-4419-6984-2_12.

[246] G. Winskel. Event structure semantics for CCS and related languages. In M. Nielsen and E. M. Schmidt, editors, *9th Colloquium on Automata, Languages and Programming (ICALP)*, volume 140 of *LNCS*, pages 561–576. Springer, 1982. doi:10.1007/BFb0012800.

[247] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995. doi:10.1017/S0269888900008122.

[248] M. J. Wooldridge. *The logical modelling of computational multi-agent systems*. PhD thesis, University of Manchester, 1992.