DOCTORAL THESIS

Programming and verifying systems relying on attribute-based communication

PHD PROGRAM IN COMPUTER SCIENCE: XXX CYCLE

Supervisor:

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

Author:

Tan DUONG
tan.duong@gssi.it

Internal advisors:

Dr. Catia TRUBIANI
catia.trubiani@gssi.it

Dr. Omar INVERSO
omar.inverso@gssi.it

July 2019

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

Abstract

Collective adaptive systems (CAS) are large conglomerates of interacting components which may be not entirely aware of themselves as members of a group and have limited mutual knowledge and rely only on local rules. However, sophisticated and hardly predictable behaviours often *emerges* from components interaction. The distinguishing features of CAS, namely adaptivity, anonymity, scalability, open-endedness make them attractive as models of modern systems but challenges practitioners to properly engineer and exploit their power. Attribute based communication is a recently proposed communication paradigm that seems to be particularly appropriate for CAS programming. This novel communication mechanism relies on send and receive primitives that allow selective multicast communication by means of predicates on the attributes exposed in the interface of components among interacting components. Indeed, attribute based communication offers high-level linguistic primitives to deal with CAS, but its practical impact to the actual CAS programming has not been fully investigated yet due to the lacking of efficient run-time environment and programming techniques, and this has prevented practical use.

In this thesis, we study the impact and the performances of attribute based communication from the programming and verification point of views. We take as starting point the recently proposed *AbC* process calculus and proceed by embedding its primitives in *Erlang*. We follow two different approaches. First, we implement in *Erlang* the process calculus *AbC* to come up with *ABEL*, a domain specific framework with an API very close to that of *AbC* and with a semantics fully faithful to that of *AbC*. Then, we introduce *AErlang* a direct extension of *Erlang* with attribute-based interaction primitives. We show that, by relying on a verification backend whose input is used as target of a structural translation of *AbC* systems, properties of *ABEL* programs can be verified by model checking. We instead evaluate the efficacy of embedding attribute based communication in the actor-based programming model by an extensive evaluation of *AErlang* on a number of case studies. The evaluation confirms that the overhead resulting from the new communication primitives is acceptable, and that *AErlang* prototype successfully preserves *Erlang* efficiency and scalability.

Acknowledgements

I would like to express my deep gratitude and appreciation to my supervisor Prof. Rocco De Nicola for his kindness, encouragement and guidance during my long journey of undertaking the PhD. His advises have been invaluable to all stages of my work, among others, which have always guided me to the right direction when I felt lost. I have growth both intellectually and personally along the way under his insightful guidance. It is my great pleasure to be his student and I am indebted to everything he has done for me. Thank you so much, Rocco!.

I am very graceful to my internal advisors Dr. Catia Trubiani and Dr. Omar Inverso for their helps and detailed suggestions for improving the quality of the work over the years. In particular, I appreciate their comments, numerous discussions and supports despite of their busy calendars and not complaining about my sloppy working behaviour.

Special thanks go to Dr. Franco Mazzanti for his enthusiastic collaboration on modeling attribute-based communication and verifying case studies. Discussions with Franco provided insights of modeling system semantics and understanding formal methods at work. In addition, discussions with Yehia A. Alramand have interested me in providing a semantics-preserving environment for attribute-based communication.

I would like to thank Prof. Gul Agha for providing me a short visit at his laboratory, UIUC. I heartfelt thank him for his time spent on weekly discussions with me, which have provided a better perspective of the research ideas in the actor model and real world distributed systems. I am also thankful to Dr. Karl Palmskog for his advice and mentoring during the internship.

I highly appreciated the reviewers Prof. Einar Broch Johnsen and Prof. Michele Loreti for their valuable comments and suggestions on an early version of this thesis. The comments undoubtedly helped me in improving the presentation and the quality of the thesis.

I thank GSSI, L'Aquila and IMT Lucca for their financial supports and providing great working environments to which I am able to concentrate on my work. I thank all my colleagues at GSSI, past and present, for their friendships. I would like to thank also my former supervisors and coworkers at HUST and at NII for initially inspiring me to pursuit a PhD, most of all, Dr. Nguyen Huu Duc and Prof. Zhenjiang Hu.

Finally, I wish to thank my parents, my brother and other family members for their emotional supports and understandings.

Declarations

This thesis has been composed by myself and the presented work is my own under the guidance of my supervisors Prof. Rocco De Nicola, Dr. Catia Trubiani and Dr. Omar Inverso. Moreover, Chapter 5 is based on R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang at Work. In *Proc. of the International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 485–497, 2017 and R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: empowering Erlang with attribute-based communication. *Science of Computer Programming*, 2018. Chapter 4 is a re-elaboration of R. De Nicola, T. Duong, O. Inverso, and F. Mazzanti. Verifying properties of systems relying on attribute-based communication. In *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma*, pages 169–190, 2017, co-authored with Prof. Rocco De Nicola, Dr. Omar Inverso and Dr. Franco Mazzanti.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**
1.1 Motivation 1
1.2 Approach 3
1.3 Contributions and Organization 4

2 Background **6**
2.1 *AbC*- A calculus for attribute-based communication 7
2.1.1 AbC syntax 7
2.1.2 *AbC* semantics 11
2.2 Case studies: *AbC* at work 14
2.2.1 Stable marriage with attributes 15
2.2.2 Distributed graph coloring 21
2.3 The Erlang programming language 25
2.3.1 Process 26
2.3.2 OTP behaviours 28
2.4 The UMC model checker 31

3 ABEL - an Erlang implementation of AbC **36**
3.1 ABEL's support for AbC 37
3.2 ABEL Implementation 42
3.2.1 Coordinating components 43
3.2.2 Coordinating processes 47
3.3 *ABEL*'s Case Studies 52
3.3.1 Attribute Stable Marriage 52
3.3.2 Graph Coloring 59
3.4 ABEL performance on case studies 62
3.4.1 Stable Marriage with Attribute 62
3.4.2 Graph Coloring 63
3.5 Concluding remarks 66

4 Systematic verification of qualitative properties **68**
4.1 Compiling from AbC into UMC 70
4.1.1 Specifying AbC systems 70

4.1.2 The translation 73
4.1.3 Implementation and Optimizations 84
4.2 Case studies 84
4.2.1 Stable Marriage 85
4.2.2 Graph coloring 92
4.3 Formal analysis and verification 95
4.3.1 Stable marriage 95
4.3.2 Graph Coloring 99
4.4 Concluding remarks 101

5 AErlang - Extending Erlang with attribute-based communication 103
5.1 AErlang support for attribute-based communication 105
5.2 Prototype Architecture 107
5.3 Forwarding Strategies 108
5.4 Implementation 111
5.5 Case Studies 115
5.5.1 Stable Marriage with preference list 116
5.5.2 Stable Marriage with attributes 118
5.5.3 An interactive market 120
5.5.4 Graph Colouring 123
5.6 AErlang Performance Evaluation 124
5.6.1 Efficiency 125
5.6.2 Scalability 126
5.6.3 Broadcasting vs. Pushing 129
5.6.4 Broadcasting vs. Push-Pulling 131
5.6.5 AErlang broadcast vs. *ABEL* broadcast 132
5.7 Concluding Remarks 134

6 Related Works 136
6.1 Communication and coordination models. 136
6.2 Implementations of AbC 137
6.3 Verification 139

7 Conclusions and Future Works 140

A Erlang programming 144
A.1 Erlang program for SMTI problem 144
A.2 Erlang program for SMI problem 144

B An *AbC* specification 146
B.1 Bottom-up Stable marriage 146

C A generated UMC model 149
C.1 UMC Model for Graph Coloring 149

Bibliography 155

Chapter 1

Introduction

1.1 Motivation

Collective adaptive systems (CAS) are a large conglomerates of components which are not entirely aware of themselves as members of a group and interact according to limited mutual knowledge and local rules, indirectly triggering global system evolution. The term was first coined in [59] after the observations of various CAS in modern society, such as stock markets, robot swarms, social networks, but also in the animal world, e.g. ant colonies or flocks of birds. These systems are usually distributed, interdependent and heterogeneous and are operating in open, nondeterministic environments. CAS components have their own attributes, capabilities and objectives, yet they continuously interact with each other and combine their behaviour to form collectives in order to achieve the system-level goals. They may also adapt their behaviour in response to changes of the environmental conditions or of the other components. Eventually, despite the simplicity of the components in isolation, the *emerging* global behaviour of the system may end up being quite sophisticated and hardly predictable. The distinguishing features of CAS can be summarized as:

- *anonymity*: the identity of components may be not known,
- *open-endedness*: new components may enter or leave at any time,
- *adaptivity*: rôles and interests of components may change,
- *scalability*: the number of components might grow very fast.

The central notion in CAS are collectives which are formed and dissolved dynamically without the intervention of external entities. Similarly, adaptation and decision making can hardly externally controlled [27]. These features pose big challenges for software developers and language designers to find appropriate tools and techniques towards understanding, reasoning and engineering CAS.

The work in this thesis is inspired by the research lines centered around the *attribute-based communication*, which was originally proposed as a feature of the SCEL language [49, 51] to model the dynamic formation of component ensembles in autonomic computing. In an attribute-based communication system, components are enriched with a set of attributes that encode relevant features such as role, status, position, etc. whose values can be modified by means of internal actions. Communication links among components are dynamically established by the interdependency relations defined through predicates over their attributes. Groups of interacting partners are dynamically formed and run-time changes of attributes enable opportunistic interactions among components. The attribute-based system is also parameterised with respect to the environment where system components are executed. The environment has a great impact on how components behave and provides a new mean of indirect communication, that allows components to mutually influence each other, possibly unintentionally.

This notion of attribute-based communication is at the basis of a kernel calculus, named *AbC* [25] that comprises a set of attribute-based related primitives. It was originally designed as a trimmed version of SCEL with the aim of assessing the impact of the novel communication paradigm and comparing it with the more classical mechanisms that handle interaction between distributed components by relying on identities (Actors [14]), on channels (π-calculus [101]), or on broadcast (Bπ-calculus [107]). Later refinements of AbC [21, 23] added programming idioms for the ease of modeling CAS. Informally, a command $send(v)@\pi$ expresses the intention of sending v to all entities satisfying predicate π while $receive(x)@\pi'$ indicates willingness to receive messages from entities satisfying predicate π' while binding the received values to x. Sending a message is non blocking while message reception is blocking in the sense that it may take place only by synchronizing with available sent message. In addition, components can update their attributes via assignments, $[a := v]$. This results in implicit changes of communication groups, which enables adaptive behaviour of components and the system. Thus, components, by changing their attributes or the communication predicates, can join and leave a collective at anytime without disrupting systems behaviour.

At a high level view, attribute-based communication can naturally capture the main distinguishing features of collective adaptive systems at no extra effort. For example, consider a social network scenario where users aim at forming groups for language exchange. Such groups may be formed by only considering the language that users wish to learn and the one in which potential partners are interested. Now, in case of multiple alternatives, it might be desirable to prefer people with similar age and interests, or even knowledgeable of a second language in common. Relevant attributes would then be: spoken languages, age, language of interest. Predicates may be built by specifying conditions on the attributes, e.g., $age \leq 25 \land \ language = English$. Here, the identity of users is irrelevant for forming the groups, and no change in the predicates is required when users join or leave the system, thus anonymity and open-endedness are no longer a concern. Under other classical paradigms such a task would require major and time-consuming operations. This is because other communication paradigms often rely on identities or channels which are independent from runtime status and capabilities of components.

While *AbC* offers high level linguistic primitives to model CAS, its efficiency for actual CAS programming is in question. Letting interacting entities communicate by relying on their dynamic attributes can be helpful in many circumstances, however it requires an efficient middleware that manages predicate evaluation and message exchange among components. Researchers have already developped implementations for *AbC*, for example, AbaCus [22] in Java and GoAt [19] in Google Go. These implementations employ a message broker to relay messages in a broadcast manner. It is the receiving component which decides whether to use or discard a forwarded message, by checking the sending and the receiving predicates. Both AbaCus and GoAt claimed to preserve *AbC* semantics. To increase the scalability, the latter used a set of brokers collaborating on mediating messages. Its performance evaluation reported in [19, 20] were conducted though simulation. Thus, the actual performance of *AbC* implementations have not yet been considered. Along this direction, we think that there are still rooms for improvements.

Equally important to obtaining efficient implementations for *AbC* is determining whether the outcome of the complex interactions in *AbC* systems is the expected one. Indeed, the dynamicity and flexibility of the paradigm may become a source of complexity. In general, the presence of non-linearity and non-determinism complicates significantly the assessment of specific properties or the forecasting emerging behaviour of CAS. Components often get involved into complex and diverse interactions and the effect of environment changes in some components would introduce opportunistic behaviour and indirectly affect to others. Also the quest for an appropriate trade off between AbC's expressiveness and efficiency needs consideration.

This thesis makes an effort towards understanding and exploiting systems featuring attribute-based communication. To this end, we take the attribute-based communication and its formalization - the *AbC* process calculus - as our starting points and investigate the impact of the new paradigm in programming and verifying distributed and collective adaptive systems.

1.2 Approach

From the programming perspective, it is desirable to support attribute-related primitives in the *AbC* style with a programming framework or a language with programming constructs closely corresponding to *AbC* ones. The design and implementation of such a framework should relieve programmers from the burden of working out details such as the explicit handling of attributes, the evaluation of predicates, while guaranteeing efficient message exchange among components. Our goal is leveraging the benefits of attribute-based communication at a programming level while addressing the performance concerns. We follow a pragmatic approach in which we embed *AbC* programming abstractions on top of an industrial strength, concurrency oriented programming language - *Erlang*- to utilize the advanced features of the language. *Erlang* [29] is well known for its light weight concurrency model and horizontal scalability on parallel machine architecture.

Our first attempt to attribute-based communication programming is providing a faithful implementation of *AbC* in *Erlang*, which we call *ABEL*. *ABEL* respects the semantics of *AbC* parallel operator both at the component and system levels, while its API has a one-to-one correspondence with *AbC* primitives. As a result, *ABEL* programs can be derived straightforwardly from *AbC* specifications and their properties can be verified indirectly via the verification of their original *AbC* specifications. *ABEL* can also be thought as an execution environment for *AbC* where a modeler can execute *AbC*-like programs and measure their performance.

Our second attempt takes an open view on assessing attribute-based programming. We do not tie ourself to *AbC* but rather consider only the key ingredients behind the paradigm. To this end, we instantiate programming abstractions in the *AbC*'s style on *Erlang* processes and study their impact. Our prototype, namely *AErlang* enables attribute-based communication in *Erlang*, in which standard communication primitives are replaced by attribute-based versions. Thus, an *AErlang* program mixes the actor-based model of the host language with attribute-based primitives. Different from the first approach, *AErlang* focuses on different ways of mediating messages and does not impose any order on message delivery. In addition, message passing among processes is inherently asynchronous, thus potentially increasing performances.

We provide implementations for both approaches and study their expressiveness and performances by using a number of case studies.

Regarding the verification aspects of non trivial system properties, our approach relies on an explicit state model checker. There is no dedicated model checker for *AbC* yet. When a new formalism is proposed, the standard initial approach of the formal methods community is to provide systematic translation from the given formalism to another verifiable one to exploit an external model checker. The main challenge for this approach is the definition of an appropriate mapping from one formalism to the other. For this part of the work, we rely on UMC, a well-studied verification framework developed along the years in the context of several European projects. UMC formal semantics model is based on doubly labeled transition system (L2TS) [53] which well capture the key aspects of *AbC* systems by allowing to label with predicates both systems transitions and systems states. Abstract system properties are specified in ACTL [52], a branching time logics similar to CTL [52]. We define a structural translation to derive formally verifiable models from *AbC* specifications and show how interesting system properties can be verified in UMC.

1.3 Contributions and Organization

The main contribution of the thesis are the following:

A faithful implementation of *AbC*. We introduce *ABEL*, an implementation of *AbC* in *Erlang* and assess its expressiveness and impact from a programming point of view by considering a number non

trivial case studies relying on attribute based communication. We also discuss how *ABEL* programs can be analyzed by verifying *AbC* specifications.

The design and implementation of *AErlang*. We extend *Erlang* processes with attribute-based related primitives of the novel communication paradigm. We perform an extensive evaluation to demonstrate the usefulness of our prototype in terms of expressiveness and runtime performance by means of non trivial case studies.

A methodology for verifying attribute-based systems. We propose a structural translation of *AbC* terms into another verifiable formalism, subjected to formal analysis and verification with an external model checker. We demonstrate the feasibility of the approach by verifying a number of the properties of the considered case studies.

The rest of the thesis is structured as follows. Chapter 2 provides background knowledges related to the research pursued in this thesis. There we also introduce the specifications of some case studies using *AbC*. In Chapter 3, we present *ABEL* - a faithful implementation of *AbC* semantics and stress how easy it is to write *ABEL* programs starting from a given *AbC* specifications. The design and implementation of *ABEL* API and the strategies used for coordinating processes and components are also presented. Chapter 4 is dedicated to the problem of verifying *AbC* specifications. We describe a structural translation of *AbC* into another verifiable formalism which is the input accepted by the UMC verification framework. We then show that non trivial system properties, such as emerging system behaviour, together with other functional, qualitative properties of the designed communication protocols, can be specified and verified using our approach. In Chapter 5, we consider another approach to *AbC* programming, in which *Erlang* processes are combined with the notion of attribute-based communication. *AErlang* APIs and different strategies of message forwarding are presented. We show how to program in *AErlang* by using a number of case studies, and assesses the performance of our *AErlang* prototype via extensive experiments. Chapter 6 discusses related works. Chapter 7 contains some concluding remarks and hints for future works.

Chapter 2

Background

This chapter presents some background material which is useful for a full appreciation of the implementations of attribute based communication (*AbC*) and of the case studies that will be later considered. In particular, we first review the syntax and semantics of the *AbC* calculus with the help of an example based on the classical version of the stable marriage problem (SMP) that is simple enough to support reader's intuition, yet makes use of all *AbC* constructs. Then, we introduce two more complex case studies which we use throughout the thesis, namely an attribute variant of the stable marriage problem and a distributed graph coloring scenario. At a superficial level, all the considered case studies are modeled as a set of interacting components. However, as their complexity grows, careful design and coordination of individual components is required for achieving meaningful interactions.

In the rest of the chapter, we review the relevant features of *Erlang* which is the host language of our implementations of *AbC*. We also provide a brief overview of the UMC model checker, its modeling language and the verification logic. More details on how UMC servers as the verification backend for the formal analysis of systems specified in *AbC* are presented in Chapter 4. There, the correctness of case studies is guaranteed by formal verification. Please, notice that below and in the rest of the thesis we do use the words agent and component interchangeably.

Example 1 (Stable Marriage Problem). The SMP problem [65] consists of deriving a one-to-one assignment (a matching) M between elements of two equally-sized sets, which are referred as Men and Women. Each element corresponds to one single person with a unique identity and a preference list ranking all members of the opposite set that the person wants to match. A complete matching S is a set of pairs $\{(m_{i_1}, w_{j_1}), (m_{i_2}, w_{j_2}), \dots, (m_{i_n}, w_{j_n})\}$ where $i_1, \dots i_n$ and $j_1, \dots j_n$ are some permutations of $1, \dots, n$ with n the number of men and women. A complete matching S is stable if for any pair (m, w) in S, there is no any other pair $(m', w') \neq (m, w)$ such that m prefers w' to w and m' prefers w to w'. Here, a man m prefers a woman w to another w' if w appears before w' in the preference list of m. A similar explanation applies for women.

2.1 *AbC*- A calculus for attribute-based communication

The process calculus AbC [21] was developed after the SCEL formal language [51] to study the merit of the new communication paradigm SCEL introduced. To this end, AbC relies on a set of minimal constructs that facilitates attribute-based communication where the interaction is based on the satisfaction of predicates over run-time attributes of communicating components. It is worth noticing that AbC has undergone several modifications with slightly different syntax and semantics, in what follows we review the latest version of AbC presented in [23].

2.1.1 AbC syntax

The syntax of AbC is given in Figure 2.1. A component can be a process P associated with an attribute environment Γ and an interface I, or the parallel composition of two components $C_1 \parallel C_2$. Attribute environment Γ is a partial map $\mathcal{A} \to \mathcal{V}$ from the set of attribute identifiers $\mathcal{A}$ to a set of values $\mathcal{V}$ that can be integers, strings or tuples, etc. We are not interested in giving formal syntax of values; the only constraint is that values do not overlap with the set of attribute names, i.e., $\mathcal{A} \cap \mathcal{V} = \emptyset$. The interface $I \subseteq dom(\Gamma)$ is a set of exposed attribute names used for controlling interactions between components.

Example 1.1 (An AbC system of SMP). The SMP can be thought as an AbC systems where each participant is modeled as an autonomous AbC component interacting with the others to form a stable matching. There are two types of components corresponding to men and women. Their attributes names can be defined as follows:

- id: a non negative integer representing person's unique identity
- partner: the matched partner (initially set to value -1)
- pref: the preference list of identifiers

Each woman is modeled as an AbC component of the form $W_i = \Gamma_i :_{\{id\}} P_W$. Each man is modeled as an AbC component of the form $M_i = \Gamma_i :_{\{id\}} P_M$. Thus the interaction interface between components is the person identity. This is because the interaction predicates of components only refer to the attribute id of each other, as the next example will show. A system of n men and n women is then rendered as:

$$SYS \triangleq M_1 \parallel M_2 \parallel \ldots \parallel M_n \parallel W_1 \parallel W_2 \parallel \ldots \parallel W_n$$

The result of any matching procedure is a complete and stable matching. The matching algorithm is encoded in the behaviour of men and women components through processes P_M and P_W as detailed in Example 1.2. □

$$
\begin{aligned}
&\text{(Components)} && C ::= \Gamma :_I P \mid C_1 \parallel C_2 \\
&\text{(Processes)} && P ::= 0 \mid (\tilde{E})@\Pi.U \mid \Pi(\tilde{x}).U \mid \langle\Pi\rangle P \mid P_1 + P_2 \mid P_1|P_2 \mid K \\
&\text{(Update)} && U ::= \ [a := E]U \mid P \\
&\text{(Predicates)} && \Pi ::= \mathrm{tt} \mid \mathrm{ff} \mid p(\tilde{E}) \mid \Pi_1 \ \wedge \ \Pi_2 \mid \neg\Pi \\
&\text{(Expressions)} && E ::= v \ \mid \ x \ \mid \ a \ \mid \ this.a \ \mid \ f(\tilde{E})
\end{aligned}
$$

FIGURE 2.1: *AbC* syntax

The behaviour of a component is modeled by a process P executing actions. A process can be either an inactive process 0, a prefixing process $\alpha.U$ where the continuation process U may have an attribute update $[a := E]$ preceding it, an awareness process $\langle\Pi\rangle P$, a choice process $P_1 + P_2$, a parallel process $P_1|P_2$, or a process call K under the assumption that each process has a unique definition $K \triangleq P$.

An expression E may be a constant value v, a variable x, an attribute name a, a reference $this.a$ to attribute a in the environment of the executing component or a k-place function f on a sequence of expressions $\tilde{E}$. A predicate Π can be either boolean constants tt, ff, a *decidable* k-place predicate p on a sequence of expressions $\tilde{E}$, a logical conjunction of two predicates $\Pi_1 \wedge \Pi_2$ or a negation of a predicate $\neg\Pi$.

The evaluation of an expression E under an environment Γ is denoted by $[\![E]\!]_\Gamma$. It has the effect of replacing the occurrences of a and $this.a$ with the value $\Gamma(a)$. The partial evaluation of a predicate Π under an environment Γ is denoted by $\{\Pi\}_\Gamma$. It returns a predicate Π' obtained from Π by replacing every occurrence of $this.a$ in Π with $\Gamma(a)$. We write $\Gamma \models \Pi$ to state that predicate Π holds in environment Γ. Figure 2.2 defines the satisfaction relation between a predicate Π and an environment Γ.

The actions of an *AbC* component are:

$(\tilde{E})@\Pi$: the *attribute-based output* that is used to send the evaluation of a sequence of expressions $\tilde{E}$ to those components whose attributes satisfy predicate Π;

$\Pi(\tilde{x})$: the *attribute-based input* that binds to the sequence of variables $\tilde{x}$ the message received from any component whose attributes, and possibly the communicated values satisfy the receiving predicate Π.

Thus, the only syntactic difference between sending and receiving predicates is that the latter can also refer to the values in the message.

An attribute update $[a := E]$ assigns the evaluation of expression E to attribute a. In *AbC*, attribute update is interpreted as the side-effect of communication actions. The execution of an action and following update(s) (if there any) is atomic.

An awareness predicate Π blocks the execution of a process until it is satisfied. This predicate is different from the ones used for sending and receiving in that it can only refer to attributes in the local environment. The activation of an awareness predicate and the execution of any action that follows are also atomic.

The choice process $P_1 + P_2$ can behave either like P_1 or P_2. The parallel process $P_1|P_2$ interleaves the executions of P_1 and P_2. Thus, a component may have parallel processes operating on the same attribute environment. Co-located processes residing within one component do not communicate. They simply interleave and may indirectly affect the behaviour of the others by modifying the shared environment.

Variables occurring in a process term are bound by input actions. Specifically, $\Pi(\tilde{x}).U$ acts as a binder for $\tilde{x}$ in U. This implies that the three types of predicates, i.e., awareness, sending, and receiving, may also contain variable names.

Remarks. In *AbC* there are three kinds of predicates.

- *Awareness predicate* can only relate the local attributes of the executing component.
- *Sending predicate* can relate the attributes of the local component and those of other components.
- *Receiving predicate* can relate the attributes of the local component and those of other components, in addition to the message to be received.

The interaction predicates (send and receive) are meant to express the properties of the potential partners who the local component wants to interact with. They can not relate the attributes among two or more external components. In fact, the design of *AbC* emphasizes on *mutual interest* between a sender and a receiver (see the rules of input and output actions in Figure 2.3).

For the same reason, there is no support for quantifiers (e.g., $\forall$,$\exists$) to be used along with the interaction predicates. Nevertheless, we still have a sense of quantifiers (at the communication time) in that an output action select all components according to the sending predicate (hence $\forall$) and an input action receives from any component according to a receiving predicate (hence $\exists$). This is the natural consequence of input and output semantics.

We consider *AbC* predicates as boolean-value functions from variables (in our case, attributes and messages) to truth values. In Figure 2.1, predicates (and expressions) may have complex forms to which we deliberately omitted the precise syntax; we only refer them as k-place operators over expressions.

Example 1.2 (Gale Shapley Algorithm). We continue to specify the behaviour of men and women components following the algorithm conceived by Gale and Shapley. The classical algorithm

$$
\begin{array}{ll}
\Gamma \models \mathrm{tt} & \text{for all } \Gamma \\
\Gamma \not\models \mathrm{ff} & \text{for all } \Gamma \\
\Gamma \models p(E_1, \ldots, E_k) & \text{iff } p(\llbracket E_1 \rrbracket_\Gamma, \ldots, \llbracket E_k \rrbracket_\Gamma) \text{ is true} \\
\Gamma \models \Pi_1 \wedge \Pi_2 & \text{iff } \Gamma \models \Pi_1 \text{ and } \Gamma \models \Pi_2 \\
\Gamma \models \neg \Pi & \text{iff not } \Gamma \models \Pi
\end{array}
$$

FIGURE 2.2: The satisfaction relation.

of [65] goes through a sequence of proposals initiated by members of one group (*the proposers*) according to their preferences. Members of the other group (*the responders*) after receiving a proposal, do choose the best proposer between their current partner and those making advances. It can be proved that such an algorithm guarantees existence of a unique stable matching. In the message passing model, we let a man send a proposal message to the first woman in his list and assume the woman becomes his partner until he gets rejected. When a man gets a reject message, he makes a proposal to the next woman in his preference list.

This behaviour of a man component can be encoded in AbC as follows:

$$
\begin{aligned}
P_M &\triangleq (\text{‘}\mathit{proposal}\text{’}, \mathrm{this.id})@(\mathrm{id} = \mathrm{hd}(\mathrm{this.pref})).[\mathrm{partner} := \mathrm{hd}(\mathrm{pref}), \mathrm{pref} := \mathrm{tl}(\mathrm{pref})]P'_M \\
P'_M &\triangleq (x = \text{‘}\mathit{no}\text{’})(x).[\mathrm{partner} := -1]P_M
\end{aligned}
$$

We assume the existence of 1-arity functions *hd* and *tl* that take a list as parameter and return the head and the rest of the list, respectively. Literal constants are quoted to avoid confusion with attribute names. P_M basically consists of two actions, an attribute-based send and an attribute-based receive. The output action uses a sending predicate as $id = hd(this.pref)$, in which id represents the attribute of other components and $this.pref$ is the value of $pref$ of the executing component. The message to be sent contains two elements, a constant and the value of attribute id. By executing this action, the process sends the message to any component whose value of id satisfies the predicate. As a side effect, the attribute *partner* gets updated and the first element is removed from *pref*. After that, P_M waits for a message. In our example, the receiving predicate only constraints on the message content. If there is one message such that its content equals to ‘no’, the attribute *partner* is reset atomically, and P_M is recursively called.

A woman on the other hand only waits for incoming proposals. For each proposal, she compares the new man with her current partner according to her preference list and keeps the better man as partner while rejecting the other one. This behaviour is encoded as follows:

$$
\begin{aligned}
P_W &\triangleq (x = \text{‘}\mathit{propose}\text{’})(x, y).(H_W \mid P_W) \\
H_W &\triangleq \langle \mathrm{bof}(\mathrm{y}, \mathrm{partner}, \mathrm{pref})\rangle(\text{‘}\mathit{no}\text{’})@(\mathrm{id} = \mathrm{this.partner}).[\mathrm{partner} := y]0 \\
&\quad + \langle \neg \mathrm{bof}(\mathrm{y}, \mathrm{partner}, \mathrm{pref})\rangle(\text{‘}\mathit{no}\text{’})@(\mathrm{id} = y).0
\end{aligned}
$$

P_W first waits for a message containing two elements such that the first element is 'propose', and then continues as a parallel process. The continuation is composed of two processes: H_W that handles the received message and P_W that guarantees handling of future proposals. This replication mechanism is useful to allow a component to receive multiple messages in parallel.

In the code of process H_W, we handle two cases by using a choice operator. Each branch is guarded with the condition encoded via a boolean-valued function namely bof. The function returns true if the first parameter precedes the second parameter in list $pref$. Intuitively, the first branch is the case where the sender identified by y is better than the current $partner$. The activation of this awareness predicate leads to a message 'no' that is sent to all components whose id equals to the value of $partner$. Atomically, $partner$ is updated in the local environment. In the second branch, we encode a reverse behaviour where y is considered not better than $partner$ and thus a 'no' message is sent. □

2.1.2 *AbC* semantics

The operational semantics of *AbC* is defined at two different levels: component semantics and system semantics. Component semantics is useful for reasoning about the behaviour of an individual component when it performs communication actions. System semantics is useful for modeling the evolution of the system as a whole, in particular for parallel composition of sending and receiving components. Below, we explain the main semantic rules given in Figures 2.3 and 2.4, where the transition relation $\mapsto$ describes components' behaviour and $\rightarrow$ describes systems' behaviour.

The semantics is given in terms of labeled transition systems. The transition labels are of the following forms:

$$\lambda ::= \Gamma \triangleright \overline{\Pi}(\tilde{v}) \mid \Gamma \triangleright \Pi(\tilde{v}) \qquad \alpha ::= \lambda \mid \widetilde{\Gamma \triangleright \Pi(\tilde{v})}$$

λ labels are used to represent input $(\Gamma \triangleright \Pi(\tilde{v}))$ and output $(\Gamma \triangleright \overline{\Pi}(\tilde{v}))$ actions. α labels includes also a negative label $\widetilde{\Gamma \triangleright \Pi(\tilde{v})}$ to model the case a component is unable to receive a message. The output label contains the sender predicate, the message and the portion environment as limited by the interface of the sender. We will sometimes make this intention clear by writing explicitly $\Gamma \downarrow I$ (that is $(\Gamma \downarrow I)(a) = \Gamma(a)$ if $a \in I$, and undefined otherwise). Each component can either receive the message by performing the input action or discard it by exhibiting the negative label.

Rule **Brd** models the execution of an output action. When a component can do this action, it broadcasts the evaluation result $\tilde{v}$ of expressions $\tilde{E}$, the closure Π' of the sending predicate Π under Γ (denoted by $\{\Pi\}_\Gamma$ which replaces all the occurrences of $this.a$ with $\Gamma(a)$), along with the portion of environment $\Gamma \downarrow I$. The executing process evolves from $(\tilde{E})@\Pi.U$ to U. The status of the sending

$$\text{Brd}\ \frac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad \{\Pi\}_\Gamma = \Pi'}{\Gamma :_I (\tilde{E})@\Pi.U \xmapsto{\Gamma\downarrow I \triangleright \overline{\Pi'}(\tilde{v})} \{\!|\Gamma :_I U|\!\}} \qquad \text{FBrd}\ \frac{}{\Gamma :_I (\tilde{E})@\Pi.U \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma :_I (\tilde{E})@\Pi.U}$$

$$\text{Rcv}\ \frac{\Gamma' \models \{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma \quad \Gamma \models \Pi'}{\Gamma :_I \Pi(\tilde{x}).U \xmapsto{\Gamma' \triangleright \Pi'(\tilde{v})} \{\!|\Gamma :_I U[\tilde{v}/\tilde{x}]|\!\}} \qquad \text{FRcv}\ \frac{\Gamma' \not\models \{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma \ \vee\ \Gamma \not\models \Pi'}{\Gamma :_I \Pi(\tilde{x}).U \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma :_I \Pi(\tilde{x}).U}$$

$$\text{Aware}\ \frac{\Gamma \models \Pi \quad \Gamma :_I P \xmapsto{\lambda} \Gamma :_I P'}{\Gamma :_I \langle\Pi\rangle P \xmapsto{\lambda} \Gamma :_I P'} \qquad \text{FAware1}\ \frac{\Gamma \models \Pi \quad \Gamma :_I P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma :_I P}{\Gamma :_I \langle\Pi\rangle P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma :_I \langle\Pi\rangle P}$$

$$\text{FAware2}\ \frac{\Gamma \not\models \Pi}{\Gamma :_I \langle\Pi\rangle P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma :_I \langle\Pi\rangle P}$$

$$\text{Choice}\ \frac{\Gamma :_I P_1 \xmapsto{\lambda} \Gamma' :_I P_1'}{\Gamma :_I P_1 + P_2 \xmapsto{\lambda} \Gamma' :_I P_1'} \qquad \text{Int}\ \frac{\Gamma :_I P_1 \xmapsto{\lambda} \Gamma' :_I P_1'}{\Gamma :_I P_1|P_2 \xmapsto{\lambda} \Gamma' :_I P_1'|P_2}$$

$$\text{FChoice}\ \frac{\Gamma :_I P_1 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_1 \quad \Gamma :_I P_2 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_2}{\Gamma :_I P_1 + P_2 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_1 + P_2}$$

$$\text{FInt}\ \frac{\Gamma :_I P_1 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_1 \quad \Gamma :_I P_2 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_2}{\Gamma :_I P_1|P_2 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P_1|P_2}$$

FIGURE 2.3: *AbC*'s component semantics

component however depends on if the process U contains any attribute update or not. The function $\{\!|C|\!\}$ is interpreted as follows:

$$\{\!|C|\!\} = \begin{cases} \{\!|\Gamma[a \mapsto [\![E]\!]_\Gamma] :_I U|\!\} & C \equiv \Gamma :_I [a := E]U \\ \Gamma :_I P & C \equiv \Gamma :_I P \end{cases} \tag{2.1}$$

where $\Gamma[a \mapsto v]$ is an updated environment such that $\Gamma[a \mapsto v](a) = v$ and $\Gamma[a \mapsto v](a') = \Gamma(a')\ \forall\ a' \neq a$. That is, attribute updates are consumed one by one until U becomes P with P not prefixed by any update. It is however more convenient to write a series of attribute updates as $[a_1 := E_1, a_2 := E_2, \ldots]$, instead of $[a_1 := E_1][a_2 := E_2], \ldots$.

Example 1.3 (The transition of attribute-based output). We illustrate how an attribute-based output takes place in men components when they send out proposal messages. If we take $M_1 = \Gamma_1 :_{\{id\}} P_M$ with $\Gamma_1 = \{id = 1, pref = [3,4]\}, partner = -1\}$ and the process P_M described in Example 1.2, by applying the rule *Brd*, we can have the following transition:

$$\frac{[\![('propose', this.id)]\!]_{\Gamma_1} = ('propose', 1) \quad \{id = hd([3,4])\}_{\Gamma_1} = (id = 3)'}{\Gamma_1 :_I P_M \xmapsto{\{id=1\} \triangleright \overline{id=3}('propose',1)} \Gamma_1[partner \mapsto 3, pref \mapsto [4]] :_{\{id\}} P_M'}$$

□

If a component can only offer an output action, it would discard any incoming messages and stay unchanged. This fact is expressed by the rule **FBrd** with the discarding label $\Gamma \triangleright \widetilde{\Pi'(\tilde{v})}$.

Rule **Rcv** modeling the execution of an input action states that a receiving component can perform an input action to receive a sequence of values $\tilde{v}$, in such case the substitution $[\tilde{v}/\tilde{x}]$ is applied to the continuation process U if two conditions hold:

1. the sending predicate Π' is satisfied in the receiving environment Γ;
2. the partial evaluation of the receiving predicate Π under the local environment, after binding the variables $\tilde{x}$ to the received values $\tilde{v}$, is satisfied in the portion of sender's environment Γ'.

In case one of the two conditions above is not satisfied the receiving component discards the incoming message and stays unchanged, as shown by the rule **FRcv**.

Example 1.4 (The transition of attribute-based input). Let us consider a system of 4 components of two men and two women, with the first component being M_1 as in the previous example and the remaining three components with the following attribute environments.

SYS ::= $\Gamma_1 :_I P_M \parallel \Gamma_2 :_I P_M \parallel \Gamma_3 :_I P_W \parallel \Gamma_4 :_I P_W$ where $I = \{id\}$ and
$\Gamma_2 = \{id = 2, pref = [3, 4], partner = -1\}$
$\Gamma_3 = \{id = 3, pref = [1, 2], partner = -1\}$
$\Gamma_4 = \{id = 4, pref = [1, 2], partner = -1\}$

By applying rule FRcv, we can derive the following transitions:

$$\Gamma_2 :_{\{id\}} P_M \xmapsto{\{id=1\}\triangleright\widetilde{(id=3)}('propose',1)} \Gamma_2 :_{\{id\}} P_M$$
$$\Gamma_4 :_{\{id\}} P_W \xmapsto{\{id=1\}\triangleright\widetilde{(id=3)}('propose',1)} \Gamma_4 :_{\{id\}} P_W$$

This is because the component M_2 can only perform an attribute-based output, whereas the component W_4 discards the message because the sender predicate $(id = 3)$ is not satisfied in its attribute environment Γ_4.

On the other hand, by using rule Rcv, we can derive the following transition:

$$\Gamma_3 :_{\{id\}} P_W \xmapsto{\{id=1\}\triangleright(id=3)('propose',1)} \Gamma_3 :_{\{id\}} H_W[1/y] \mid P_W$$

□

$$\text{ICom}\ \frac{\Gamma :_I P \overset{\lambda}{\mapsto} \Gamma' :_I P'}{\Gamma :_I P \overset{\lambda}{\to} \Gamma' :_I P'} \qquad \text{FCom}\ \frac{\Gamma :_I P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P}{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \Pi'(\tilde{v})} \Gamma :_I P}$$

$$\text{Com}\ \frac{C_1 \xmapsto{\Gamma' \triangleright \overline{\Pi} \tilde{v}} C_1' \quad C_2 \xmapsto{\Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\overline{\Pi} \tilde{v}} C_1' \parallel C_2'} \qquad \text{Sync}\ \frac{C_1 \xmapsto{\Pi(\tilde{v})} C_1' \quad C_2 \xmapsto{\Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\Pi(\tilde{v})} C_1' \parallel C_2'}$$

FIGURE 2.4: AbC's communication rules

Rules **Aware** and **FAware1** describe the semantics of the awareness operator: the component $\Gamma :_I \langle\Pi\rangle P$ behaves as $\Gamma :_I P$ only when $\Gamma \models \Pi$, otherwise it gets stuck (**FAware2**).

The remaining four rules describe the behaviour of co-located processes inside a component. Choice and parallel processes have the usual semantics when they are able to perform input or output actions. For simplicity we have omitted the symmetric rule of **Choice** and **Int**. The interesting case is when they discard a message if both subprocesses discard that message (**FChoice**, **FInt**). Although not presented here, a process invocation on an identifier K has the same behaviour as its process definition P and an inactive process 0 can only discard messages.

The *AbC* system semantics is briefly presented in Figure 2.4. Rules (**ICom**) and (**FCom**) lift the transitions from component level to system level. Rule (**Com**) states that two parallel components C_1 and C_2 can communicate if C_1 can send a message and C_2 can receive it. Rule (**Sync**) states that multiple components can receive the same message and in such case, they evolve together in a single move. Note in rule **FCom** that, at the system level, discarding of messages by a component, cannot be observed. That is, an external observer of the system can not tell if a message is consumed or discarded by a component.

Example 1.5 (An interaction fragment of SMP). Consider again the SMP system in our example M1 sent a proposal message:

$$\Gamma_1 :_I P_M \xmapsto{\{id=1\} \triangleright \overline{id=3}('propose',1)} \overbrace{\Gamma_1[partner \mapsto 3, pref \mapsto [4]] :_{\{id\}} P_M'}^{M_1'}$$

As explained in Example 1.4, M_2 and W_4 discard this message while W_3 receives it. The whole system thus evolves with rule COM with the synchronization of M_1 and W_3: $M_1 \parallel M_2 \parallel W_3 \parallel W_4 \xmapsto{\overline{id=3}('propose',1)} M_1' \parallel M_2 \parallel \Gamma_3 :_{\{id\}} H_W[1/y] \mid P_W \parallel W_4$ $\square$.

2.2 Case studies: *AbC* at work

Having introduced syntax and semantics of the *AbC* calculus, we proceed in presenting two additional case studies with the aim of showing how non trivial interaction protocols may be programmed with

the AbC primitives.

2.2.1 Stable marriage with attributes

The classical version of SMP is helpful to illustrate the basic constructs of AbC, however the communications among components are essentially point-to-point as the interaction predicates are based on person's identity.

We now consider a variant where men components propose with predicates over women's characteristics instead of their identifiers. Specifically, we extend the set of attributes of men and women to include personal *characteristics*. The notion of *preferences* is also adjusted to consider specific values of potential partner characteristics that a person wants to match. In this respect, each agent can be associated with several attributes as follows:

- $m_1, \ldots, m_k$, $w_1, \ldots, w_k$: k characteristics of a man and of a woman, respectively
- $pw_1, \ldots, pw_k$, $pm_1, \ldots, pm_k$: k preferences that a man and a woman have over potential partner characteristics, respectively

To express the notion of preference lists as in the classical case for men and women, we build a list of predicates for agents. Each predicate in the list is a conjunction of equality check between the preferences of an agent and the characteristics of potential partners. In particular, we consider the highest demand that a man possessing the preferences pw_i can have over women as: $\Pi_1 = (pw_1 = w_1 \wedge pw_2 = w_2 \wedge \ldots \wedge pw_k = w_k)$. This means that a man prefers the most a parter who satisfies all the preferences. Other predicates in the list are derived from Π_1. A simple schema could be allowing agents to drop one checking clause at a time[1]. For example, $\Pi_2 = (pw_1 = w_1 \wedge pw_2 = w_2 \wedge \ldots \wedge pw_{k-1} = w_{k-1})$ is the relaxation of Π_1, i.e., when the last requirement, pw_k, is dropped. Interestingly, $\Pi_k = tt$ states that the agent is willing to accept any member of the opposite set.

Our goal is the same as the classical problem, i.e., to obtain a complete and stable matching. We also assume the two sets Men and Women have the same number of elements n. In our model, men actively use predicates in the list as sending predicates to propose themselves. Women wait for proposal messages and do the comparison and rejection as before. Given a particular demand (or requirement) Π_i of a man m, we can define the notion that m prefers some woman w to other w' by comparing the number of characteristics of w and w' that match m's preferences. A similar reasoning is applied for women.

Before presenting an AbC specification of the problem, we explain the matching algorithm. The key idea is based on the following observations. Proposal messages from a man may target zero or many

[1] This is to make our variant realistic, as dropping the preferences results in less selective predicates and in an increase of the chances of finding a partner.

women. In the first case, the man tries with the next predicate in the list. In the second case, extra acknowledgment messages are needed to select a partner. Later on, he may be rejected by his partner, and he moves on with the next predicate if all interested women have rejected him. In both situations, the man needs to know how many answers he may receive.

The behaviour of a man is as follows. Before making a proposal, he learns about the number of women who may answer his proposal. Without further complicating the protocol, we assume that each man knows the total number of women in the system, i.e., n. Otherwise, this number can be learnt from an additional agent who keeps track of the number of agents joining the matching system. For instance, a more involved interaction protocol would need a special agent working in parallel with other agents. Men and women which join

- **Querying.** For a specific requirement Π_i, man m prepares a query message q_i and sends it to all women. m waits for n messages from women and counts the number of interest messages using a counter c. If there is no interest messages, m restarts this phase using the next requirement Π_{i+1}, otherwise proceeds with the proposing phase.
- **Proposing.** Man m sends a propose message, and waits for c messages. As soon as he receives a first yes from a woman (if any), he sends her a confirm message and considers himself engaged to her. He then rejects any other yes answer by sending out a *toolate* message. If he receives a no message, the sender is added to a black list.
- **Waiting.** After a proposing phase, a man can be either alone or engaged. In the first case, he relaxes the current predicate and goes back to the querying phase. In the second case, he takes no action unless he receives a bye message from his partner. He adds the new ex-partner to the black list and propose again without relaxing his predicate, eliminating those in the black list.

On the other hand, a woman keeps listening for messages and processes them as follows.

- **Answering.** If a woman receives a querying message, she compares the attributes against the requirement specified in the message, then she replies to the sender either an *interested* or an *uninterested* message.
- **Processing.** If a woman receives a proposal message, she compares the attributes of the sender (attached in the message) against those of the current partner. She then replies with a *yes* message if the proposer is better or a *no* message if the proposer is not better. If a woman receives a $confirm$ message from the proposer, it means that she answered with a *yes* before. She breaks with her current partner (if any) by sending it a *bye* message, and takes the new man as partner.

The above matching procedure is a natural extension of the classical algorithm by Gale Shapley. The key to obtain stability is that a man proceeds with a relaxed requirement only when all the potential partners satisfying a previous requirement have rejected him. In this way, the algorithm can find man-optimal matching(s) (i.e., matchings where each man gets the best possible partner among other existing matchings).

To make our model concrete, we consider $k = 2$ and define explicitly the following demands for men: $\Pi_1 = (pw_1 = w_1 \wedge pw_2 = w_2)$, $\Pi_2 = (pw_1 = w_1)$, $\Pi_3 = (pw_2 = w_2)$, $\Pi_4 = tt$ with the intuition that: if a man fails to get a partner with a given requirement Π_i, he continues the matching procedure with a lower requirement Π_{i+1}. In the extreme case, a man uses Π_4 which would broaden the target set of women with the hope to find a partner eventually.

In our variant, the matching system is the parallel compositions of men and women, i.e., $M_1 \parallel M_2 \parallel \ldots \parallel M_n \parallel W_1 \parallel W_2 \parallel \ldots \parallel W_n$. Each man is defined as $M_i \triangleq \Gamma_i :_{\{id,m_1,m_2\}} P_M$ where Γ_i and P_M represent its attribute environment and behaviour, respectively. Similarly, each women is represented as $W_j \triangleq \Gamma_j :_{\{id,w_1,w_2\}} P_W$ with the attribute environment Γ_j and behaviour P_W.

We can now introduce the behaviour of a man. In addition to the attributes representing characteristics and preferences mentioned above, the attribute environment of men contains also:

- id: a non negative integer representing man's identity
- $partner$: the current partner identity, initially, $partner = 0$
- $sent$: an attribute used to disable or enable sending propose messages, initially $sent = 0$
- bl: a set storing women identities who have rejected a man, initially $bl = \emptyset$
- n: the number of women in the system
- c: a counter that counts the number of "interest" answers after a man sends a query message, initially $c = 0$
- $counter$: a counter that counts the number of answers after a man sends a query message, initially $counter = 0$

Note the role of the list bl. If a woman w rejects a man m, it must happen that w has already paired with another m' who is better than m. bl is used to prevent the possibility of m contacting with w again in the future iterations. This does not violate the stability condition because the pair (m,w) is not a blocking pair.

The process P_M is a parallel composition of several processes. Process Q defines the querying phase, process P defines the proposing phase, processes R and A together implement the waiting phase.

$$P_M ::= Q \mid P \mid R \mid A$$

Q is in charge of querying the number of women in the system whose characteristics match a given requirement. Recall that we explicitly consider requirements as Π_1, Π_2 Π_3, Π_4. Q in turns contains several parallel processes; they are programmed to be executed in a given order by relying on the attribute *counter*. The number of matched women is stored in the attribute c. The code of Q is reported below.

$$
\begin{aligned}
Q &\triangleq Q_1 \mid Q_2 \mid Q_3 \mid Q_4 \\
Q_1 &\triangleq \langle \text{counter} = 0 \rangle (`q_1', \mathit{this.id}, \mathit{this.pw}_1, \mathit{this.pw}_2)@(\text{id} \notin \text{this.bl}). \\
&\quad [\text{counter} := \text{counter} + |\text{bl}|] Q_1' \\
Q_1' &\triangleq \langle \text{counter} < \text{n} \rangle (x = `\mathit{interest}')(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1] Q_1' \\
&\quad + \langle \text{counter} < \text{n} \rangle (x = `\mathit{uninterest}')(x).[\text{counter} := \text{counter} + 1] Q_1' \\
Q_2 &\triangleq \langle \text{counter} = \text{n} \wedge \text{c} = |\text{bl}| \rangle (`q_2', \mathit{this.id}, \mathit{this.pw}_1, \mathit{this.pw}_2)@(\text{id} \notin \text{this.bl}). \\
&\quad [\text{counter} := \text{counter} + |\text{bl}|] Q_2' \\
Q_2' &\triangleq \langle \text{counter} < 2 * \text{n} \rangle (x = `\mathit{interest}')(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1] Q_2' \\
&\quad + \langle \text{counter} < 2 * \text{n} \rangle (x = '\mathit{uninterest}')(x).[\text{counter} := \text{counter} + 1] Q_2' \\
Q_3 &\triangleq \langle \text{counter} = 2 * \text{n} \wedge \text{c} = |\text{bl}| \rangle (`q_3', \mathit{this.id}, \mathit{this.pw}_1, \mathit{this.pw}_2)@(\text{id} \notin \text{this.bl}). \\
&\quad [\text{counter} := \text{counter} + |\text{bl}|] Q_3' \\
Q_3' &\triangleq \langle \text{counter} < 3 * \text{n} \rangle (x = `\mathit{interest}')(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1] Q_3' \\
&\quad + \langle \text{counter} < 3 * \text{n} \rangle (x = `\mathit{uninterest}')(x).[\text{counter} := \text{counter} + 1] Q_3' \\
Q_4 &\triangleq \langle \text{counter} = 3 * \text{n} \wedge \text{c} = |\text{bl}| \rangle ()@(\text{ff}).[\text{counter} := \text{counter} + 1] 0
\end{aligned}
$$

At the beginning, Q_1 sends a query message of a form $(`q_1', \mathit{this.id}, \mathit{this.pw}_1, \mathit{this.pw}_2)$ to all women. After that it counts the number of 'interest' replies from women and ignores all 'uninterest' messages. Other querying processes Q_i are activated if conditions on c and *counter* are satisfied. Intuitively, the condition on *counter* is to make sure that processes Q_i are activated one after another. The important bit is at the role of the set *bl* which contains ids of women who have rejected the man. Thus, in specifying sending predicates, the set of potential destinations are only those whose ids are not in the *bl*. The condition $c = |bl|$ implies that either there is no 'interest' messages received, or all the interested women have been added to *bl*. In both cases, the control is passed to the next querying process. This condition on c, as we shall see, depends on the proposing process P in which the set *bl* is incrementally expanded.

Process P contains several branches that send proposal messages according to a given demand or requirement. The propose message contains a specific tag 'propose' and characteristics of a man.

Process P is defined below.

$$\begin{aligned}
P \triangleq\ & \langle counter = n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{'}propose\text{'}, this.id, this.w_1, this.w_2)@(\Pi_1 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 2 * n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{'}propose\text{'}, this.id, this.w_1, this.w_2)@(\Pi_2 \wedge\ \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 3 * n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{'}propose\text{'}, this.id, this.w_1, this.w_2)@(\Pi_3 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 3 * n + 1 \wedge partner = 0 \wedge n > |bl| \wedge sent = 0\rangle \\
& (\text{'}propose\text{'}, this.id, this.w_1, this.w_2)@(\Pi_4 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P
\end{aligned}$$

Again in here, we use *counter* in the awareness predicates to schedule the execution of each branch of P: the first branch takes place only after Q_1 has finished, the second one follows the completion of Q_2, and so on. Furthermore, in each branch of P, the attribute *sent* is used to control when to send a propose message. The other conditions state that a given branch is activated, i.e., a 'propose' message should be sent if there are still interested women ($c > |bl|$) and if the agent has no partner ($partner = 0$).

Process A is used to handle 'yes' messages. If a man has no partner, he sends a confirm message to the sender and updates the attribute *partner*. Otherwise, he answers with a 'toolate' message. After receiving a 'yes' message, process A (like Example 1.2) is activated. This is needed to receive other 'yes' messages when process H is active.

$$\begin{aligned}
A \triangleq\ & (x = \text{'}yes\text{'})(x, y).(H \mid A) \\
H \triangleq\ & (\langle partner = 0\rangle(\text{'}confirm\text{'})@(id = y).[partner := y]0 \\
& + \langle partner > 0\rangle(\text{'}toolate\text{'})@(id = y).0
\end{aligned}$$

Finally, process R is used to handle the case a man receives 'bye' and 'no' messages. A 'bye' message can come from the man's partner, in which case, the attribute partner is reset. A 'no' message is received from a woman who rejects the man's proposal. In both cases, the man adds the *id* of the sender to the attribute *bl* to avoid to contact her in the future. Furthermore, the attribute *sent* is also reset, potentially triggering the sending of new proposal in the process P.

$$\begin{aligned}
R \triangleq\ & (x = \text{'}bye\text{'})(x, y).[partner := 0, bl := bl \cup \{y\}, sent := 0]R \\
& + (x = \text{'}no\text{'})(x, y).[bl := bl \cup \{y\}, sent := 0]R
\end{aligned}$$

The behaviour of a woman component is specified by process P_W. It is the parallel composition of R_j processes implementing the answering phase and a W process implementing the processing phase.

$$P_W \triangleq P_1 \mid P_2 \mid P_3 \mid W$$

The attributes of a woman include:

- *id*: the unique identity
- w_1,w_2: the characteristics of a woman
- pm_1,pm_2: the preferences of a woman has over mens
- *partner*: the current partner identity, initially partner = 0
- cm_1, cm_2: the characteristics of the current partner, initially set to 0
- *lock*: the attribute implementing a lock in order to sequentialize the processing of *propose* messages, because a woman may receive many of them in parallel. Initially $lock = \text{ff}$.
- *bl*: a set storing men identifiers to which a woman does not consider their propose messages, initially $bl = \emptyset$

The processes P_1, P_2, P_3 handle querying messages tagged with 'q_1', 'q_2', 'q_3', respectively. For each query identified by such a 'q_i', a woman performs appropriate checks and reply with either an 'interest' or 'uninterest' message. For example, the tag 'q_1' indicates the highest level of requirement that a sender is asking. The process P_1, handling this type of query is defined as below.

$$\begin{aligned}
P_1 &\triangleq (x = \text{`}q_1\text{'})(x, y, z, t).(H_1 \mid P_1) \\
H_1 &\triangleq \langle (z = w_1 \wedge t = w_2) \rangle (\text{`}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(z = w_1 \wedge t = w_2) \rangle (\text{`}\mathit{uninterest}\text{'})@(id = y).0
\end{aligned}$$

The process sends an 'interest' message to the sender if the agents characteristics match the requirement (i.e., $z = w_1 \wedge t = w_2$). Otherwise, it says 'uninterest'. The parallel composition of P_1 with itself is to make sure that multiple querying messages of this type are eventually processed. Processes P_2 and P_3 have similar definitions:

$$\begin{aligned}
P_2 &\triangleq (x = \text{`}q_1\text{'})(x, y, z, t).(H_2 \mid P_2) \\
H_2 &\triangleq \langle (z = w_1) \rangle (\text{'}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(z = w_1) \rangle (\text{'}\mathit{uninterest}\text{'})@(id = y).0 \\
P_3 &\triangleq (x = \text{`}q_1\text{'})(x, y, z, t).(H_3 \mid P_3) \\
H_3 &\triangleq \langle (t = w_2) \rangle (\text{`}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(t = w_2) \rangle (\text{'}\mathit{uninterest}\text{'})@(id = y).0
\end{aligned}$$

Process W handles propose messages sent from men. After a message is received, the process replicates itself in order not to loose other 'propose' messages. The sender, which is the second element in the

message, is added to a black list and the received message is handled by process H.

$$
\begin{aligned}
W &\triangleq (x = \text{‘}propose\text{’} \wedge y \notin this.bl)(x, y, z, t).[\text{bl} := \text{bl} \cup \{\text{y}\}](H \mid W)\\
H &\triangleq A + R\\
A &\triangleq \langle new_man_is_better \wedge \text{lock} = \text{ff}\rangle(\text{‘}yes\text{’}, this.id)@(id = y).[\text{lock} := \text{tt}]A'\\
R &\triangleq \langle \neg\, new_man_is_better\rangle(\text{‘}no\text{’}, this.id)@(id = y).0\\
A' &\triangleq (m = \text{‘}confirm\text{’})(m).B\\
&\quad + (m = \text{‘}toolate\text{’})(m).[\text{lock} := \text{ff}, \text{bl} := \text{bl}\backslash\{\text{y}\}]0\\
B &\triangleq (\text{‘}bye\text{’}, this.id)@(id = this.partner).[\text{lock} := \text{ff}, \text{partner} := \text{y}, \text{cm}_1 := \text{z}, \text{cm}_2 := \text{t}]0
\end{aligned}
$$

H models a choice between two possibilities: if the new man is better than the current partner, a ‘yes’ message is sent (process A), otherwise a ‘no’ message is sent (process R). The attribute *lock* of process A is set to delay the decision making on other ‘propose’ messages that may be received in the meanwhile. After sending out a ‘yes’ message, the agent continues as process A' waiting for either a ‘confirm’ or a ‘toolate’ message.

In the first case, the agent proceeds with the process B that sends a ‘bye’ message to the current partner that becomes the ex one. A list of attribute updates follows: the identity of new man is set as *partner*, his characteristics are kept and the attribute *lock* is reset. Note that in the above specification, the pair (cm_1, cm_2) stores the characteristics of the current partner. An agent, whose preferences are (pm_1, pm_2), compares the current partner against a new proposer with characteristics (z, t) using the condition *new_man_is_better*, which is encoded as:

$$(partner = 0 \vee (z = pm_1 \wedge cm_1 \neq pm_1) \vee (z = cm_1 \wedge t = pm_2 \wedge cm_2 \neq pm_2)$$

In the second case, i.e., the agent gets a ‘toolate’ message, the proposer has accepted another woman as partner. However, he may also propose again. The identity of the proposer, bound to y, is thus removed from the black list *bl* to open this possibility, and the attribute *lock* is reset.

2.2.2 Distributed graph coloring

We consider a distributed variant of the graph coloring problem [30] in which the task is to label the vertices of a graph with different colors (in our case, integers) in such a way that there are no adjacent vertices sharing the same color. This problem can be rendered as a *AbC* system consisting of components, one for each vertex of the graph, that collaborate on the color selection to complete the task without any centralized control. A specification in *AbC* of this scenario has been already presented in [24]. In what follows, we base our description on that specification. The main difference is at a modification presented in the end of this section, to make the protocol more robust. In particular, we spot out a potential issue in the original specification and give it a simple fix. The formal analysis in Chapter 4 will confirm the necessity of this modification, if the reader is not convinced. Furthermore,

we experiment with a slightly alternating behaviour of vertices, in order to gain some efficiency and will confirm this experimentally in Chapter 3.

An undirected graph of N vertices is naturally modeled by components in the form $\Gamma_i :_I V$, each corresponding to a vertex. Each vertex has an unique id and a set of adjacent vertices, which are captured by attributes *id* and *nbr*. Vertices only interact with their neighbors. We define the interaction interface for all vertices as $I = \{id,\ nbr\}$ and interaction predicates of the form $this.id \in nbr$, meaning that the predicates involve components whose *nbr*'s contains *id* of the executing component. Since we are considering undirected graphs, this sending predicate is no different from $id \in this.nbr$. However under a broadcast implementation, such as *ABEL* (Chapter 3), the latter would occupy more bandwidth since the set *nbr* needs to be sent together with the message.

The coloring scenario is then represented as: $\Gamma_1 :_I V \parallel \Gamma_2 :_I V \parallel \ldots \parallel \Gamma_n :_I V$ where V is the process defining the behaviour of each vertex.

To derive a specification for V, we start by illustrating the main algorithmic idea from the point of view of a participating component. Basically, the procedure consists of a sequence of r rounds. During each round, a component i can exchange two types of messages with the neighbors in the set nbr_i: a 'try' message $(`try', c, r)$ is used to inform that it wants to have the color $c$ at round $r$; a 'done' message of the form $(`done', c, r+1)$ indicates that color c is definitely used at the end of round r. At the beginning of each round, component i chooses the first available color c and sends out a 'try' message. Component i also collects 'try' messages from its neighbors to check for the presence of conflict. Vertex i can only attain c as its final color if it has the greatest id among neighbors who are trying to get the same color at the same round. Otherwise, i waits for the startup of a new round, which is triggered by a message, either 'done' or 'try' associated with a round r' such that $r' > r$.

The above algorithm can be implemented in *AbC*. Apart from attributes *id* and *nbr* whose values depend on a specific input graph, components maintain several private attributes for their local computation:

- *round*: the current round that a component is operating
- *counter*: a counter that counts the number of try-messages received from neighbors at a given round
- *done*: counts the number of neighboring vertices that decided on a color
- *color*: the color value
- *used*: the set containing colors that are already used by neighbors
- *constraints*: the set of the proposed colors of neighbors whose ids are greater than the component at a given round

- *send*: the flag controlling the sending of try-message
- *assigned*: the flag indicating the coloring status of a vertex

The values of these attributes are initialized as follows: $round = counter = done = 0, used = constraints = \emptyset, send =$ tt and $assigned =$ ff, $color =$ -1. The pair of attributes *counter* and *constraints* is used to record the status of neighboring components who are still participating the color selection. The pair *done* and *used* is used to record the status of neighbors who have finished their coloring. A component can make decision on taking *this.color* if all neighbors status is known by checking the condition $this.counter + this.done = |this.nbr|$ and if there is no conflict $this.color \notin this.used \cup this.constraints$.

All the components execute the same code specified in V. Process V is defined as a parallel composition of four processes $V \triangleq F \mid T \mid D \mid A$.

Process F chooses the first available color for the vertex and sends this information, along with the current round to neighbor nodes.

$$\begin{aligned} F \triangleq\ & \langle \text{send} \wedge \neg\text{assigned}\rangle ()@(\text{ff}).[color := \min\{\text{i} \notin \text{this.used}\}] \\ & (\text{‘}try\text{’}, \text{this.color}, \text{this.round})@(id \in this.nbr).[\text{send} := \text{ff}]F \end{aligned}$$

where min{i $\notin$ this.used} is a function that returns the smallest element which does not appear in the set *this.used*. Recall that in the *AbC* paradigm, attribute updates are side effects of communication actions. To model a local attribute update, we thus exploit an empty send of the form: $()@(\text{ff}).[a := E]$. In the above code, *color* is assigned the value of expression min{i $\notin$ this.used}. Then, a ‘try’ message, associated with *color* and *round* is sent out. The attribute *send* is turned off as a side effect of this action.

Process T collects ‘try’ messages sent from neighbors. The messages may come from vertices operating during the same round or during a successive round of the algorithm. For each case, the messages may originate from vertices whose ids are greater or smaller than this vertex’s id. Therefore, we have four possibilities implemented as alternative nondeterministic choices.

$$\begin{aligned} T \triangleq\ & (x = \text{‘}try\text{’} \wedge \text{this.id} > \text{id} \wedge \text{this.round} = z)(x, y, z). \\ & [\text{counter} := \text{counter} + 1]T \\ +\ & (x = \text{‘}try\text{’} \wedge \text{this.id} < \text{id} \wedge \text{this.round} = z)(x, y, z). \\ & [\text{counter} := \text{counter} + 1, \text{constraints} := \text{constraints} \cup \{y\}]T \\ +\ & (x = \text{‘}try\text{’} \wedge \text{this.id} > \text{id} \wedge \text{this.round} < z)(x, y, z). \\ & [\text{round} := z, \text{send} := \text{tt}, \text{counter} := 1, \text{constraints} := \emptyset]T \\ +\ & (x = \text{‘}try\text{’} \wedge \text{this.id} < \text{id} \wedge \text{this.round} < z)(x, y, z). \\ & [\text{round} := z, \text{send} := \text{tt}, \text{counter} := 1, \text{constraints} := \{y\}]T \end{aligned}$$

The first two branches deal with messages coming from components executing the same round of the algorithm. The component keeps track of the number of 'try' messages in the attribute *counter*. If the sender's id is greater than the receiver's id, the set *constraints* has to be updated so that the colors of these senders would not be taken by the receiver. The last two branches, if executed, can trigger the execution of a new round. In this case, private attributes are updated accordingly: the new round is set to z and the flag *send* is turned on. Furthermore, the *counter* is reset to 1 because this is also a 'try' message to be collected whereas *constraints* is reset to a value depending on the sender's id.

Process D handles the reception of 'done' messages and possibly triggers a new round of the executing component. The private attribute *done* and *used* are updated to keep track of neighbors who have decided their colors. In case the messages belong to an associated greater round, the attributes related to the execution of a new round, i.e., *counter*, *constraints*, *send* are reset to their initial values, and *round* is set to be z.

$$\begin{aligned} D \triangleq\ & (x = \text{`}\mathit{done}\text{'} \wedge \text{this.round} \geq z)(x, y, z). \\ & [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{y\}]D \\ +\ & (x = \text{`}\mathit{done}\text{'} \wedge \text{this.round} < z)(x, y, z). \\ & [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{\text{y}\}, \\ & \text{send} := \text{tt}, \text{counter} := 0, \text{constraints} := \emptyset, \text{round} := z]D \end{aligned}$$

Process A is used to report the completion of color selection of the executing component. The activation of A has the effect of sending a 'done' message to all neighbors and set *assigned* to tt. A is then terminated, indicating that the component does not participate in the color selection any longer.

$$\begin{aligned} A \triangleq\ & \langle(\text{counter} + \text{done} = |\text{nbr}|) \wedge \text{color} > 0 \wedge \text{color} \notin \text{constraints} \cup \text{used})\rangle \\ & (\text{`}\mathit{done}\text{'}, \text{this.color}, \text{this.round} + 1)@(\text{id} \in \text{this.nbr}).[\text{assigned} := \mathit{tt}]0 \end{aligned}$$

Our modifications. We have made some modifications to the above specification to prevent a critical situation. Note that the process F contains two separate actions: the first selecting a color (f1) and the second sending a 'try' message (f2). Because A and F operate in parallel, it could happen an execution trace containing (f1 – A – f2). This would mean that after reporting the completion of its coloring (by A), there is a possibility for a component to send a 'try' message (by f2). This is because the awareness predicate of F applies only for f1. The execution of f2 therefore causes inconsistent *counter* values at the neighboring components, which prevents their completion.

To fix this problem we may define a process F' that combines the two actions of F into one.

$$\begin{aligned} F' \triangleq\ & \langle\text{send} \wedge \neg\text{assigned}\rangle(\text{`}\mathit{try}\text{'}, \min\{\text{i} \notin \text{this.used}\}, \text{this.round})@(\mathit{id} \in \mathit{this.nbr}). \\ & [\text{color} := \min\{\text{i} \notin \text{used}\}, \text{send} := \text{ff}]F' \end{aligned}$$

Now F' sends out a 'try' message containing the selected color, and as a side effect, updates this value to the attribute *color*.

Moreover, it could happen that, at a given round, a component may execute processes T and D before F and thus obtain some information about the status of its neighbors. When F' takes place, it can make use of this information in proposing the color. In other words, it is better for a component to propose a color which does not appear in *constraints*, because it can not take those in *constraints* at the same round. To introduce this behaviour, we can rewrite F' into

$$F'' \triangleq \langle \text{send} \wedge \neg \text{assigned} \rangle (\text{`}try\text{'}, \min\{\text{i} \notin \text{this.used} \cup \text{this.constraints}\}, \text{this.round})@(id \in this.nbr).$$
$$[\text{color} := \min\{\text{i} \notin \text{used} \cup \text{constraints}\}, \text{send} := \text{ff}]F''$$

Our argument is that using F'' instead of F' can potentially reduce the number of rounds of the algorithm, however depending on the input graph.

The presented case studies introduce a common communication pattern in distributed systems, in which an autonomous agent sends out a message and needs to consider a number of replies before it can take decision. This is a typical situation that often arose in designing coordination protocols, requiring the capability of an agent to react on a set of messages, instead of a single message. In AbC, when a message is sent to destinations specified via a predicate, the sender in general has no idea of how many potential receivers would receive that message. As a consequence, in order to react on a set of reply messages from these destinations, a component must know the number of components who have received the message. This is the case of the stable marriage with attributes, in which, men agents have to learn this number explicitly by sending querying messages before actually involving into the matching protocol. The protocol can be made simple if we have somehow a global view of the system at hand. In the second case study, we do not run into this extra treatment because the number of interacting partners for any vertex is known (i.e., the number of neighbors). In general, however, it is useful to provide additional coordination and synchronization abstractions to ease the programming task. Chapter 5 attempts to provide a simple coordination construct concerning with the mentioned communication pattern. There we will see how the case studies can be expressed with the support of such a construct.

2.3 The Erlang programming language

Erlang [31, 40] is a concurrent functional programming language originally designed for building telecommunication systems [36] and recently successfully adapted to broader contexts, such as large-scale distributed messaging platforms [91, 104]. It supports concurrency [32] and inter-process communication natively through a compact set of powerful primitives. The lightweight and scalable concurrency model and the modularity of functional-style programming [82, 83] make *Erlang* particularly appropriate for building massively scalable distributed systems. The work of this thesis heavily relies on *Erlang* programming as the developed prototypes are entirely based in *Erlang*. This section reviews the relevant features of *Erlang* and of OTP behaviours which are used in our work.

2.3.1 Process

Erlang is a functional language whose syntax is borrowed from Prolog [30]. *Erlang* code is organized into modules which contain functions as building blocks. Functions are first class citizens containing expressions. The value of a function is the value of the last expression in the function body. Recursion is the main mechanism for programming loops.

For example,

```
bof(_,_,[]) -> false;
bof(_,0 _) -> true;
bof(Y,P,[H|T]) ->
  case H of
    Y -> true;
    P -> false;
      _ -> bof(Y,P,T)
  end.
```

is a function named bof that takes three parameters. The function contains 3 clauses. Each clause pattern matches on the value of function parameters where underscores are "don't care" variables. bof essentially checks if Y appears before P in the list, which is pattern matched into a head H and a tail T.

As a functional language, *Erlang* provides immutable data types such as list, tuples, map; it encourages the function style rooted at common concepts such as pure functions (i.e., no side effect), referential transparency (i.e., single assignment), high-order functions (with use of currying and partial evaluation) and pattern matching (on data structures). *Erlang* is dynamically typed; the lack of a static type system was considered as a trade off for its hot-code swapping feature.

To this core, *Erlang* adds concurrency via built-in constructs for processes creation and asynchronous message passing. Unlike operating system's threads or processes, *Erlang* processes are part of the language and implemented by the *Erlang* runtime system (which is eventually implemented in C). Processes have their own memory space which grows and shrinks during their life time. Thus, a garbage collector can work independently on one process while others are still running. This reduces the latency of garbage collection pauses and makes Erlang suitable for soft real-time systems.

Process creation in *Erlang* is obtained by simply calling the built-in $spawn(Fun)$ function that creates a new process and evaluates Fun. This newly created process is a very light weight thread. Each process has an unique identifier which is assigned at the creation time as a result of the spawn function. More importantly, processes can collaborate by point-to-point asynchronous message passing. Process identifiers must be known by others for the communication purpose. Messages that are sent to a process are stored in the receiver (theoretically unbounded) mailbox until they are processed. The arrival order of messages sent from different processes is indeterminate, however a linear order is guaranteed from messages sent by a specific process. Messages are retrieved from the mailbox by

pattern matching on their contents, and thus are not necessarily consumed in the same order as they arrive.

A message is sent by using the bang operator whose the generic form is Pid ! Msg where Pid is the identifier of the target process. A sender needs not to wait for actual message delivery before continuing with its own computation.

A receiving processes can extract messages from its mailbox by using the selective *receive* construct whose general form is:

```
receive
   Pattern1 when Guard1 -> Expression1;
   Pattern2 when Guard2 -> Expression2;
   ...
after TimeOut
   TimeoutClause
end
```

When a process enters a receive construct, it tries to match the oldest message in the mailbox against the patterns in the body of *receive* in the order they are presented. If there is a message that matches one of the patterns and the corresponding guard is satisfied, the corresponding expression is evaluated and the message is removed from the mailbox. Otherwise, the receive construct tries with the next message. If there are no messages in the mailbox that can be retrieved in this way, the construct blocks the executing process. The *guards* and the *after* sections are optional. If the latter is specified, the receive construct will timeout after a TimeOut, expressed in milliseconds, while waiting for a matching message. In this case, the TimeOutClause will be evaluated.

The inter process communication mechanism is illustrated with the program in Figure 2.5. There are two types of processes that evaluate functions *man* and *woman*, where the last parameter for both functions are lists. We assume that the list in the function *man* contains process identifiers of all *woman* processes and vice versa. That is, we assume that the code for initialization is written somewhere else. A *man* process sends a propose message (line 2) to the first element H of the parameterized list where H is obtained by pattern matching. This process then blocks until a message *no* arrives, to which it performs the same function *man* with the tail T (line 5). A *woman* process waits for a message that matches the pattern at line 10 and binds the variable Y to the second element in the message; depending on the return value of bof, the process may repeat the procedure with a new parameter set.

Erlang further adds monitoring (one direction) and linking (two directions) capabilities to processes; a monitoring process is sent a system message if the monitored process crashes. This is very useful to build fault-tolerant systems. Some processes can be designed with a “supervisor” role, supervising another “worker” process or a group of processes: supervisor can then restart the process group when something goes wrong. In complex systems, this idea is pushed forward to structure multi-level

```
1  man(Id,[H|T]) ->
2    H ! {propose,Id},
3    receive
4      no ->
5        man(Id,T)
6    end.
```

```
1  woman(Id,P,Pref) ->
2    receive
3      {propose, Y} ->
4        case bof(Y,P,Pref) of
5          true ->
6            P ! no,
7            woman(Id,Y,Pref);
8          false ->
9            Y ! no,
10           woman(Id,P,Pref)
11       end
12   end.
```

FIGURE 2.5: An example of process communication

hierarchy of supervisors, allowing fine-grain granularity reboots of sub-systems. Methodology and patterns for structuring applications into a supervision tree are encompassed in the OTP framework.

2.3.2 OTP behaviours

OTP is a middleware and a set of libraries written in *Erlang* and shipped together with *Erlang* distribution. In fact, *Erlang* standard libraries and OTP are so intertwined nowadays that one may find it difficult to separate them out. Among other things, OTP provides an application framework for efficient development of scalable, fault-tolerant and high available systems. The framework promotes the use of programming patterns which applications may need, such as the server, state machine, error logger, and so on. Specifically, OTP behaviours formalize recurring patterns of concurrent programming. For example, a server process in a client model typically sits on a loop and reacts to requests sent from clients by performing some predefined tasks. Clients may need an explicit reply for each request. Hence, the server should be designed to handle synchronous or asynchronous requests. A behaviour gives ready-to-use implementation of these reusable, generic parts in the form of library modules, besides supporting conventions for structuring applications into supervision trees, dynamic code upgrades and deployments. Programmers have the responsibility to implement the application-specific code in a *call back* module following a conventional template of the behaviour.

In our work, we have exploited two commonly used OTP behaviours, namely: *generic server* and *state machine* that we describe below.

Generic Server

A generic server is implemented as the library module *gen_server*. This module provides specific functions for a client process to interact with the behaviour engine (i.e., the server process):

- gen_server:start_link(ServerName, M, Args, Options) is used to start a server with some optional parameters, i.e., Options. The function returns the server address. The argument ServerName is optional; if provided, a different process can interact with the created server via ServerName instead of its address. M is the name of the *callback* module.
- gen_server:call(Server, Msg) sends to Server an asynchronous message Msg and wait for a reply.
- gen_server:cast(Server, Msg) sends to Server an asynchronous message

The *start_link* function starts a *gen_server* process in the background. A client sends requests to this created server using *call* and *cast*. The way the server handles requests is defined in the module M, provided by the programmer. In OTP terms, programmers have the task to provide the implementation for *behaviour interfaces* residing in the call back module. This is considered as a contract in order to make use of the gen_server behaviour.

As an example, let us consider the *Erlang* program in Figure 2.6 that implements a factorial server. It is conventional in *Erlang* that the callback module contains both API functions (used by clients) and callback functions (used by the behaviour engine). Clients can start/stop the server, and ask for a factorial computation as well as for the number of requests the sever has received. The server state is a counter that is increased each time a request is received.

In Figure 2.6, APIs act as wrappers of the three gen_server functions described before. The behaviour interface is a set of five callback functions[2] whose signatures are exported at line 7. Their implementations start at line 23. The init function is called by the behaviour engine at the startup, and is used for initializing the server state. Requests sent by clients via functions *call* and *cast* are handled by *handle_call* and *handle_cast*, respectively. These functions return a tuple describing the new state of the server after processing the request, together with a reply to the client (in case of *call*). Any other request that does not come from the standard interfaces *call*, *cast* is directed to *handle_info*. For example, a message is sent directly by the ! operator. The *terminate* function is used to perform the necessary cleaning up when the server is about to stop.

Generic State Machine

The generic state machine (gen_statem) is a behaviour that has been recently added to OTP (available from OTP version 20). It can be thought as a mixing behaviour which combines gen_server and an event_driven state machine. In an event_driven state machine, a machine in state S that receives an event E performs a set of actions A and changes its state to S'. Similarly to other *gen_∗* behaviour, *gen_statem* can react to synchronous or asynchronous requests, it can handle info messages and hook to the structure of a supervision tree. What makes this behaviour attractive, among other things,

[2] actually, there are 8 call back functions in total while in the code we only present commonly-used ones. Those omitted are optional.

```
1  -module(fact_server).
2  -behaviour(gen_server).
3  %% API
4  -export([start_link/0, get_count/0, get_fact/1, stop/0]).
5
6  %% behaviour interfaces, to be called by the gen_server process
7  -export([init/1, handle_call/3, handle_cast/2, handle_info/2,
8           terminate/2]).
9
10 -define(SERVER, ?MODULE).
11 -record(state, {count}).
12
13 %% API functions
14 start_link() ->
15     gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
16 get_count() ->
17     gen_server:call(?SERVER, get_count).
18 get_fact(N) ->
19     gen_server:call(?SERVER, {get_fact, N}).
20 stop() ->
21     gen_server:cast(?SERVER, stop).
22
23 %% call back functions
24 init([]) ->
25     {ok, #state{count = 0}}.
26 handle_call(get_count, _From, State = #state{count = Count}) ->
27     {reply, Count, State};
28 handle_call({get_fact, N}, _From, State = #state{count = Count}) ->
29     Reply = fact(N),
30     {reply, Reply, State#state{count = Count + 1}}.
31 handle_cast(stop, State) ->
32     {stop, normal, State};
33 handle_cast(_Msg, State) ->
34     {noreply, State}.
35
36 handle_info(_Info, State) ->
37     {noreply, State}.
38 terminate(_Reason, _State) ->
39     ok.
40
41 %% internal functions
42 fact(N) -> N * fact(N-1);
43 fact(0) -> 1.
```

FIGURE 2.6: A factorial server

is the following set of features: i) postponing events. This is the ability of delaying the processing of events. In other words, if an event comes in while the machine is in a "not ready" condition to handle the event, the machine can postpone the event. A deferred event is automatically retried when the state has changed. Because of this, *gen_statem* separates two kinds of information: a state part and a data part. Changes in the data do not account for retrying postponed events. ii) generating internal events. Internal events are events from the machine to itself. By handling an internal event, the machine can perform an extra task without any stimulus from the external environment. This is useful to separate concerns in the business logic. For example, when the machine needs to perform some task based on a common condition, which is emerged after processing a number of external events.

These features can be emulated in pure *Erlang* by using a selective receive construct, but of course with vastly different levels of efforts.

2.4 The UMC model checker

UMC [81] is one of the model checkers belonging to the KandISTI [117] formal verification framework used for analyzing functional properties of concurrent systems. In UMC, a system is represented as a set of communicating UML-like state machines, each associated with an active object in the system. UMC adopts doubly-labelled transition systems (L2TS) [53] as semantic model of the system behaviour. A L2TS is essentially a directed graph in which nodes and edges are labelled with sets of predicates and of events, respectively. The model checker allows to interactively explore this graph and to verify behavioral properties specified in the state-event logic UCTL [58]. UCTL allows to express state predicates over (the labelling of) system states, event predicates over (the labelling of) single-step system evolutions, and combine these with temporal and boolean operators in the style of CTL [43] and ACTL [52].

UMC was specifically oriented towards the early analysis of (likely wrong) initial system designs, that trades the capacity of dealing with very large systems with the capacity of helping users to easily understand the source of design errors. This is achieved, among other things, by providing interactive explanations of the results of the evaluations and by allowing the user to observe and reason on systems at a high level of abstraction without being distracted, if not overwhelmed, by all the details of the specifications. In the following we briefly review the modeling language and the specification logics supported in UMC.

The modeling language

As an experimental framework for promoting the use of formal methods according to UML paradigm [98], UMC supports textual descriptions of UML state charts. A model consists of several classes and objects instantiations and of a set of abstraction rules that enable us to make observable important aspects of the model. A UML-like state machine is described in UMC in the form of a class declaration structured as follows:

```
class Name is
 Signals:
 -- asynchronous signals accepted by this class
 Vars:
 -- local variables of this object
 Transitions:
 -- transitions that determine the behaviour of the class
end Name
```

where a list of Signals summarises the set events to which an active object may react[3]. A signal denotes an asynchronous event that may trigger the transitions of an object. An object can send signals to

[3]UMC also supports an Operations section for the definition of synchronous events, which is however not relevant in our study.

itself by executing self.signal_name. The Vars section contains the private, non statically-typed, local variables of the class and optionally their initial values. Values can denote object names, boolean values, integer values or, recursively, (dynamically sized) sequences of values. The Transitions section declares a set of transition rules that describe the behaviour of the class and have the following general form:

```
source -> target {trigger [guard] / actions}
```

to denote a state transition from state source to state target. A transition is triggered by a suitable trigger event trigger (which is a signal name) and if the guard expression is satisfied, all actions inside the transition body are executed. The execution of actions may in turn change the state of the object or trigger other transitions.

To illustrate the syntax of UMC, let us look at an example of a COUNTER state machine.

```
Class COUNTER
Signals: decr
Vars x:int
Transitions:
s1 -> s2 {-/self.decr; x:=2}
s2 -> s2 {decr[x>1]/self.decr;x:=x-1}
s2 -> s3 {decr[x=1]/x:=x-1}
end COUNTER;

Objects: c1 : COUNTER
```

COUNTER has s1 as the starting state; the machine takes the first move to s2 and the set of actions to be performed includes sending a signal to itself, and initializing the local variable x to 2. While COUNTER is in s2, the signal sent from a previous step may trigger one of the two possible transitions: as long as x is greater than 1, the machine goes in a loop at state s2 where it keeps generating the internal signal decr and decreases x by 1. Otherwise it transits to the s3 state and set the value of x to 0, which forces the machine to remain in this state. Finally, for the actual creation of state machines, we need to declare *objects* of the class. This is illustrated in the last line of the code example.

UMC supports a fairly rich language to specify composite actions, for example, by using conditional and looping constructs. For more examples and details of the language we refer to the UMC website [11] and the documentation therein.

While the structure of the semantic in terms of L2TS of an UMC specification is directly defined by the system behaviour, the labels associated to nodes and edges of the graph are specified by *abstraction* rules that allow the designer to define the relevant internal aspects of the system. These rules are defined inside the Abstractions section:

```
Abstractions {
  Action: <internal event> -> <edge label>
  ...
```

$$\begin{array}{ll} \text{(event formulae)} & \chi := tt \mid e \mid \tau \mid \neg\chi \mid \chi \wedge \chi' \\ \text{(state formulae)} & \phi ::= true \mid p \mid \neg\phi \mid \phi \wedge \phi' \mid E\pi \mid A\pi \\ \text{(path formulae)} & \pi ::= \mathrm{X}_{\chi}\phi \mid \phi_{\chi}\mathrm{U}_{\chi'}\phi' \mid \phi_{\chi}\mathrm{W}_{\chi'}\phi' \end{array}$$

TABLE 2.1: UCTL syntax

```
    State:  <internal system state> -> <node label>
    ...
}
```

The possibility of obtaining a L2TS which only focuses on the aspects of the system that are considered relevant is particularly useful in many cases. For example one can visualize a compact summary of the computation trees, factorized via appropriate behavioural equivalence notions. Or she/he can model check abstract L2TS (without any knowledge of the underlying UMC), and reason on systems without a detailed knowledge of the underlying concrete implementation.

The formal model and specification logic

UMC constructs L2TSs from input UML-like specifications. A L2TS is essentially an extension of LTS where both states and transitions can be labeled with predicates.

Let AP be a finite set of atomic propositions. A L2TS is a tuple (Q, q_0, Evt, R, L) where Q is a finite set of states, $q_0 \in Q$ is the initial state, Evt is a set of events, $R \subseteq Q \times 2^{Evt} \times Q$ is a transition relation, $L : Q \rightarrow 2^{AP}$ is a labeling function which labels each state with a subset of AP.

Let e ranges over Evt, and η over 2^{Evt}. In the context of a L2TS, a path σ from a state q is either empty ($\sigma = q$) or a (possible infinite) sequence $(q_1, \eta_1, q_2), (q_2, \eta_2, q_3), \ldots$ where $q_1 = q$ and $(q_i, \eta_i, q_{i+1}) \in R \ \forall i > 0$. σ is a full path if it can not be prolonged. If σ is a path, the ith state and the ith set of events in σ are denoted by $\sigma(i)$ and $\sigma\{i\}$, respectively. Finally, we let $Path(q)$ be the set of full paths starting from state q.

UCTL is the temporal logic used by UMC tool for reasoning about UMC models. It supports the specification of action-based and state-based formulae. The syntax of UCLT is given in Table 2.1:

In the definition above, other logical operators such as ff, false, $\rightarrow$ (implication), $\vee$ (or) can be derived accordingly. Formulae in UCTL can quantify over states or paths with ϕ used to denote the first kind and π for the latter. In addition, the π formulae include also event formulae χ. The semantics of these UCTL formulae is interpreted over a L2TS model. Below, we explain the semantics of UCTL in order.

An event formula χ specifies the property of a L2TS transition labeled with some set of labels $\eta \in 2^{Evt}$. The satisfaction relation follows Table 2.2. The formula tt is always true for η, and e is true when e belongs to the set η. The event formula τ satisfies the empty set of transition labels while negation and conjunction have the usual meanings.

$\eta \models tt$	$\eta \models \tau$ iff $\eta = \emptyset$
$\eta \models e$ iff $e \in \eta$	$\eta \models \neg\chi$ iff not $\eta \models \chi$
$\eta \models \chi \wedge \chi'$ iff $\eta \models \chi$ and $\eta \models \chi'$	

Table 2.2: Event formulae semantics

For a state q, the formula *true* is always true in q. A state label p is true in q if p is in the set of labels associated to q. E and A are existential and universal operators over paths. Their semantics, together with other formulae are defined in Table 2.3.

$q \models true$	$q \models p$ iff $p \in L(q)$
$q \models \neg\phi$ iff not $q \models \phi$	$q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$
$q \models E\pi$ iff $\exists\sigma \in Path(q) : \sigma \models \pi$	$q \models A\pi$ iff $\forall\sigma \in Path(q) : \sigma \models \pi$

Table 2.3: State formulae semantics

The semantics of path formulae is defined in Table 2.4.

$\sigma \models \mathrm{X}_\chi\phi$ iff $\sigma\{1\} \models \chi$ and $\sigma(2) \models \phi$
$\sigma \models \phi_\chi U_{\chi'}\phi'$ iff $\exists j \geq 1 : \sigma\{j\} \models \chi'$ and $\sigma(j+1) \models \phi'$ and $\forall i < j : \sigma\{i\} \models \chi$ and $\sigma(i) \models \phi$.
$\sigma \models \phi_\chi W_{\chi'}\phi'$ iff $\sigma \models \phi_\chi U_{\chi'}\phi'$ or $\forall 0 < i : \sigma\{i\} \models \chi$ and $\sigma(i) \models \phi$.

Table 2.4: Path formulae semantics

For a path σ, the formula $X_\chi\phi$ satisfies for σ if ϕ holds in the next state of the path, reached by a transition where χ holds. The formula $\phi_\chi U_{\chi'}\phi'$ is true for σ whenever in the future moment there exists a state in which ϕ' holds and χ' satisfies the last transition reaching to that state; whereas ϕ holds in all the states along the path and χ holds for all the transitions in between (as illustrated in Figure 2.7).

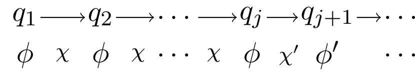

Figure 2.7: A path where $\phi_\chi U_{\chi'}\phi'$ holds

The weak until formula $\phi_\chi W_{\chi'}\phi'$ is true for a path if its corresponding until version holds, or for all states in the path ϕ holds and for all transitions along the path χ satisfies.

From the syntax of path formulae, other useful formulae can be defined. For example, the formula $true_{tt}U_{tt}\phi$ is true if ϕ holds at some future state in the considered path. It is often used to assert if the system eventually reaches an expected state. UMC provides such useful operators to provide convenience in expressing properties in UCLT logics. For example:

- $\phi_\chi U\phi'$ stands for $\phi' \vee \phi_\chi U_\chi\phi'$
- $AF\phi$ stands for $A(true_{tt}U\phi)$
- $EF\phi$ stands for $E(true_{tt}U\phi)$

- EF_γ stands for $E(true_{tt}U_\gamma true)$
- $AG\phi$ stands for $\neg EF\neg\phi$
- $FINAL$ is a formula which is always true in a final state (i.e., without any outgoing transitions)

In [46], we have manually modeled a simple instance of the stable marriage with attributes and verified some simple properties using UMC. Our experience suggests that encoding a concurrent system using the UMC modeling language is a rather tedious task. Every detail of a system under consideration must be worked out so that a user could inline each transition she/he has in mind into a UMC transition. It is desirable to obtain UMC models as the result of translations from higher level description languages, as also noted in [98]. This is crucial to proper model complex systems relying on complex interactions such as AbC. In Chapter 4, we propose an approach for translating AbC into UMC to bridge this gap.

Chapter 3

ABEL - an Erlang implementation of AbC

In this chapter, we present an implementation of *AbC* in *Erlang*, which we call *ABEL*. *ABEL* is a domain-specific framework with support of an API that mimics *AbC* constructs for the purpose of developing and experimenting with systems whose elements interact according to the *AbC*'s style. *ABEL* employs two coordinators: the intra-coordinator, for coordinating co-located processes within one single component, and the inter-coordinator, for coordinating the activities among different components. The intra-coordinator or simply the coordinator takes charge of implementing the interleaving operator (|) of *AbC* processes. The inter-coordinator to which we also refer as coordination infrastructure implements the parallel operator (||) of *AbC*.

The design and implementation of *ABEL* aims at an efficient executor for *AbC*. Given an *AbC* specification, programmers and modelers can derive an *ABEL* program from the corresponding *AbC* specification. Such a task as we shall see, is quite natural. To gain confidence on the actual behaviour of the program, the same specification can be verified by using the method presented in Chapter 4.

The structure of this chapter is the following. We start by explaining how basic concepts of the *AbC* paradigm are supported in *ABEL* and provide a short description of *ABEL*'s API. Like in Chapter 2, we use examples and the previously introduced case studies to illustrate the programming model. We then present the implementation at the system level and at the component level. For the former, we explain the total order broadcast protocol, defined in [20] on which we have based the *ABEL* coordination infrastructure. For the latter, some technical challenges are discussed and an implementation is presented in details. Finally, we show how the case studies can be naturally programmed in *ABEL* starting from their *AbC* specifications and evaluate *ABEL*'s performance through a number of experiments.

3.1 ABEL's support for AbC

An *ABEL* program is based on a number of component definitions. A component definition in turn includes the definitions of its individual processes, an attribute environment and an interaction interface. In *ABEL*, a component is started in two steps: (i) it is created and (ii) it gets assigned a behaviour.

Creating a component. To create a new component, an attribute environment Env and an interface I are provided to the function *new_component(Env,I)*. new_component returns a unique address C. Internally, the function creates a coordinator process that maintains the component environment. It is also responsible for connecting this Comp to the messaging infrastructure. More on this will be elaborated later.

Environment of a component is represented as an *Erlang* map whose keys are atoms denoting attribute names. Attribute values can be any *Erlang* terms. Interface is a tuple of attributes names exposed for interaction. For example, $Env = \{id => 1, pref => [3,4], partner => -1\}$ declares an environment with three attributes id, pref and partner together with their initial values. $I = \{id\}$ declares that id is the exposed attribute. Accordingly, while components may have many attributes they only interact considering those attributes whose names appear in their interfaces.

To start the execution of a component whose address is C, *ABEL* provides the function start_beh with the following syntax: $start_beh(Comp, [BRef])$ where $[BRef]$ is a list of functions describing component processes. Intuitively, this means that the component behaviour consists of several parallel processes, each with the behaviour described by a function in the provided list. Components execute concurrently and interact with each other according to the combined behaviour of components processes.

$$\begin{aligned} \mathsf{C} = \ & \textbf{new_component}(Env, I), \\ & \textbf{start_beh}(\mathsf{C}, [BRef]) \end{aligned}$$

FIGURE 3.1: Starting a component

Behaviour Definition. The behaviour of a process is defined in terms of *Erlang* functions with a restricted structure in their bodies. In essence, process definitions rely heavily on a set of *ABEL* library functions mimicking *AbC* constructs. Since *ABEL* API is written in *Erlang*, *Erlang* code can also co-locate in an *ABEL* program. However, it is mainly used for the purpose of defining additional helper functions, for example, for building the definitions of complex operations like *op* and *p* operators (Chapter 2). To make our intention clear and to enforce writing process definitions in *AbC*'s style, we report the BNF syntax for *ABEL* processes in Figure 3.2. There, elements wrapped by $\langle\rangle$ are optional and $[Elem]$ is a list of elements of type *Elem*. Furthermore, we use g, m, s, r, x, u to denote

an awareness predicate, message, sending, receiving predicates, input-binding variables and attribute updates, respectively.

A behaviour definition *BDef* is an *Erlang* function *beh_name* with two parameters where the first denoting the component address C, and the second a list of bound variables V (initially empty and gradually updated by *ABEL*). The body of the behaviour definition is a single command that can take several forms: prefix, choice, parallel or a behaviour call. The reference *BRef* of a given behaviour is obtained by means of *Erlang* anonymous function. *BRef* is passed to a command so that the process, after finishing the command can proceed with this referred behaviour. This way of programming is a sort of continuation-passing style.

In Figure 3.2, the component address C and the list of bound variables V have to be passed to *every* command as their parameters.

$$
\begin{array}{rll}
BDef ::= & beh_name(\mathsf{C}, V) \to Com. & \\
BRef ::= & \mathbf{fun}(_V) \to beh_name(\mathsf{C}, _V)\ \mathbf{end} & \\
 & \mathbf{nil} & \\
Act ::= & \{\langle g \rangle, m, s, \langle u \rangle\} & \text{Output} \\
 & \{\langle g \rangle, r, x, \langle u \rangle\} & \text{Input} \\
Com ::= & \mathbf{prefix}(\mathsf{C}, V, Act, BRef) & \textbf{Send} \\
 & \mathbf{choice}(\mathsf{C}, V, [\{Act, BRef\}]) & \textbf{Choice} \\
 & \mathbf{parallel}(\mathsf{C}, V, [BRef]) & \textbf{Parallel} \\
 & \mathbf{call}(\mathsf{C}, V, BRef) & \textbf{Call}
\end{array}
$$

FIGURE 3.2: ABEL API for process definitions.

Before explaining the commands and their operations, we first describe how the basic *AbC* terms are supported in *ABEL*.

Messages and Variables. Messages that are exchanged among components are represented as *Erlang* tuples. From a sending component point of view, a message element m can be either an *Erlang* term or a function parameterized with an environment $fun(E) \to \ldots end$. The last form is introduced for making it possible to refer the value of attributes in a message. For example, $\{`done', fun(E) \to att(colour, E)\ end, fun(E) \to att(round, E) + 1\ end\}$ is a message consisting of three elements with the second referring to the value of attribute color and the third referring to the value of attribute round plus 1. In *AbC* terms, this is the same as the message $('done', this.color, this.round + 1)$.

From a receiver point of view, a received message is a tuple whose elements are bound to some variables. The input binding variables x that come together with an input action are represented as a tuple, e.g, $\{x, y, z\}$. These variable names can be used to access the element of the messages received.

Interaction Predicates. Interaction predicates including sending and receiving predicates are represented as binary and ternary functions, respectively. A sending predicate s is a function parameterized with the sender and receiver environments in that order. A receiving predicate r is a function parameterized with both environments of the receiver and the sender, and the communicated message in that order. In the bodies of these predicates, the value of attribute *attr* in an environment E is referred by using the *ABEL* function $att(attr, E)$. The message content can be referred by using the *Erlang* built-in function *element(nth, M)*. Sequential *Erlang* is exploited to specify the code of predicate. For convenience, we may specify the parameter names in predicates using L (local environment), R (remote environment), M (message).

For example, given a set *nbr*, a sending predicate $this.id \in nbr$ in *AbC* can be represented in *ABEL* as the function:

```
fun(L,R) → sets:is_element(att(id,L),att(nbr,R)) end
```

where L, R are environments of sender and receiver, respectively. L is supplied first, meaning that it is also the environment of the executing component. In the function body, attribute values are referred accordingly based on these names. *Erlang* sets is used to specify the expected membership relation.

A receiving predicate ($x =$ '*done*'$\wedge z = this.round$) that conditions on an incoming message containing three elements (x, y, z) is written in *ABEL* as:

```
fun(L,_,M) → element(1,M) == done andalso element(3,M) == att(round,R) end
```

which needs three parameters. This predicate constraints only on the message and the local environment, thus the second parameter may be written as _. In the example, logical operators are of *Erlang*; message elements are obtained by the built-in function *element*.

Awareness and Update. We distinguish between awareness predicates and the interaction predicates mentioned above. An awareness predicate is local to the executing component, and thus is represented as a unary function, i.e., $fun(L) \ \rightarrow \ \ldots \ end.$

Attribute update is represented as a list of pairs; in each pair, the first element is an attribute name and the second denotes the value to be updated. The second element is a function parameterized with the local environment for the possibility of accessing attributes, i.e., $fun(L) \ \rightarrow \ \ldots \ end.$ This function, in case the update operation is associated with an input action, is parameterized also with the communicated message, i.e, $fun(L, M) \ \rightarrow \ \ldots \ end.$

Communication Actions. In *AbC*, the awareness predicate and attribute update, if specified at all, are performed atomically together with the associated communication actions. Thus, in *ABEL*, these operations are bundled into send and receive commands. An output action has the form of $\{\langle g\rangle, m, s, \langle u\rangle\}$. An input action has the form of $\{\langle g\rangle, r, x, \langle u\rangle\}$.

For example, the following code snippet

```
M = {'propose', fun(L) → att(id,L) end},
SP =  fun(L,R) → hd(att(pref,L)) == att(id,R) end,
U = [{partner, fun(L) → hd(att(pref,L)) end}, {pref, fun(L) → tl(att(pref,L)) end}],
Act = {M,SP,U}
```

corresponds to the AbC output action

$$(\text{'}propose\text{'}, \text{this.id})@(\text{id} = \text{hd}(\text{this.pref})).[\text{partner} := \text{hd}(\text{pref}), \text{pref} := \text{tl}(\text{pref})]$$

whereas the following

```
RP = fun(_, _, M) → element(1,M) == 'no' end,
U = [{partner, −1}],
Act = {RP,{x},U}
```

corresponds to the AbC input action $(x = \text{'}no\text{'})(x).[partner := -1]$.

A command Com has parameters C, V bounded by those of an outer function, and a third parameter specifying basic actions possibly paired with references, depending on the command type. It is worth mentioning that C and V are names that simply act as place holders in commands and processes definitions. *ABEL* supports the following commands.

Prefix Command. takes as parameter an action *Act* and a continuation $BRef$. *Act* can be either input or output action and its description is a tuple as shown in Figure 3.2. This command executes *Act* and then the behaviour encapsulated in $BRef$. The execution of an input action (if successful) returns a message; then continues by calling $BRef$ on an updated list of bound variables, calculated by appending the message to the current list V. If *Act* is an output action, the continuation is determined by applying $BRef$ to V.

As an example of prefixing process, the following process definition in AbC (Example 1.2, Section 2.1)

$$P_M \triangleq \ (\text{'}proposal\text{'}, \text{this.id})@(\text{id} = \text{hd}(\text{this.pref})).[\text{partner} := \text{hd}(\text{pref}), \text{pref} := \text{tl}(\text{pref})]P_{M1}$$

can be written in *ABEL* as a behaviour definition

```
pm(C,V) →
   M = {'propose', fun(L) → att(id,L) end},
   SP = fun(L,R) → hd(att(pref,L)) == att(id, R) end,
   U = [{partner, fun(L) → hd(att(pref,L)) end}, {pref, fun(L) → tl(att(pref,L)) end}]),
   prefix(C, V, {{M,SP,U}, fun(_V) → pm1(C,_V) end}).
```

Choice Command. takes as parameter a list of pairs. Each pair provides a description *Act* of the prefixing action and a continuation behaviour $BRef$. This means that each sub-process in a choice must be started with a prefixing action. This command executes one of the actions and continues with

the associated behaviour. An example of choice is given in Figure 3.3. *ABEL* currently does not allow a choice that mixes sending and receiving processes. This representation simplifies our implementation in which a coordinator can handles all the branches of the choice in a consistent manner; that eventually select one possible action among others to execute. Moreover, such a mixing behaviour, if desirable at all, can be encoded using parallel processes with appropriate awareness predicates.

Parallel Command. The **parallel** command, when executed creates dynamically processes whose definitions are supplied in the parameter list. It thus behaves in a similar way to *start_beh*. Note that a prefixing action is needed in case one wants to model replication via recursion. A common use case of replication is at modeling the behaviour that upon receiving a message, process the message according to a prescribed manner and at the same time being available to deal with another message. To express this type of behaviour in AbC, we write $X \triangleq a.(P \mid X)$. That is, a parallel construct with a recursive call in it must be prefixed by an action.

Consider the process P_W in Example 1.2 (Chapter 2). It is tempted to specify:

$$P_W \triangleq (x = \text{‘}propose\text{’})(x, y).H_W|P_W$$

The above code when translated in *ABEL* looks like the following:

```
p(C,V) →
    parallel(C,V,[fun(_V) → q(C,_V) end, fun(_V) → p(C,_V) end]).
q(C,V) →
    prefix(C,V,{fun(_, _, M) → element(1,M) == ‘propose’ end, {x,y}, fun(_V) → h(C,_V) end}).
```

which spawns an infinite number of processes executing the function p. This use is invalid and must be avoided. On the other hand, if we specify P_W as:

$$P_W \triangleq (x = \text{‘}propose\text{’})(x, y).(H_W|P_W)$$

In *ABEL*, this is equivalent to the code snippet

```
p(C,V) →
    prefix(C, V, {fun(_, _, M) → element(1,M) == ‘propose’ end, {x,y}, fun(_V) → q(C,_V) end}).
q(C,V) →
    parallel(C,V,[fun(_V) → h(C,_V) end, fun(_V) → p(C,_V) end]).
```

which spawns a new process only when the prefixing action is consumed. Thus the number of newly created processes is equal to the number of ‘propose’ messages received.

Call Command. executes the behaviour referenced by $BRef$ by applying $BRef$ to V.

To conclude this section, we show the full definitions (Figure 3.3) of processes for components in the classical stable marriage problem (Chapter 2). The verbosity of the program is caused by the

```
%% The definition of men

pm(C,V) →
    M = {'propose', fun(L) → att(id,L) end},
    SP = fun(L,R) → hd(att(pref,L)) == att(id, R) end,
    U = [{partner, fun(L) → hd(att(pref,L)) end}, {pref, fun(L) → tl(att(pref,L)) end}]),
    prefix(C, V, {{M,SP,U}, fun(_V) → pm1(C,_V) end}).


pm1(C, V) →
    RP = fun(_, _, M) → element(1,M) == 'no' end,
    U = [{partner, −1}],
    prefix(C, V, {RP,{x},U}, fun(_V) → pm(C,_V) end}).

%% The definition of women
pw(C,V) →
    RP = fun(_, _, M) → element(1,M) == 'propose' end,
    prefix(C,V,{RP,{x,y},fun(_V) → pw1(C,_V) end}).

pw1(C,V) →
    parallel(C,V,[fun(_V) → h(C,_V) end, fun(_V) → pw(C,_V) end]).

h(C,V) →
    G1 = fun(L) → bof(var(y,V), att(partner,L), att(pref,L)) end,
    SP1 = fun(_,R) → att(id, R) == var(y,V)  end,
    G2 = fun(L) →  not bof(var(y,V), att(partner,L), att(pref,L)) end,
    SP2 = fun(L,R) → att(id, R) == att(partner,L) end,
    M  = {'no'},
    U =[{partner, var(y,Y)}],
    A1 = {G1,M,SP1},
    A2 = {G2,M,SP2,U},
    choice(C,V,[{A1, nil},{A2, nil}]).
```

FIGURE 3.3: The derived code for SMP specification

current presentation of the basic elements such as predicates, messages, ... that can be addressed by automatic translation. However, when looking at the uses of *ABEL* commands (highlight), we can notice a one-to-one structural correspond between the *AbC* specification and the derived program.

3.2 ABEL Implementation

This section shows how an implementation can deal with the operators parallel (‖) and interleaving (|) of *AbC*. For the former, we are concerned with achieving broadcast communication among components while ensuring its instantly synchronous semantics. For the later, we are concerned with picking representative actions among processes within a component while taking into account the affect of component environment. Accordingly, we present two coordinators, one for coordinating components in a system and the other for coordinating processes in a component.

3.2.1 Coordinating components

First, we will discuss the design of the inter-coordinator implementing the parallel operator ($\|$) among different components. The actual semantics of $\|$ is given in Chapter 2. Informally, when a component sends a message, the message is delivered to all other components in a single move. They are the components where the message is delivered that decide whether to actually receive the message or to discard it, by checking on both the sending and receiving predicates. In the message passing model of AbC, output actions are non blocking in that they can take place even if there is no component willing to receive it. Input actions, instead, wait to synchronize on the available messages in order to be performed. This implies that message passing happens in synchronous rounds of sending actions. At each round, only one component is scheduled to send its message. A new round starts when the message in the previous round is delivered to all components in the system. In other words, message delivery at AbC components is performed according to a total order.

In fact, AbC abstracts away an underlying messaging infrastructure and leaves this issue to the implementation. In [20], the authors proposed a coordination infrastructure to mediate messages among AbC components. The infrastructure also plays the role of a fixed sequencer [54] to guarantee that different messages are delivered in the same order to AbC components. The main idea can be summarized as follows. All components joining the system are connected to a common infrastructure. The internal structures of components at this level are abstracted, i.e., at a given moment, any component only offers either a send or a receive action. Each component maintains a local counter c, initially set to 1 and a message set, initially set to $\emptyset$. When a component is willing to send a message, it first asks for a fresh id from the infrastructure. If $c = id$, the component can actually send a message, labeled with id to the infrastructure. The latter broadcasts this data message to all other components except the sender. During this operation, the local counter of the sending component is increased by 1. Data messages forwarded to a component are stored in its message set. A component delivers a message in the set if its local counter c equals to the id of the message. By doing so, the local counter of the receiving component is also increased by 1. The approach is illustrated in Figure 3.4.

Activities performed by participants are presented in the form of a set of rules (started with $\Box$) whose guards, at any time hold will cause the execution of the associated actions. The left of Figure 3.4 shows the behaviour of a connected component. The extra variable mid holds the value of fresh id (line 14), and is reset each time the component has sent a data message (line 19). It is noted that while a component is waiting for sending a message after getting a fresh id, other messages might be dispatched to it (line 9) and possibly delivered (line 22). The right of Figure 3.4 sketches the behaviour of the infrastructure. On receiving a request message for a fresh id, the infrastructure communicates to the sender the value of its internal counter (line 8) and increases this value (line 9). On receiving a data message, the infrastructure buffers the message into a message set (line 12), and only broadcasts the message to all other components (line 15) if it has the expected id (line 14).

```
1  Component comp_i
2    Initialization:
3      c_i := 1
4      msgset_i := ∅
5      mid := -1
6      inf := the address of the infrastructure
7    Behavior:
8      □ receive(id',m') →
9        msgset_i := msgset_i ∪ {(id', m')}
10
11     □ (mid = -1) and request-to-send →
12       send(comp_i,req) to inf
13       receive(id) →
14         mid := id
15
16     □ (mid = c_i) and (comp_i is able to send m) →
17       send(comp_i,(mid,m)) to inf
18       c_i := c_i + 1
19       mid := -1
20
21     □ ∃(id, m) ∈ msgset_i : id = c_i →
22       deliver(m)
23       msgset_i := msgset_i \ {(id, m)}
24       c_i := c_i + 1
```

```
1  Infrastructre
2    Initialization:
3      c := 1
4      msgset := ∅
5      all := the addresses of all components
6    Behavior:
7      □ receive(comp_i,req) →
8        send(c) to comp_i
9        c := c + 1
10
11     □ receive(comp_i,(id,m)) →
12       msgset := msgset ∪ {(id, m, comp_i)}
13
14     □ ∃(id, m, comp_i) ∈ msgset : id = c →
15       send(id,m) to all \ comp_i
16       msgset := msgset \ {(id, m, comp_i)}
```

FIGURE 3.4: The fixed sequencer variant for building the total order

To increase efficiency, the messaging infrastructure may be composed out of a set of logically structured nodes. Out of these nodes, only one is assigned the role of sequencer to guarantee generation of unique message ids. Nodes in the infrastructure collaborate in forwarding requests and data messages. Three different structures of the infrastructure have been considered in [20], namely cluster (nodes are structured as a complete graph, with a distinct counter node), ring (nodes are connected in a ring topology, with a distinct counter node) and tree (nodes are organized as a tree where the root is also the counter node). In the same paper, the behaviour of each infrastructure is decribed via an operational semantics and it is proved that messages are delivered according to the expected total ordering.

Although our presentation of total order broadcast in Figure 3.4 is slightly different with that in [20], we believe that its overall is the same and that the delivering ordering is maintained. This claim is based on the observation that the local counters of each infrastructure component eventually converge to the same value. Clearly, a sent message causes the sending component to increase its counter by 1 and delivering this message at other components also leads to the increase by 1 of their counters. Furthermore, a component cannot send a message labeled with a fresh id until all the messages with $id' < id$ have been delivered to it. In fact, the local counters of components and of infrastructure will eventually converge to the same value; the delivery order of messages is the increasing order of messages ids.

In the above protocol, while waiting for sending a message, a component can deliver some of messages forwarded to it, and may consume them. At line 16 on the left of Figure 3.4, an actual send can only happen if at the moment the condition on local counter is satisfied, the component is still be able to send a message. Since the component state may changed due to consuming a message, there

may be the case that the actual send is not the same as the one which initiated the request for fresh id (line 11). Moreover, the changes in one component environment could disable a send. This in principle can make the overall system blocked. A critical reader may argue that once a component has "decided to send a message" (i.e., when the component requests a fresh id at line 11), it must send out the message under any circumstances. For example, assume a component C that decides to send a 'umbrella' when a condition ($raining = \text{tt}$) is true. Then the component should not withdraw this decision, even if in the meanwhile condition ($raining = \text{tt}$) has become false due to an attribute update caused by another parallel process. However, this reasoning is only applicable when we consider the component in isolation. When taking into account other parallel components, a relative schedule over the components has to be imposed. If we compose C with a Weather

$$\overbrace{\Gamma_1 : \langle \text{raining} = tt\rangle(\text{`}umbrella\text{'})@\Pi.P_1 \mid (x = \text{`}sunny\text{'})(x).[\text{raining} := \text{ff}]P_2 \mid \ldots}^{C} \parallel \overbrace{\Gamma_2 : (\text{`}sunny\text{'})@\Pi'.Q}^{Weather}$$

and assume $\Gamma_1(raining) = tt$, thus C is allowed to send a "umbrella". In addition, C may have another process that upon receiving a 'sunny', would update attribute $raining$ to ff. Now, notice that Weather can also send a 'sunny' as implied from its process structure. According to AbC's semantics of $\parallel$, either C or $Weather$ can take place first (but not both at the same time). If Weather takes the first move which sends out 'sunny', C can not send a 'umbrella' afterward. This is because 'sunny' is delivered to all components in the same move, in which C has consumed it.

In the context of the sequencer-based protocol, the above situation falls into the case that Weather gets a fresh id of 1 and C gets that of 2 from the infrastructure. It follows that both Weather and C deliver the message 'sunny' labeled with id 1. This leads to the output action of C, previously available, becomes disabled. This means C does not make use of his turn (i.e., fresh id) and thus indirectly block other components who are waiting for a message to be delivered (imagine the above example system is composed with more components in the right).

Therefore, the moment an infrastructure component requesting a fresh id should be interpreted as the component is asking to be scheduled, not as the component has decided to send a message. In other words, a component decides on an actual send only when the fresh id equals to its local counter. Nevertheless, this problem can be overcome by letting the components send out empty messages in case the second clause of the condition at line 16 could not be satisfied. On the other hand, we think that it could also be avoided at the specification level with careful design of processes behaviour. Note that this issue was not considered in [20].

A tree-based coordinator

In [20], it was shown that the tree infrastructure offers better performances under different scenarios. Because of this, we have based our implementation of *ABEL* on the tree infrastructure and describe

it below.

In a tree topology, each node knows the addresses of its parent and its children at the initialization time. As AbC components are anonymous, the implementation relies on a *registration node* which is globally named. AbC components contact the registration node which in turn assigns them to one of the tree nodes. More precisely, a connected component and its assigned node exchange their addresses at this point for later communications.

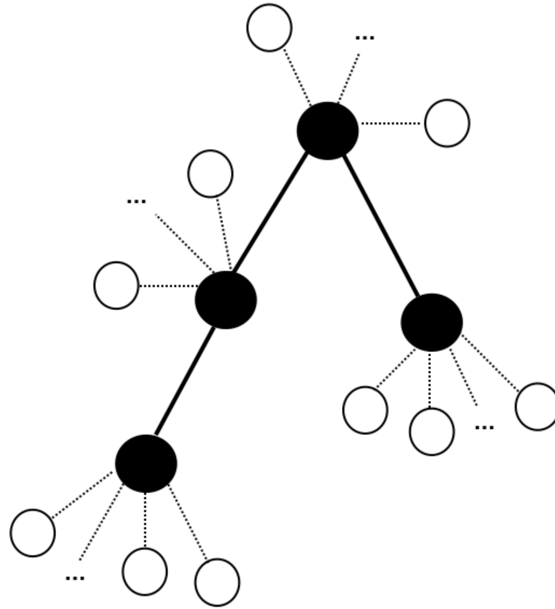

FIGURE 3.5: Binary tree-based infrastructure

Figure 3.5 shows an example of a tree-based infrastructure. The black nodes belong to the tree while the white nodes are connected AbC components. A node can only communicate with those connected to it. Similarly, an AbC component interacts with the node to which it has been assigned. Tree nodes collaborate on dispatching messages sent from connected AbC components, which can be either requests for fresh ids or actual messages. Since, in the tree, each node is responsible for a group of components some efficiency can be gained by using more nodes. However, the root of tree, that plays the role of a single sequencer, is a potential bottleneck.

The message flow in the infrastructure when a component performing different sending requests is illustrated in Figure 3.6. When a non-root tree node receives a request message for a fresh id the message is forwarded up to its parent. The procedure is repeated in this way until the message reaches the root. The root maintains a counter and increases it every time it handles a request message. The fresh id is forwarded from the root along the same path of its request, but in a reverse order. Eventually, the tree node which initiated the request receives the fresh id and sends it to the requesting component. On the other hand, when a tree node receives a data message, it forwards the messages to the other connected nodes and to connected AbC components, except the sender.

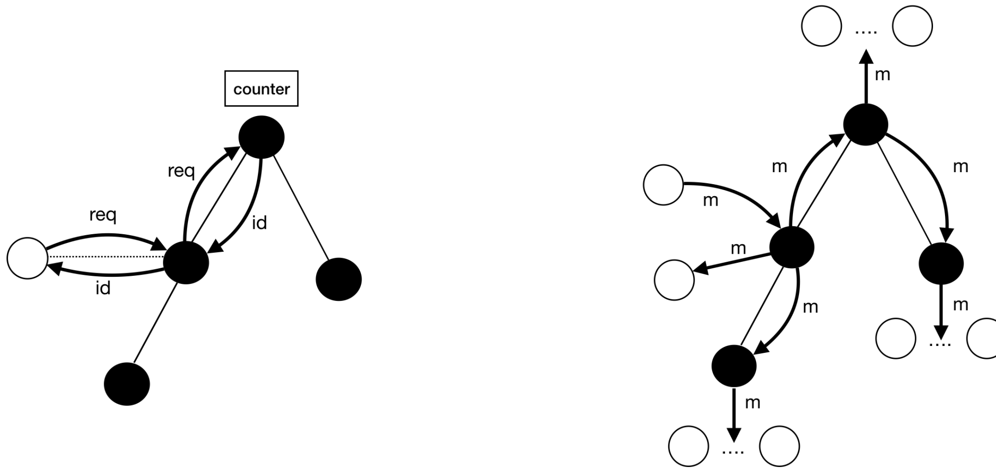

FIGURE 3.6: Message flows when a component requests a fresh id (left) and send a message m (right)

The implementation of a tree-based structure follows the behaviour described in the right of Figure 3.4 (in fact, what presented there is for the tree consisting of one node, which is also the root). Accordingly, every node has its own message set and a local counter. A message in the set is delivered at a node if it has the id equals to the local counter. The root in addition uses a dedicated counter for allocating fresh ids. The address list kept by nodes contain their children, parent and *AbC* components. Therefore, tree nodes can deal with data messages forwarded in the same manner. To handle fresh id requests, any node, when forwarding a request (originated from some *AbC* component) to its parent, also attaches a list of nodes that the request has traveled so far, including the current node. This list is then used by nodes when forward a reply (originated from the root), which is shorten each time it passes by one node. A node replies the fresh id to a *AbC* component if it is the last node in the chain. Moreover, note that messages can be exchanged asynchronously between tree nodes.

Erlang gen_server facilitates quick prototyping of the tree: the behaviour of nodes are very similar in which only some extra pattern matching on list is needed to distinguished between leaves and root nodes. As a result, the call back module was written with only 150 lines of *Erlang* code excluding comments. In our implementation, the tree is started by supplying the number of nodes as the main parameter. Nodes are then spawned concurrently by a supervisor that keeps track of their addresses. After this, the supervisor sends a request to form a tree to all nodes. Essentially, this is done by sending to each node the addresses of its parent and children. In addition, a globally named registration node is also started and informed of the address list of tree nodes. The registration node will use this list for assigning components to nodes. Currently, we assume a binary tree topology and that the registration node assigns components to nodes in a round robin fashion.

3.2.2 Coordinating processes

An *AbC* component is an autonomous entity with multiple threads of control sharing the same attribute environment or component state. The behaviour of a component is the one of its processes. As these

processes operate independently, we need appropriate strategies to coordinate their activities in order to guarantee the interleaving semantics (|) of AbC. (i) If more than one processes offers an output actions, then *only one* of them can take place. (ii) If there is more than one process that can actually receive a message, again only only one of them can use that message. (iii) A component may discard a message only if *all* of its processes discard that message. (iv) The presence of an *awareness* predicate may affect the behaviour of the executing component in a subtle way, for example, by disabling some behaviour which was previously available (or the other way around).

These considerations imply that an executor at any moment should know at least the following information, in order to determine the actions of a component:

- the number of processes in the component: this number is dynamic as it may increase due to an activation of new process in presence of parallel operator (|) in recursive definitions. It can also decrease when terminated processes are garbage collected. Examples for these are the process P_W in the stable marriage problem and the process A in the graph coloring problem.
- the communication status of each process: at a given time, an active process can either perform just an input or an output action, regardless of the process definition.
- the attribute environment: AbC actions involve attribute values in their awareness and interaction predicates, and may have immediate side effects on the environment via attribute updates.

For this reason, our view about the attribute environment is that it should not be considered as a static data store, but a reactive process which acts according to its current state and the incoming events. A naive implementation is more likely to follow the shared-memory model, in which parallel threads rely on locks for atomic access to a shared environment. While updates from different processes on the shared memory may be serialized this way, it is difficult to manage implementations as the number of processes grows. Our implementation instead treats the attribute environment as a separate process besides AbC processes. In order to deal with requirements of the (|) operator, this process is designed as a state machine and plays the role of a coordinator for AbC processes. Figure 3.7 pictures the internal structure of an AbC component in $ABEL$. The component consists of a number of AbC processes that are represented by *Erlang* processes, and a coordinator implementing *gen_statem* behaviour. Component processes execute the given process definitions as explained in Section 3.1 while the coordinator keeps the attribute environment as its state. The processes communicate *synchronously* with the coordinator by message passing via commands (expressed in API). Each process commits one command at a time, and continues only after receiving an acknowledgment message.

For each command received, the coordinator determines whether or not the command can be executed. If so, it sends back an acknowledgement message in order to unblock the requesting process. Otherwise, the relevant information about the command is stored for later try. In fact, it is the coordinator that does most f the work, component processes are state-less and simply propose actions for execution.

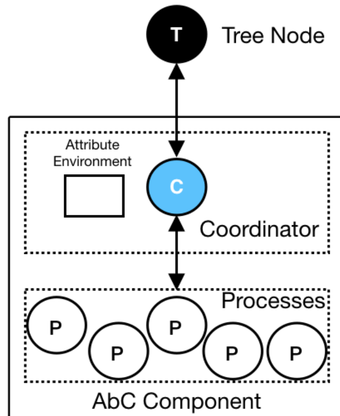

FIGURE 3.7: AbC component in ABEL

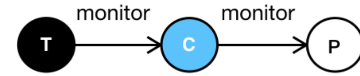

FIGURE 3.8: Process Monitoring

The relative timing of actions is left to the runtime system. This approach may sound strange in that processes almost do not perform any computations, yet given the complex behaviour of *AbC* component and discussions raised, we think it is the suitable way to implement an interleaving semantics.

A state-machine based coordinator

A coordinator is created for a component by a call to $new_component(Env, I)$. The environment Env becomes the machine state, the interface I is part of its data. At startup, the coordinator registers itself to the infrastructure, on the registration node. The component and an assigned tree node exchange their addresses. The component is monitored by the tree node (Figure 3.8) that will trigger removal of its address from the infrastructure when it terminates. In general, the coordinator, inheriting the state machine behaviour receives messages as events that are handled according to their types and contents. The set of events coming in includes those sent from *AbC* processes, from the infrastructure and other implicit kinds.

Messages from *AbC* processes are external events that are recognized and treated according to the *ABEL* commands issuing them. *ABEL* API are in fact wrappers of functions that, when invoked, prepare appropriate messages and send them to the coordinator. In our case, the machine designates an event handler for each command: send, receive, choice and parallel. As already mentioned, a choice operator cannot mix send and receive actions. Thus, there are two kinds of choices: among sending processes and among receiving processes. This means, apart from parallel events, the *communication status* of processes is made clear by the command they execute.

The coordinator records the communication status of a process each time it receives an event from the process. The communication status is exploited for message delivery. Basically, processes in a status of sending are not subjected to receive any message; processes in a status of receiving, instead can consume a message and thus their status contain also tuples describing their input actions, i.e., awareness predicates, receiving predicates and update descriptors. In fact, the status of a process may

be augmented with more information to deal with specific uses of the API. All this are kept in the data part of the machine, for instance by using a map where keys are process identifiers (pids) and values are the corresponding status for each type of action. After successfully executing an action, the record associated with pid of the corresponding process is removed.

In addition, the coordinator handles external events sent from the infrastructure, i.e., data messages originating from other components. For this purpose, the coordinator takes charge in the *deliver* procedure which are similar to the one in Figure 3.4. It keeps a local counter whose value increases each time a message is sent or delivered, and a message set for storing the forwarded messages. However, message delivery can only be activated if the following two conditions hold: there is a message in the set whose id equals to the local counter and the communication status of all processes have been collected. The first requirement may be met each time a data message is forwarded. The second may happen when a process has terminated, or when a process has communicated its status. In other words, message delivery can be triggered in different event handlers.

In the following, we describe how a coordinator reacts to the different kinds of events it may receive.

Parallel events. These events are generated by calls to *start_beh* or *parallel*, to which, the coordinator processes them in the same event handler. An *Erlang* process is spawned and monitored for each behaviour definition in the list. In addition, the total number of processes is updated. Thus the commands *parallel* and *start_beh* are essentially the same. Monitoring *AbC* processes (Figure 3.8) allows the coordinator to detect their terminations and thus to maintain a consistent view on the total number of processes.

Data Events. These are the data messages sent from a tree node. On receiving such an event, the coordinator stores the message in the message set. It can generate an internal event for message delivery if all processes have communicated their status.

Sending events. On receiving a sending action from a process, the coordinator checks the awareness predicate preceding this action against the attribute environment. If the guard is not satisfied, the event is postponed for handling later in a future state different from the current one. Otherwise, the coordinator proceeds with a send procedure similar to the one in Figure 3.4: if a fresh id is not available, the coordinator requests for a new id from the tree node it connects to. If the fresh id equals to the local message counter, the message is labeled with the fresh id and forwarded to the tree node. Actually, the message to be forwarded is more verbose: it includes the portion of the environment Env constrained by interface I, the partial evaluation of the sending predicate, a possibly new message where (some of) elements in the old one have been evaluated to concrete values. This is the consequence of the broadcast semantics of output action. After the message is being forwarded, the coordinator updates the environment according to the update descriptor attached in the event. The sending action is considered to be success and the local counter is increased by 1. The coordinator also sends an acknowledgement to the process issuing the event. On the other hand, if the fresh *id*

is *not* equal to the local counter, the sending event is postponed. Accordingly, postponed events are retried if there are changes in the attribute environment or if the local counter equals to the fresh *id*.

In case of the sending action can not be executed, the communication status of the corresponding process is marked as discarding, meaning that this process can not accept any message. If the status of all processes have been collected and if the message set is not empty, an internal event may be generated for message delivery.

Receiving events. The coordinator records the communication status of the requesting process as receive. All the related information including awareness predicate, receiving predicate and update descriptor are stored. Unlike sending events, the coordinator never postponed receiving events; they simply wait for message delivery. The handler can fire an internal event for dispatching message if the conditions meet. In the internal transition that handles message delivery, any receiving process that "wins" a message, it is forwarded the message and the communication status of the process is removed.

Choice events. Since mixing choice is not allowed, choice events are classified according to their types: either input or output. The coordinator treats a choice process as a single process and the list of actions it provides as a single action. The coordinator handles a choice event in a very similar way to normal send and receive events.

In case of a choice among output actions, the coordinator inspects the list of actions from left to right. The first available action (i.e., whose awareness predicate is satisfied and the fresh id is equal to the local counter) is executed. The choice process is acknowledged with the continuation corresponding to the selected branch. If there is no actions in the list can be enabled, the event is postponed.

Choice among input actions is the same as handling for a normal receiving events. If executed, the acknowledgement message to the choice process in this case contains both the received message and the continuation.

Other events. Other events include implicit events caused by info messages generated when the monitored processes terminate and internal events generated by the machine. For termination events, the coordinator removes the process status and accordingly updates the total number of processes. These events, as mentioned may trigger a message delivery.

Finally, the coordinator itself might generate internal events while processing other events to dispatch data messages. It has an event handler for this event kind where a deliver procedure is exercised: the message with the smallest *id* is extracted and removed from the message set. If its *id* equals to the local counter, the message is checked against the communication status of receive actions. For each action, three checks, corresponding to three kinds of predicates, i.e., awareness, sending and receiving predicates are needed. The first receive action that passes these checks will have its update descriptor applied to the current state of the coordinator and have the message delivered to its corresponding

process. If there is no receive action can consume the message in this way, and if there still is pending messages, the procedure repeats.

3.3 *ABEL*'s Case Studies

We show how to program for the case studies introduced in Chapter 2 by using *ABEL* APIs. For completeness, we repeat the *AbC* code snippets in Section 2.2, Chapter 2. Our message in this section is that given an *AbC* specification, the mapping from *AbC* to *ABEL* code is really straightforward.

3.3.1 Attribute Stable Marriage

Following the *AbC* paradigm, we build a system of interacting components using our API. Basically, we have two types of components, man and woman. Each component type executes the same code. It is the specific attribute environments provided when starting components make their behaviour different.

Man Component

Defining the attribute environment. The function *init_m* takes as parameters attributes characterizing a man component, from which an environment in form of map is built (line 2). The interface contains personal *id* and two characteristics $m1, m2$. Afterward, the component is created by a call to *start_component*. The component starts its execution via an invocation to *start_beh*.

```
init_m([Id, M1, M2, PW1, PW2, Size]) →
    Env = {id => Id, m1 => M1, m2 => M2, pw1 => PW1, pw2 => PW2,
            partner => 0, sent => 0, c => 0, counter => 0, n => Size, bl => sets:new()},
    I = {id,m1,m2},
    C = new_component(Env,I),
    start_beh(C,[q,p,r,a]).
```

Defining behaviour. A man component has a number of predicates for contacting potential partners. He would propose in the top down order, i.e., using first the most preferred predicate and down to the least one. Before sending a propose message, however, a query message is sent to collect the number of potential receivers. There is one querying process for each type of requirement. Take the definition of querying process Q in *AbC*

$$Q \triangleq Q_1 \mid Q_2 \mid Q_3 \mid Q_4$$

one can write this in *ABEL* using a parallel command as follows:

```
q(C,V) →
    parallel(C,V,[fun(_V) → q1(C,_V) end, fun(_V) → q2(C,_V) end,
              fun(_V) → q3(C,_V) end, fun(_V) → q4(C,_V) end]).
```

.

In *AbC*, the querying process for the first requirement has the following definition:

$$Q_1 \triangleq \langle \text{counter} = 0 \rangle (\text{`}q_1\text{'}, this.id, this.pw_1, this.pw_2)@(\text{id} \notin \text{this.bl}).[\text{counter} := \text{counter} + |\text{bl}|]Q_{11}$$

This can be defined in *ABEL* by using a prefix command, followed by a call to the function q11.

```
q1(C,V) →
   ... % Defining elements for the command
   prefix(C,V,{{G, M, SP, U}, fun(_V) → q11(C) end}).
```

where the variables passed to the command are defined as:

```
G = fun(L) → att(counter,L) == 0 end
M = {q1, this(id), this(pw1), this(pw2)}
SP = fun(L,R) → not sets:is_element(att(id,R),att(bl,L)) end
U = [{counter, fun(L) → att(counter,L) + sets:size(att(bl,L)) end}]
```

The continuation process Q_{11} in *AbC*, reported below is a choice process among two receives, both recursively call the same process as continuations.

$$Q_{11} \triangleq \langle \text{counter} < \text{n} \rangle (x = \text{`}interest\text{'})(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1]Q_{11}$$
$$+ \langle \text{counter} < \text{n} \rangle (x = \text{`}uninterest\text{'})(x).[\text{counter} := \text{counter} + 1]Q_{11}$$

We implement this process in *ABEL* in function q11 with use of a choice command.

```
q11(C,V) →
   ... % Defining elements for the command
   A1 = {G1, RP1, {x}, U1},
   A2 = {G2, RP2, {x}, U2},
   Con = fun(_V) → q11(C,_V) end,
   choice(C,V,[{A1, Con},{A2, Con}]).
```

where the variables passed to the command are defined as:

```
G1 = fun(L) → att(counter,L) < att(n,L) end
RP1 = fun(_,_,M) → element(1,M) == 'interest' end
U1 = [{c, fun(L,M) → att(c,L) + 1 end}, {counter, fun(L,_) → att(counter,L) + 1 end}]}
G2 = fun(L) → att(counter,L) < att(n,R) end,
RP2 = fun(_,_,M) → element(1,M) == 'uninterest' end,
U2 = [{counter, fun(L,_) → att(counter,L) + 1 end}]}
```

Following this way, we implement querying processes for the second and third requirements in functions q2 and q3 respectively. Their definitions are very similar to q1, as they are derived from *AbC* code, to which we sketch how their definitions in *ABEL* look like in the sequence.

$$
\begin{aligned}
Q_2 &\triangleq \langle \text{counter} = \text{n} \wedge \text{c} = |\text{bl}|\rangle(\text{‘}q_2\text{’}, \textit{this.id}, \textit{this.pw}_1, \textit{this.pw}_2)@(\text{id} \notin \text{this.bl}).\\
&\quad [\text{counter} := \text{counter} + |\text{bl}|]Q_{22}\\
Q_{22} &\triangleq \langle \text{counter} < 2 * \text{n}\rangle(x = \text{‘}\textit{interest}\text{’})(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1]Q_{22}\\
&\quad + \langle \text{counter} < 2 * \text{n}\rangle(x = \text{’}\textit{uninterest}\text{’})(x).[\text{counter} := \text{counter} + 1]Q_{22}\\
Q_3 &\triangleq \langle \text{counter} = 2 * \text{n} \wedge \text{c} = |\text{bl}|\rangle(\text{‘}q_3\text{’}, \textit{this.id}, \textit{this.pw}_1, \textit{this.pw}_2)@(\text{id} \notin \text{this.bl}).\\
&\quad [\text{counter} := \text{counter} + |\text{bl}|]Q_{33}\\
Q_{33} &\triangleq \langle \text{counter} < 3 * \text{n}\rangle(x = \text{‘}\textit{interest}\text{’})(x).[\text{c} := \text{c} + 1, \text{counter} := \text{counter} + 1]Q_{33}\\
&\quad + \langle \text{counter} < 3 * \text{n}\rangle(x = \text{‘}\textit{uninterest}\text{’})(x).[\text{counter} := \text{counter} + 1]Q_{33}
\end{aligned}
$$

```
q2(C,V) →
  . . . % Definitions for elements G, M, SP, U in Q2
  prefix(C,V,{{G,M,SP,U}, fun(_V) → q22(C,_V) end}).
q22(C,V) →
  . . . % Definitions for elements of actions A1, A2 in Q22
  Con = fun(_V) → q22(C,_V) end,
  choice(C,V,[{A1, Con},{A2, Con}]).

q3(C,V) →
  . . . % Definitions for elements G, M, SP, U in Q3
  prefix(C,V,{{G,M,SP,U}, fun(_V) → q33(C,_V) end).
q33(C,V) →
  . . . % Definitions for elements of actions A1, A2 in Q33
  Con = fun(_V) → q33(C,_V) end,
  choice(C,V,[{A1, Con},{A2, Con}]).
```

The last case is when all the previous requirements have been considered, yet the man is not able find any partner. This behaviour is modeled as an empty output as illustrated in the *AbC* process *Q*4.

$$Q_4 \triangleq \langle \text{counter} = 3 * \text{n} \wedge \text{c} = |\text{bl}|\rangle()@(\text{ff}).[\text{counter} := \text{counter} + 1]0$$

This process is implemented in function q4 as follows:

```
q4(C,V) →
 G = fun(L) → att(counter,L) == 3 ∗ att(n,L) andalso att(c,L) == sets:size(att(bl,L))) end,
 M = {},
 SP = fun(_,_) → false end,
 U = [{counter, fun(L) → att(counter,L) + 1 end}]),
 prefix(C, V, {{G, M, SP, U},nil}).
```

Next, we report on the process implementing the proposing phase of a man. In AbC, the definition of this process has the definition as

$$
\begin{aligned}
P \triangleq\ & \langle counter = n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{`}propose\text{'}, this.id, this.m_1, this.m_2)@(\Pi_1 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 2 * n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{`}propose\text{'}, this.id, this.m_1, this.m_2)@(\Pi_2 \wedge\ \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 3 * n \wedge partner = 0 \wedge c > |bl| \wedge sent = 0\rangle \\
& (\text{`}propose\text{'}, this.id, this.m_1, this.m_2)@(\Pi_3 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P \\
& + \langle counter = 3 * n + 1 \wedge partner = 0 \wedge n > |bl| \wedge sent = 0\rangle \\
& (\text{`}propose\text{'}, this.id, this.m_1, this.m_2)@(\Pi_4 \wedge \text{id} \notin \text{this.bl}).[\text{sent} := 1]P
\end{aligned}
$$

which uses the values of *counter* and c collected by querying processes in order to send propose messages. The behaviour is modeled as a choice among four sending processes, each has an output action, followed by same continuation. We can derive the *ABEL* code for this process as below.

```
p(C,V) →
    %% Definitions for elements of A2, A3, A4
    A1 = {G1, M, SP1, U},
    A2 = {G2, M, SP2, U},
    A3 = {G3, M, SP3, U},
    A4 = {G4, M, SP4, U},
    Con = fun(_V) → p(C,_V) end,
    choice(C,V,[{A1, Con}, {A2, Con}, {A3, Con}, {A4, Con}]).
```

A man component, after sending a 'propose' message waits for a number of replies. On receiving a first 'yes', the sender is considered as the current partner to which a 'confirm' is sent. Other subsequent senders who reply with a 'yes' are instead sent a 'toolate'. In AbC, we have the definition

$$
\begin{aligned}
A &\triangleq\ (x = \text{`}yes\text{'})(x, y).(H \mid A) \\
H &\triangleq\ (\langle partner = 0\rangle(\text{`}confirm\text{'})@(id = y).[partner := y]0 \\
& \quad + \langle partner > 0\rangle(\text{`}toolate\text{'})@(id = y).0
\end{aligned}
$$

In *ABEL* terms, we have corresponding functions a and h, reported below.

```
a(C,V) →
    RP = fun(_,_,M) → element(1,M) == yes end,
    prefix(C, V, {RP, {x}, fun(_V) → a1(C,_V) end}).

a1(C,V) →
    parallel(C, V, [fun(_V) → h(C, _V) end, fun(_V) → a(C,_V) end]).
h(C,V) →
    G1 = fun(L) → att(partner,L) == 0 end,
    M1 = {'confirm'},
    SP1 = fun(_,R) → att(id,R) == var(y,V) end,
```

```
  U = [{partner, var(y,V)}]},
  G2 = fun(L) → att(partner,L) > 0 end,
  M2 = {'toolate'},
  SP2 = fun(_,R) → att(id,R) == var(y,V) end},
  A1 = {G1, M1, SP1, U},
  A2 = {G2, M2, SP2},
  choice(C,V,[{A1, nil},{A2, nil}]).
```

On the other hand, whenever receiving a 'no', a man adds the sender to *bl* to not avoid further consideration. Moreover, during the waiting phase, a man may receives a 'bye' message from his partner. These two kinds of messages are handled by process R in AbC, whose definition is:

$$\begin{aligned} R \triangleq\ & (x = \text{`}bye\text{'})(x,y).[bl := bl \cup \{y\}, sent := 0, partner := 0]R \\ & + (x = \text{`}no\text{'})(x,y).[bl := bl \cup \{y\}, sent := 0]R \end{aligned}$$

In *ABEL*, the function r below uses a choice command where the first branch (A1) receiving 'bye' and the second (A2) receiving 'no'.

```
r(C,V) →
  RP1 = fun(_,_,M) → element(1,M) == 'bye' end,
  U1 = [{bl, fun(S,M) → sets:add_element(element(2,M),att(bl,S)) end},{sent, 0},{partner, 0}],
  RP2 = fun(S,R,M) → element(1,M) == no  end,
  U2 = [{bl, fun(S,M) → sets:add_element(element(2,M), att(bl,S)) end},{sent, 0}]},
  A1 = {RPred1, Upd1},
  A2 = {RPred2, Upd2},
  Con = fun(_V) → r(C,_V) end},
  choice(C,V,[{A1, Con}, {A2, Con}]).
```

Woman Component.

Defining attribute environment. The function *init_w* below takes charge in starting a women component. Its purpose is the same as *init_m*. The attribute names and their initial values can be seen from the code.

```
init_w([Id, W1, W2, PM1, PM2]) →
  Env = {id => Id, w1 => W1, w2 => W2, pm1 => PM1, pm2 => PM2, partner => 0,
        cm1 => −1, cm2 => −1, lock => 0, bl => sets:new()}
  C = new_component(Env,I),
  start_beh(C,[p1,p2,p3,w]).
```

Defining behaviour. The behaviour is essentially structured according to the types of incoming messages. First of all, a woman may receive query messages, to which she answers either 'interest' or 'uninterest'. As there are three different levels of a query, we use three parallel processes to handle

them. Their behaviour are defined in processes P_1, P_2, P_3 in AbC as follows:

$$
\begin{aligned}
P_1 &\triangleq (x = \text{`}q_1\text{'})(x, y, z, t).(H_1 \mid P_1) \\
H_1 &\triangleq \langle (z = w_1 \wedge t = w_2) \rangle (\text{`}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(z = w_1 \wedge t = w_2) \rangle (\text{`}\mathit{uninterest}\text{'})@(id = y).0 \\
P_2 &\triangleq (x = \text{`}q_2\text{'})(x, y, z, t).(H_2 \mid P_2) \\
H_2 &\triangleq \langle (z = w_1) \rangle (\text{`}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(z = w_1) \rangle (\text{`}\mathit{uninterest}\text{'})@(id = y).0 \\
P_3 &\triangleq (x = \text{`}q_3\text{'})(x, y, z, t).(H_3 \mid P_3) \\
H_3 &\triangleq \langle (t = w_2) \rangle (\text{`}\mathit{interest}\text{'})@(id = y).0 \\
&\quad + \langle \neg(t = w_2) \rangle (\text{`}\mathit{uninterest}\text{'})@(id = y).0
\end{aligned}
$$

to which we can write them in functions p1, p2 and p3 in *ABEL*, respectively. Below we illustrate only the definition of p1 as the other definitions follow the same structure.

```
p1(C,V) →
   RP = fun(_,_,M) → element(1,M) == 'q1' end,
   prefix(C,{RP,{x,y,z,t},fun(_V) → p11(C,_V) end}).

p11(C,V) →
   parallel(C,V[fun(_V) → h1(C,_V) end, fun(_V) → p1(C,_V) end]).

h1(C,V) →
   A1 = {fun(L) → var(z,V) == att(w1,S) andalso var(t,T) == att(w2,L) end,
         {'interest'},
         fun(_,R) → att(id,R) == var(y,V) end},
   A2 = {fun(L) → not (var(z,V) == att(w1,L) andalso var(t,V) == att(w2,L)) end,
         {'uninterest'},
         fun(_,R) → att(id,R) == var(y,V) end},
   choice(C,V,[{A1, nil}, {A2,nil}])
```

It receives a message whose the first element equals to atom $q1$, and binds the last three ones to variables Y, Z, T in that order. There, Y is the id of the sender, Z and T are his first and the second preferences. The follow-up choice process uses these variables in its awareness predicates to decide a reply message to be sent. In particular, action A1 send a 'interest' message to Y if both characteristics of the woman match Z and T. Action A2 sends a 'uninterest' in the other case. Notice that there is a replication of p1 because there may be many querying messages from different senders.

Secondly, a woman also receives and answers to 'propose' messages. In AbC, this task is handled by process W

$$W \triangleq (x = \text{'}propose\text{'} \land y \notin this.bl)(x, y, z, t).[\mathrm{bl} := \mathrm{bl} \cup \{\mathrm{y}\}](H \mid W)$$

$$H \triangleq \langle new_man_is_better \land \mathrm{lock} = 0\rangle(\text{'}yes\text{'}, this.id)@(id = y).[\mathrm{lock} := 1]W1$$

$$+ \langle \neg\, new_man_is_better\rangle(\text{'}no\text{'}, this.id)@(id = y).0$$

$$W1 \triangleq (m = \text{'}confirm\text{'})(m).B$$

$$+ (m = \text{'}toolate\text{'})(m).[\mathrm{lock} := 0, \mathrm{bl} := \mathrm{bl}\backslash\{\mathrm{y}\}]0$$

$$B \triangleq (\text{'}bye\text{'}, this.id)@(id = this.partner).[\mathrm{lock} := 0, \mathrm{partner} := \mathrm{y}, \mathrm{cm}_1 := \mathrm{z}, \mathrm{cm}_2 := \mathrm{t}]0$$

to which, we write a function w in $ABEL$, reported below.

```
w(C,V) →
   RP = fun(L,R,M) → element(1,M) == propose andalso not sets:is_element(att(id,R), att(bl,L)) end,
   U = [{bl, fun(L,M) → sets:add_element(element(2,M),att(bl,L)) end}],
   prefix(C, V, {{RP, {x,y,z,t}, U}, fun(_V) → wp(C,_V) end}).

wp(C,V) →
   parallel(C,V, [fun(_V) → h(C,_V) end, fun(_V) → w(C,_V) end]).
```

The choice command of h resolves for two possibilities: a proposer Y is better than the current partner and the opposite. This check is done by a helper function bof, which compares the characteristics of a proposer and a partner, with respects to the preferences of a woman.

```
h(C,V) →
   G1 = fun(L) → att(lock,L) == 0 andalso
          (att(partner,L) == 0 orelse bof(var(z,V),var(t,V), att(cm1,L), att(cm2,L), att(pm1,L), att(pm2,L))) end,
   M1 = {'yes', fun(L) → att(id,L) end},
   SP1 = fun(_,R) → att(id,R) == var(y,V) end,
   U = [{lock,1}],
   %% Definitions for elements of A2
   A1 = {G1,M1,SP1,U}
   A2 = {G2,M2,SP2},
   choice(C,V,[{A1, fun(_V) → w1(C, _V) end}, {A2, nil}]).
```

The continuation of A1 is implemented in function $w1$ that waits for an acknowledgement message from the same proposer Y. If a 'confirm' is received, the proposer is updated as the new partner. If a 'toolate' is received, the proposer is removed from the set *bl* to enable him to propose again.

```
w1(C, V) →
   A1 = {fun(_,_,M) → element(1,M) == 'confirm' end},
   A2 = {fun(_,_,M) → element(1,M) == 'toolate' end,
            [{lock, 0}, {bl, fun(R,_) → sets:del_element(Y,att(bl,R)) end}]},
   choice(C,V,[{A1, fun(_V) → b(C,_V) end},{A2, nil}]).

b(C,V) →
```

```
M = {'bye', fun(L) → att(id,L) end},
SP = fun(L,R) → att(id,R) == att(partner,L) end,
U = [{lock,0}, {partner,var(y,V)},{cm1, var(z,V)},{cm2, var(t,V)}],
prefix(C,V,{{M,SP,U},nil}).
```

3.3.2 Graph Coloring

The graph coloring problem consisting of assigning a color (a natural number) to each vertex such that two neighbours do not share the same color. In distributed setting, the proposed algorithm is a variant of the greedy algorithm. Each vertex is given an unique id, which implies an ordering over vertices. Color selection goes through a sequence of rounds. In each round, each vertex chooses a color which has not yet been used by any neighbors and propose to use that color. A vertex can take a proposed color if it has the greatest id among neighbors who are proposing the same color. In this case study, we only have one type of component representing for vertices of the input graph.

Defining attribute envionment. The listing below instantiates a vertex. The attribute environment is defined at line 2, where all the attribute names and their initial values are indicated. Environments are differ at vertex id and the set of neighbours. Interfaces for all vertices are the same, exposing vertex *id* for interaction. We also make use of *Erlang sets* for operations on sets.

```
init(Id, Nbr) →
  Env = #{id => Id, nbr => Nbr, color => −1, round => 0, done => 0, send => true,
          assigned => false, used => sets:new(), counter => 0, constraints => sets:new()},
  I = {id,nbr},
  C = new_component(Env,I),
  start_beh(C,[fun() → f(C) end, fun() → t(C) end, fun() → d(C) end, fun() → a(C) end]).
```

In the proposed algorithm for distributed graph coloring, each component maintains the following attributes:

- *id*: the unique id of a vertex.
- *nbr*: the set containing ids of neighbors.
- *color*: the color of a vertex.
- *round*: the current round a vertex is operating on.
- *done*: the number of neighbors who have finished color selection.
- *counter*: the number of neighbors who have not finished coloring (in the same round).
- *assigned*: a boolean in which a true value indicates the completion of coloring.

- *constraints*: a set of colors that have been proposed by neighbors with greater id (in the same round).
- *used*: a set of colors that have been used by neighbors.
- *send*: a boolean which is used to enable the sending of a proposed color.

A component is created by supplying Env and I to the function new_component. After that start_beh starts the execution of this component. The initial behaviour is a parallel composition of 4 processes F, T, D and A, whose behaviour definitions are f, t, d and a, respectively.

Defining behaviour. The behaviour of process F is encoded in the function f. If a vertex is not assigned a color and it has not yet proposed a color, this process sends a 'try' message associated with the value of *min_color* and the current round. *min_color* is a helper function returning the minimum number that does not appear in the argument set.

$$F \triangleq \langle \text{send} \wedge \neg \text{assigned} \rangle (\text{`}\mathit{try}\text{'}, \min\{\text{i} \notin \text{this.used}\}, \text{this.round})@(\mathit{id} \in \mathit{this.nbr}).$$
$$[\text{color} := \min\{\text{i} \notin \text{used}\}, \text{send} := \text{ff}]F$$

```
f(C,V) →
   G = fun(S) → att(send,S) andalso not att(assigned,S) end,
   M = {'try', fun(S) → min_color(att(used,S)) end, this(round)},
   SP = fun(S,R) → sets:is_element(att(id,S), att(nbr,R) end),
   U = [{send, false}, {color, fun(S) → min_color(att(used,S)) end}],
   prefix(C, {{G,M,SP,U}, fun(_V) → f(C,_V) end}).
```

Process T is implemented by a *choice* command among 4 receiving processes, each contains an input action and a continuation which is also the process T. Its definition in *AbC*

$$
\begin{aligned}
T \triangleq\ & (x = \text{`}\mathit{try}\text{'} \wedge \text{this.id} > \text{id} \wedge \text{this.round} = z)(x, y, z).\\
& [\text{counter} := \text{counter} + 1]T\\
+\ & (x = \text{`}\mathit{try}\text{'} \wedge \text{this.id} < \text{id} \wedge \text{this.round} = z)(x, y, z).\\
& [\text{counter} := \text{counter} + 1, \text{constraints} := \text{constraints} \cup \{y\}]T\\
+\ & (x = \text{`}\mathit{try}\text{'} \wedge \text{this.id} > \text{id} \wedge \text{this.round} < z)(x, y, z).\\
& [\text{round} := z, \text{send} := \text{tt}, \text{counter} := 1, \text{constraints} := \emptyset]T\\
+\ & (x = \text{`}\mathit{try}\text{'} \wedge \text{this.id} < \text{id} \wedge \text{this.round} < z)(x, y, z).\\
& [\text{round} := z, \text{send} := \text{tt}, \text{counter} := 1, \text{constraints} := \{y\}]T
\end{aligned}
$$

is translated as a function t which makes use of a choice command:

```
t(C) →
  RP1 = fun(R, S, M) → size(M) == 3 andalso element(1,M) == 'try'
              andalso att(id,R) > att(id,S) andalso att(round,R) == element(3,M) end,
  U1 = [{counter, fun(S,_) → att(counter,S) + 1 end}],
```

```
A1 = {RP1,U1},
%% Definitions of input actions A2, A3, A4
T = fun() → t(C) end,
choice(C,[{A1, T}, {A2, T},{A3, T},{A4, T}]).
```

In this function, A1, A2, A3, A4 are input descriptions corresponding to the prefix actions of the 4 branches of the choice. Particularly, A1, A2 responsible for receiving messages from neighboring vertices operating at the same round. These actions both increase the value of *counter*. A2 further records the proposed color of neighbors in *constraints* to avoid conflict. A3 and A4 instead receives 'try' messages associated with greater rounds. In which cases, the process indirectly starts a new round of color selection by enabling *send* and updating *counter*, *round*, *constraints*. At the startup of the new round, *counter* = 1 while *constraints* may contain the proposed color of a neighbor whose id is greater than the component's id.

Process D is also a choice, which receives 'done' messages sent from neighbors. Such a message causes the process to increase the attribute *done* and to store the color attached in the message in the set *used*. However, only messages associated with greater rounds have the effect of triggering a new round at the executing component. This is handled explicitly in action A2 where relevant attributes are updated accordingly. In particular, the current *round* is updated to the greater round, while attributes *send*, *counter* and *constraints* are reset to their initial values. Below are the process definition in *AbC*, followed by the corresponding *ABEL* code.

$$
\begin{aligned}
D \triangleq\ & (x = \text{`}\mathit{done}\text{'} \land \text{this.round} \geq z)(x, y, z).\\
& [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{y\}]D\\
+\ & (x = \text{`}\mathit{done}\text{'} \land \text{this.round} < z)(x, y, z).\\
& [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{\text{y}\},\\
& \text{send} := \text{tt}, \text{counter} := 0, \text{constraints} := \emptyset, \text{round} := z]D
\end{aligned}
$$

```
d(C,V) →
  RP1 = fun(R,_,M) → size(M) == 3 andalso
           element(1,M) == 'done' andalso att(round, R) >= element(3,M) end,
  U1 = [{done, fun(R,_) → att(done,R) + 1 end},
        {used, fun(R,_) → sets:add_element(element(2,M),att(used,R)) end}],
  A1 = {RP1,U1},
  %% The definition of A2
  D = fun(_V) → d(C,_V) end,
  choice(C,V,[{A1, D}, {A2, D}]).
```

Process A reports the completion of coloring a vertex. This function contains one send command, as reported below. The command is activated if the component with the environment S has selected a a color, the information of all neighbours are collected and the selected color does not give rise a conflict. The message to be sent is a tuple including the final color and the current round plus 1. Finally, the

atomic update on *assigned* disables the execution of process F.

$$A \triangleq \langle(\text{counter} + \text{done} = |\text{nbr}|) \wedge \text{color} > 0 \wedge \text{color} \notin \text{constraints} \cup \text{used})\rangle$$
$$(`done\text{'}, \text{this.color}, \text{this.round} + 1)@(\text{id} \in \text{this.nbr}).[\text{assigned} := tt]0$$

```
a(C,V) →
   G = fun(S) → att(color,S) > 0 andalso
               att(counter, S) == length(att(nbr, S)) − att(done,S) andalso
               not sets:is_element(att(color,S), sets:union(att(constraints,S),att(used,S))) end,
   M = {'done', this(color), fun(S) → att(round,S) + 1 end},
   SP = fun(S,R) → sets:is_element(att(id,S), att(nbr,R)) end,
   U = [{assigned, true}],
   prefix(C, V, {{G,M,SP,U},nil}).
```

3.4 ABEL performance on case studies

We have developed a prototype *ABEL* in Erlang. Our implementation can be found at [2]. We reports the performance evaluation of the prototype by using two case studies presented above:

- Stable marriage with Attribute (Section 3.3.1)
- Distributed graph coloring (Section 3.3.2)

For experiments, we used a workstation with a dual Intel Xeon processor E5-2643 v3 (12 physical cores) and 128 GB of memory. The OS version is a 64-bit generic Linux kernel 4.4.0 and the Erlang/OTP version is 21.1. It is noted that we are not interested in the optimality of the outcomes of these problems but rather consider if a particular outcome is valid according to the problem settings. In particular, for stable marriage problem, we are interested in if the matching returned is complete and stable; for the graph colouring problem, we are interested in if the number of colour in use is bound by the maximum degree of graph plus 1.

3.4.1 Stable Marriage with Attribute

We perform some experiments to evaluate the *ABEL* program solving SMA instances. Our input is randomly generated from the predefined probabilities of attributes and preferences in the following way. We first define ranges of values for attribute and preference. In a given range, a probability is associated to each value so that the sum of the probabilities is 1. In this way, an attribute (or preference) can take a concrete value v with a probability $p(v)$. A generator is implemented to generate different instances based on these parameters and on the number of agents. At the end, we

consider 2 attributes and 2 preferences with ranges of two values. We select 10 different combinations of probabilities values and generate 10 instances for each.

For each instance, we ran the *ABEL* program 5 times over 5 different configurations of the binary tree structure, and took the average of their execution times. This means that each number reported below are averaged over 2500 runs. For each matching returned, we have also checked for the completeness and stability conditions. Note that any men agent can relax his requirement to a "true" predicate, the matching is expected to be complete.

Table 3.1 presents the numbers for some problem sizes, which correspond to the number of pairs of *AbC* components.

# pairs	times (s)
60	1.52
80	3.47
100	6.7
200	62.5

Table 3.1: Results of stable marriage with attributes

In addition, we have also measured the number of messages exchanged and noticed that the case study required a huge number of messages, which increases very fast when adding more agents. Indeed, all agents need to remain in the matching system (and thus keep delivering messages) until the stability and completeness conditions are satisfied for a pair of agents, when gets matched can still be split afterwards. During this time, participating agents get to deliver irrelevant messages, for example, messages come from those of the same type and extra acknowledgement messages which was meant to be point to point.

3.4.2 Graph Coloring

To evaluate the performance of the *ABEL* code modeling the graph coloring scenario, we conducted some experiments several DIMACS graphs collected from various public sources: flat300_28_0.col (300 vertices and 21695 edges), dsjc500.1.col (500 vertices and 12458 edges), will199GPIA.col (701 vertices and 7065 edges) and dsjc1000.1.col (1000 vertices and 49629 edges). The datasets chosen provides an increasing number of vertices which are considered as *AbC* components.

For each graph, we ran the *ABEL* code 10 times. In addition, we considered different configurations of the tree infrastructure in terms of the number of nodes. The following metrics are measured: the running time in seconds, the number of color used, the number of rounds, the total number of messages exchanged between vertices and the infrastructure and the total message size in MB. In our measurement, vertices do not wait for each other to reports the completion of their coloring: as soon as a vertex decides on a color, it communicates that color, the number of messages exchanged (and message size) thus far to an external process. This process reports the final outcome once all vertices

have communicated their colors. To make sure the returning colors are valid, we have also checked for the presence of conflicts. We have also validated that the number of colors does not exceed the maximum graph degree plus 1 (due to the greedy nature of the algorithm).

Table 3.2 shows the results of the experiment. T is the number of tree nodes involving in mediating the messages. These numbers correspond to binary trees with depths from 1 to 5. Next columns to the right show the numbers for color, round, messages and the total size in that order. The execution times are the average of 10 runs, other numbers are instead the average over 10 runs and 5 different configurations.

Graph	**Execution Times** (in seconds)					**#C**	**#R**	**#Msg** (in milli.)	**Size** (in MB)
	T = 1	T = 3	T = 7	T = 15	T = 31				
flat300_28_0	37.4	20.5	15.4	13.8	13.6	47.7	151.7	4.67	22,229
dsjc500.1	29	16.8	12.1	9.7	9	19.6	69.7	5.58	10,449
will199GPIA	71.9	46.7	35.9	27.7	20.4	9.1	165.2	15.9	14,734
dsjc1000.1	372.1	175.5	124	105.3	98.8	31.5	144	46.3	156,966

Table 3.2: Results of graph coloring with different number of tree nodes

Overall, the *ABEL* code can perform coloring for experiment graphs without any conflicts, and resulted in speedups when increasing the number of tree nodes. On the other hand, its performance varies on different graphs. This has something to do with the specific topology of each graph. In general, for graph with more edges, components are more likely to resolve the color conflict among neighbors and thus require more interactions. This contributes to an increased number of rounds and messages. For graph with more vertices, components get to deliver broadcasted messages from non neighbors which can delay the actual participation in coloring.

Table 3.2 suggests it is the number of rounds and the total message size that affected most to the performance. The number of rounds varies due to non determinism resulted from the execution of the program. The total message size does not depends only the number of messages. For example, the graph with 300 vertices uses less messages than the one with 700 vertices, yet its total message size is much larger than the latter. The reason for this is as follows. In the current model of the graph coloring scenario, the interaction interfaces of vertices contain two attributes: an id and neighbor list. Consequently, every message a vertex sends out has to include also the portion of environment derived from the interface (see the semantics in Chapter 2). In our case, this portion contains the list of neighbors of the vertex, which is large for dense graphs (e.g., flat300_28_0.col) and small for spare graphs (e.g., will199GPIA.col). This shows that a faithful but native implementation of AbC semantics could make the performance overhead high.

We are interested in understanding better these costs and considering optimization strategies that may help to mitigate them. First of all, observe that vertices do not need to include their neighbor lists in messages. This can be seen from receiving predicates in processes T and D where only sender's id is needed. Secondly, assigned vertices, although do not involve into graph coloring, still delivers broadcasted messages and hold the system resources. Note that further guarding T and D with a

condition on the attribute *assigned* does not prevent a vertex from delivering messages. However, assigned vertices can be terminated safely without affecting other vertices. To this end, we perform another experiment taking into account the following strategies.

(a) At the algorithm-wise, we modify the color proposal process (as explained in Chapter 2) so that the selection of a color proposal takes into account the set *constraints*.

(b) At the specification-wise, we limit the portion of environment by introducing a limited interface containing only the attribute *id*.

(c) At the system-wise, we terminate assigned vertices by stopping their coordinators upon processing termination events generated by process A.

Although somewhat program-specific, they are helpful for the following reasons. Experimenting with the first point confirms our intuition when reasoning the coloring scenario at the specification level. It further implies that *ABEL* faithfully preserve *AbC* semantics. The second strategy can be generalized to apply for other case studies. The last strategy could lead to future works in which new constructs can be introduced to support expressing when a component can terminate.

We have prepared several programs combining the above strategies and evaluated their performances: a) the program that uses the modified version of color proposal process, b) the program in which vertices use a limited interface, c) the program in which assigned vertices are terminated naturally by the coordinator, d) the program with use of the last two strategies and e) the program with modified color proposal, featuring the last two strategies.

Tables 3.3, 3.4 show the execution times, the number of colors, the number of rounds, the number of messages exchanged and the total size of messages for these programs. We only include numbers for two graphs and for two configurations of the infrastructure, i.e., the smallest and largest ones among them.

Program	**Exec. Times** (s)		**#C**	**#R**	**#Msg** (in milli.)	**Size** (in MB)
	T = 1	T = 31				
o) orig.	37.4	13.6	47.7	151.7	4.67	22,229
a) mod.	28.2	11.2	47.7	126.3	3.71	17,668
b) orig. + lim. intf.	15.5	8.7	47.6	151.4	4.7	1,471
c) orig. + g.c.	24	9.3	47.5	150.6	4.65	22,138
d) orig. + lim. intf. + g.c.	9.9	5.9	47.5	150.5	4.72	1,444
e) mod. + lim. intf. + g.c.	7.8	5.2	47.7	127	3.77	1,177

TABLE 3.3: Results on 300 vertices and 21695 edges graph

We can see the differences between the performance of the original program o) and those of optimization versions in terms of the number of message exchanges, message size and the number of rounds. In particular, an increasing number of algorithmic rounds has slowed down the coloring procedure (17%

Program	**Exec. Times** (s) T = 1	T = 31	**#C**	**#R**	**#Msg** (in milli.)	**Size** (in MB)
o) orig.	372.1	98.8	31.5	144	46.3	156,966
a) mod.	310.6	77.6	31.4	124.7	37.6	127,716
b) orig. + lim. intf.	131.3	51.3	31.6	144.5	45.7	14,282
c) orig. + g.c.	226.6	61.7	31.4	144.3	46.1	156,343
d) orig. + lim. intf. + g.c.	85.2	35.6	31.4	144.6	46.2	14,452
e) mod. + lim. intf. + g.c.	71.3	27.3	31.3	124.3	38	11,876

Table 3.4: Results on 1000 vertices, 49629 edges graph

for Table 3.4 and 25% for Table 3.3). With the introduction of limited interface, the differences between o) and b) shows that the message size is the main overhead. Despite the facts that the number of messages exchanged and the number of rounds are similar in o) and b), the total message size of b) is only 7-8% that of o). This reduced the computation times significantly with 60-65%. An interesting case is at terminating assigned vertices in the program c). The outcome of c) is very similar to o) in both experiment graphs, however using c) have reduced 35-40% computation times. As a result, when combining all optimization strategies, we have the best performance.

3.5 Concluding remarks

We have presented *ABEL*, an implementation of *AbC* in *Erlang*. We build an API that mimics *AbC* constructs for the purpose of developing and experimenting with the *AbC* paradigm. The API is integrated seamlessly with underneath coordination mechanisms that together simulate the original synchronous semantics. Because of the direct correspondence between the two formal semantics and our actual implementations, we can perform formal verification of *ABEL* programs by considering their *AbC* abstractions. Indeed, from *AbC* specifications we can obtain verifiable models that can be provided as input to model checkers.

Generally, it is still not clear how to detect when an *AbC* component can be garbage collected. Our prototype does not provide automatic garbage collection for components. A viable solution is to provide explicit guidance from the program with support of appropriate constructs, for example by introducing a *kill* process such as the one in [93]. Beside this, we are currently investigating an asynchronous semantics for *AbC* in order to relax the total order of message delivery.

To conclude this chapter, we elaborate on some knowledge gained in the experiments with *ABEL*.

The *AbC* calculus provides a new way to select communicating partners at run-time that are highly dynamic and flexible. This may be considered as a big advantage compared to other paradigms when it comes to modelling complex interactions. At the implementation level, it may be challenging to preserve the *AbC* semantics.

Previous approaches to providing runtime environments for *AbC* have two limitations: i) they do not exactly preserve the original semantics of the calculus, making it hard to reasoning about the

program and ii) the APIs provided do not closely capture *AbC* primitives, making the task of deriving executable code from *AbC* specification not obvious.

One of the main issues is at the implementation of guards (or awareness). In *AbC*, the evaluation of guards, the communication actions and the attribute updates are all performed atomically. The existing programming frameworks have separated the evaluation of guards from executing actions. This separation exhibits the chance of interleaving between the evaluation of guards and the execution of an associated action. The problem arises if an awareness process, say $\langle \Pi_1 \rangle P_1$ becomes active (i.e., the predicate Π_1 holds). However, before P_1 can take any action, another parallel process may change the shared environment in a way that Π_1 is no longer hold. Now, if P_1 continues with its action then we run into semantic problem.

Our experiences with *ABEL* have showed that:

- a solution for ensuring atomicity between awareness and communication actions is to use a special process acting as a coordinator (that keeps track of the environment). And it is the coordinator that decides the actual actions to be executed, not individual processes. *ABEL* views component environment as a living process instead of a static data store, reflecting a shift from "shared memory" (where individual processes execute actions by accessing the environment) to "message passing" (where individual processes send actions to the environment for execution);
- the development of the intra- and inter-coordinators within *ABEL* for preserving *AbC* semantics are not *Erlang*- specific. Although we have exploited the OTP library for fast prototyping and robustness, any framework that follows the main design principles in this chapter can be an alternative to *ABEL*;
- the cost of preserving semantics may be high. On the one hand, broadcast is used to ensure reliable messaging which causes the problem of large message sizes as any sent messages must be decorated with enough information for receivers to check (i.e., the closure of sending predicate and the sender environment). On the other hand, the implementation of component must be prepared to perform the checks for any incoming message. It can happen that the components would waste times on filtering out uninterested messages instead of involving to actual computations.

Chapter 4

Systematic verification of qualitative properties

In this chapter, we will be concerned with the automatic verification of systems relying on attribute-based communication, for which the AbC calculus serves as a vehicle to build formal verifiable models. Our goal is to understand the complexity of systems described via the AbC calculus, such as those presented in Chapter 2, and the way properties of interests can be verified against a given specification. To this end, we discuss how to perform a systematic analysis of AbC models and verify their qualitative properties. We have not yet considered probabilistic aspects of the model in this attempt, and leave them for future work.

Our approach relies on the UMC verification framework [73, 81], which provides an execution environment for UML-like state machines. The underlying formal model of UMC, based on doubly-labeled transition systems (L2TSs) [53], is well-represented for AbC systems, in which nodes are labelled with the components attributes, and edged are labelled with the occurring communication events. Properties of interest that a designer would like to verify can then be specified over (a set of) composite labels by using the supported event/state based temporal logic, reasoning over the L2TS.

To use the UMC framework, the initial step in our verification approach consists in mechanically translating AbC terms into UMC. The main effort of our translation lays in the careful modelling of the attributes, these implicitly require providing some kind of global view of the system. Since in our target modelling language there is no concept of global data among different state machines, a simple solution would require implementing shared states and appropriate synchronisation. To avoid this, we gather all the processes in the initial system, along with their attributes, into a unique object of the target model, where the behaviour described by a single process term is captured by one or more transitions. This guarantees direct access to attributes by any of the modelled processes, while components interleaving is triggered via global conditions which requires some ingenuity to preserve the semantics of the initial system. To deal with process interleaving, we explicitly track the

execution points of the processes and introduce appropriate guards on the transitions that can be used to guarantee that any point of the evolution of the system only feasible transitions are performed. Eventually, depending on the properties of interest, we may need to instrument the translated model and introduce appropriate abstraction rules to increase the efficiency of the analysis. The resulting model and the set of properties can then be used as inputs to UMC model checker.

We illustrate the impact of our approach by considering two case studies, namely stable marriage and graph coloring problems presented in Chapter 2. For the former, we consider, besides the classical version [65], other two variants of the problem which allows agents to express their interests in potential partners by using their attributes rather than their identities ordered by means of an explicit preference list. Member's preferences are represented as predicates over the attributes of potential partners. For one variant, we follow the classical algorithm where men first propose to the women they prefer most and then relax their expectations if no partner is willing to accept their proposal. In the other variant, men start proposing with the lowest requirements, to make sure to get a partner, and gradually increase their expectations hoping to find better partners. We are interested properties such as the stability of the matching and its completeness, in the existence of a unique solution, and in the level of satisfaction of the components.

For the latter, our solution is based on the *AbC* specification originally proposed in [23]. We are interested in the termination and in the soundness of the algorithm, e.g., in terms of checking valid final colors assigned to vertices. Interestingly, with our approach we could spot some issues in the original specification of the graph coloring scenario [24], which have been also mentioned in Section 2.2.2 (Chapter 2).

These properties are first described informally and then rendered as logical formulae to be formally checked against the generated models. The outcome of our verification allows us to make some considerations both on the different algorithms and on the used tool. Indeed, the results of our experiments have shown that systems relying on attribute-based communications can be particularly complex to design and analyse. However, by exhaustively verifying a specification over all possible inputs, despite the small problem size considered, we have experienced that many non-trivial emerging properties and potential problems can indeed be discovered.

In the rest of the chapter, we first briefly describe the syntax of the *AbC* calculus supported by our approach. We then present in details the translation from *AbC* process terms into UMC's textual description of UML-like state machines. We show how to specify in *AbC* algorithmic solutions for both case studies, and present fragments of the result of our verification and discuss their outcomes. Finally, we discuss the difference between the previous work [45] and our refinement.

4.1 Compiling from AbC into UMC

4.1.1 Specifying AbC systems

We define a template, shown in Figure 4.1 for the modeling purpose. It is used to specify an *AbC* system in a prescribed manner. Overall, the template allows defining components with given types

```
-- Component type declarations
component Type
   attributes: attr1 , attr2 , ...
   observables: attr1 , attr2 , ...
   behaviour:
       let {
       -- process definitions in AbC-like syntax
       }
       init
       -- a process call or a process expression
end
-- other components declarations ...

-- an instantiate of component Name
C1 : Type (attr1 → val1 , attr2 → val2 ...)
-- other instances ...
```

FIGURE 4.1: A template for *AbC* specification

in blocks starting at **component** and ending at **end**. A component consists a set of **attributes** and a **behaviour** description. It supports also the declaration of **observables** attributes, which is an optional subset of component attributes. Those attributes declared in this section will be exposed as state labels in the translated model. The behaviour is described by a set of process expressions, thus one first defines sub-processes inside the block **let** $\{\ldots\}$ and initiates the behaviour after the keyword **init** which can be either a call or a process expression composed from defined processes. Each concrete component is then instantiated by assigning concrete values to attributes. Finally, the *AbC* system is considered as the collection of component instances defined.

Before presenting the translation, we need to clarify some assumptions made on the input syntax. Basically, the syntax for process definitions inside the **let** and **init** sections (Figure 4.1) follows closely to that of the AbC calculus ([23], Chapter 2). The syntax is shown in Figure 4.2.

We have adopted the following notational conventions. The inactive process is written as *nil*. The notations $\tilde{E}$ and $\tilde{x}$ stand for sequences $E_1, E_2, \ldots$ and $x_1, x_2, \ldots$, respectively. Similarly, an update descriptor $[\tilde{a} := \tilde{E}]$ is a shorthand for $[a_1 := E_1, a_2 := E_2, \ldots]$.

Predicates can be either the constants true and false, or are built from some binary operator $\bowtie$ over two expressions. A predicate can combine other predicates by using logical operators and, or, not. The relational operator $\bowtie$ can be either comparison operators such as $>, <, =, \ldots$ or membership operator on sets, e.g., *in*, *notin*.

$$
\begin{array}{rl}
\text{(Processes)} & P ::= nil \mid (\tilde{E})@(\Pi).U \mid (\Pi)(\tilde{x}).U \mid \langle\Pi\rangle P \mid P + P \mid P|P \mid P|^{m} P \mid K \\
\text{(Update)} & U ::= \ [\tilde{a} := \tilde{E}]P \mid P \\
\text{(Predicates)} & \Pi ::= \text{true} \mid \text{false} \mid E_1 \bowtie E_2 \mid \Pi \text{ and } \Pi \mid \Pi \text{ or } \Pi \mid \text{not } \Pi \\
\text{(Expressions)} & E ::= v \mid x \mid \$x \mid a \mid this.a \mid E_1 \ \circ \ E_2 \mid E.hd \mid E.tl \mid E[E] \mid |E| \\
\text{(Relations)} & \bowtie ::=> \ \mid \ < \ \mid \ = \ \mid / = \ \mid \ <= \ \mid \ >= \ \mid in \mid notin \\
\text{(Operators)} & \circ ::= + \mid \ - \ \mid \ * \ \mid / \mid \ ++ \mid \ --
\end{array}
$$

FIGURE 4.2: Process syntax

An expression can be a constant v, an attribute name a, a variable x, the value $\$x$ of a variable, the value $this.a$ of an attribute a in the local attribute environment, or the result of an arithmetic operation $\circ$ between two expressions. A constant v can be either numerals, literals or UMC vectors [12]. For numeral expressions, operator *op* are standard ones, i.e., $+, -, /, *$. For a given vector E, the expressions $E.hd$, $E.tl$, $E[i]$, $|E|$ return the first element, the rest, the i-th element and the length of the vector, respectively. An empty vector is a constant and declared as []. UMC vectors are further exploited to represent (ordered) sets. Two operators on sets are $++$ and $--$ for set union and set subtraction.

To guarantee a finite representation of the model, we further assume that the specified system consists of a fixed number of components and that the interleaving operator generally does not occur inside a recursive definition. The only allowed exception is the definition of a process of the form $P := Q|P$, where $|$ is replaced by its bounded version $|^{m-1}$, with $m > 1$ the number of parallel instances to be created. For example: $P := Q \ |^2 \ P$ is interpreted as three processes $P := Q_1 \mid Q_2 \mid Q_3$, where $Q_i \triangleq Q$. This is useful to model replication behaviour.

Example 4.1 As an example, we illustrate modeling the SMP problem using the above interface. The classical SMP considers matching two equally-sized sets of men and women according to their preferences specified as an ordered list of partner's identifiers. The algorithm goes through a sequence of proposals initiated by men. Men propose themselves to women according to their preference lists. A woman, when receiving a proposal, chooses the best man between her current partner and the man making advances. Gale and Shapley [65] showed that such an algorithm guarantees existence of a unique stable matching, in the sense that the algorithm always leads to a situation where everyone is engaged and it is not possible that any two, not married, persons prefer each other over their current partners.

Men (initiators) and Women (responders) are AbC components whose attributes include the identifier `id`, the preference list `pref`, and the current `partner`. Accordingly, we define two types of components, `Man` and `Woman`. The first component type has the declaration as shown in Figure 4.3.

```
component Man
  attributes: id, partner, pref
  observables: partner
  behaviour:
        let {
            M := (‘propose’, this.id)@(id = this.pref.hd).
                  [partner := pref.hd, pref := pref.tl]M1
            M1 :=(x = ‘no’)(x).[partner := 0]M
           }
        init M
end
```

Figure 4.3: Declaring component Man

```
component Woman
  attributes: id, partner, pref
  observables: partner
  behaviour:
    let {
       W := (x = ‘propose’)(x,y).H |^(n-1) W
       H := <pref[partner] < pref[$y]>(‘no’)@(id = this.partner).
             [partner := $y]nil
             +
             <pref[partner] > pref[$y]>(‘no’)@(id = $y).nil
      }
     init W
end
```

Figure 4.4: Declaring component Woman

The behaviour of an individual initiator is specified by process `M`. It sends a ‘`propose`’ message to components whose `id` equals to the first element of `pref`, and then updates attributes `partner` and `pref`. The continuation process `M1` waits for a ‘`no`’ message to reset `partner`, before restarting with `M`.

The `Woman` component is declared in a similar fashion. Its behaviour is specified by process `W` (Figure 4.4). Specifically, `W` is a parallel composition of several instances of the subprocess (`x` = ‘`propose`’)(`x`, `y`).`H`, each waits for an incoming proposal and proceeds as `H`. In this way, the component is able to receive new ‘`propose`’ messages while processing the current one. The number of messages a woman component can receive in parallel is the number of men in the matching system (denoted by n).

Process `H` checks whether or not the sender, whose *id* stored in $y is better than the current partner. We use a reversed form of preference lists `pref` in order to perform this comparison. In the first case, `H` sends a ‘`no`’ message to the now ex partner, and updates `partner` to the corresponding sender. In the second case, `H` sends a ‘`no`’ to the sender. Both branches terminate with a `nil` process.

Finally, assume that we have 3 pairs of agents. The following code shows how to instantiate concrete components by giving initial values to attributes. Each concrete component is given a name, and is an instance of a component type. We can see the instantiation of vectors `pref` and in particular, the reversed form of `pref` for Woman components.

```
C1: Man(id -> 1, partner -> 0, pref -> [4,5,6])
C2: Man(id -> 2, partner -> 0, pref -> [4,5,6])
C3: Man(id -> 3, partner -> 0, pref -> [4,5,6])
C4: Woman(id -> 4, partner -> 0, pref -> [0,3,2,1])
C5: Woman(id -> 5, partner -> 0, pref -> [0,1,2,3])
C6: Woman(id -> 6, partner -> 0, pref -> [0,1,3,2])
```

4.1.2 The translation

The input system is a collection of a number of *AbC* components, in which the specification for the k-th component, denoted with

$$\Gamma_k : \langle D_k, P_{init_k} \rangle$$

includes an attribute environment Γ_k, a set D_k of process declarations, and an initial behaviour P_{init_k} which refers to the processes defined in D_k. The output of our translation is a UMC class whose general structure is depicted in Figure 4.5. It includes fixed code snippets (boldfaced) such as the necessary signals and data structures to model *AbC* input and output actions. It also contains the dynamic parts such as the declarations of attributes (line 10), attribute values initialization (line 31), and the transitions for modeling the components behaviour (lines 21 - 24).

An *AbC* system is modelled as a UML parallel state machine (`SYS`), where each component is modelled by its own region (`Ck`). In UML the behaviors inside parallel regions proceed in a single move: a trigger event from the machine queue is dispatched to all the regions, which may be then evolved in parallel, making a unique system transition. The semantics of attribute input and output is modelled by using two unique events: i) the `bcast(tgt, msg, j)` event that triggers all the receive actions in all components. It carries out the set `tgt` of component indexes allowed to receive the message `msg`, and the index `j` of the sending component; ii) the `allowsend(i)` event with `i` ranges over component indexes, that is used to schedule the components through interleaving when sending messages. According to the semantics of *AbC*, *receive* actions are blocking and executed together, and *send* actions of all the components should be handled in an interleaved way. To accommodate this, we use the event queue of the state machine to store a set of `allowsend(i)` signals, one for each *AbC* component. These signals are declared in the top state of the system as `Defers`, to prevent them from being removed from the event queue when they do not trigger any transition. Moreover, the queue is defined as `RANDOM` so that the relative ordering of signals is not considered relevant. In this way, at each step in which an *AbC send* has to be performed, a single `allowsend(i)` signal is nondeterministically selected from the queue, allowing a single component, whose index equals to i, to proceed.

The `Transitions` section collects all the transitions generated from the process terms while visiting the process structure. Transitions have the following form:

```
SYS.Ck.s0 -> Ck.s0 {Trigger[... & pc[k][p]=CNT_in]/
```

```
0  [[(Γ0 : ⟨D0, P_init0⟩, Γ1 : ⟨D1, P_init1⟩ ..., Γs : ⟨Ds, P_inits⟩)]] =
1    Class System is
2     Signals: allowsend(i:int),
3              bcast(tgt,msg,j:int);
4     Vars:
5        RANDOMQUEUE;
6        receiving:bool := false;
7        pc:int[];
8        bound:obj[];
9       /* Attributes vectors of the form - attr1:int[]; attr2:int[]; ...*/
10      A[[(Dom(Γ0), Dom(Γ1), ..., Dom(Γs))]]
11   State Top Defers allowsend(i)
12    Transitions:
13      /* Initial movement of the system */
14      init → SYS {-/
15       /* program counter instantiation */
16       pc :=  [[1, 1, ...], [1, 1, ...], ..., [1, 1, ...]];
17       for i in 0..pc.length-1 {
18            self.allowsend(i);
19       }}
20      /* Transitions of all components */
21      S[[⟨D0, P_init0⟩]]
22      S[[⟨D1, P_init1⟩]]
23      ...
24      S[[⟨Ds, P_inits⟩]]
25   end System
26    SYS : System (
27       /* Attributes values instantiation of the form
28       attr1 → [Γ0(attr1), Γ1(attr1), ..., Γs(attr1)],
29       attr2 → [Γ0(attr2), Γ1(attr2), ..., Γs(attr2)],
30       ... */
31       V[[(Γ0, Γ1, ..., Γs)]]
32   )
33    Abstraction {
34       /* labels on communication actions */
35       Action sending($1, $2) → send($1, $2)
36       Action received($1, $2) → receive($1, $2)
37
38       /* labels on observable attributes */
39       State attr1[comp_id] = $v → has_attr1(comp_name,$v)
40       State attr2[comp_id] = $v → has_attr2(comp_name,$v)
41       ...
42   }
```

FIGURE 4.5: Translation of *AbC* specifications.

```
    -- transition body
    pc[k][p]:=CNT_out;
}
```

where pc[k][p] is the *program counter* of a process (indexed by) p in a component (indexed by) k. `CNT` models the *execution point* of a transition (or an action of the process), whose values are calculated by the translation. More concretely, we associate two values to each transition: an *entry point* $\mathtt{CNT}_{\mathtt{in}}$ and an *exit point* $\mathtt{CNT}_{\mathtt{out}}$. A guard pc[k][p] $=$ $\mathtt{CNT}_{\mathtt{in}}$ on a transition makes sure that the program counter holds the "address" of the transition so that it can be executed. At the end of a transition, pc[k][p] is assigned a new value in order to correctly enable the next set of feasible transitions. The values of pc, `CNT`, and the full guards are worked out according to the structural mapping procedure described in the following section.

4.1.2.1 Structural Mapping.

Let us denote with $\mathcal{S}[\![P]\!]_\rho^{k,p,v}$ the function that maps a process P of component k into a number of UMC transitions, where p the process index, and v the entry point. The information carried while traversing the process structure is stored in a map ρ, which are necessary for translating a specific AbC action. Initially, $\rho = \{upd \to \emptyset, aware \to \emptyset, parent \to \emptyset, visit \to [], entry \to []\}$. upd and $aware$ store the update descriptor and the awareness predicate associated to an action, respectively. For a process name K indexed by some p, $visit[K_p]$ is a boolean value indicating if K_p has been translated before. $entry[K_p]$ stores the entry point of K_p, $parent$ instead stores the exit point of the process that spawns the process P under the translation.

In addition, the translation maintains several global variables: a) the current process index $pid[k]$ of component k, initially set to 0; b) the global counter $cnt[k][p]$ of a process with index p in component k, initially set to 1 and c) the set of process declaration D_k for component k. At the beginning, the parameter set passed to $\mathcal{S}[\![\cdot]\!]$ contains P_{initk} - the initial behaviour of the component k, and p, v receiving the values of $pid[k][p]$ and $cnt[p]$, respectively.

The translation may perform updates on global variables and modify the parameter map ρ. In the following translation rules, we write $(\downarrow_g)$ to express side effects on global variables and use this font to access global variables. Operators on ρ include accessing the value of a key x in ρ - $\rho(x)$ and creating a new map ρ' from an existing ρ by overwriting some key values in ρ, i.e., $\rho' = \rho\{x \mapsto val1, \ldots\}$. For brevity, we write $\mathsf{cnt[p]}$ for $\mathsf{cnt[k][p]}$ and pid for $\mathsf{pid[k]}$.

Figure 4.6 presents our translation rules from AbC process terms to UMC transitions, while Figure 4.7(b)-(d) gives an idea of how transitions are glued together according to the process structure.

Inaction. An inaction process is translated into nothing.

Awareness. The translation of $\langle\Pi\rangle P$ accumulates predicate Π into variable $aware$ of ρ and returns the translation of P under the updated map ρ'.

Nondeterministic choice. The translation of $P_1 + P_2$ is a sequence of two translations of sub-processes under the same process index p, entry value v, and map ρ.

Parallel composition. The translation of $P_1|P_2$ is a sequence of two translations of the sub-processes. It generates two new processes indices p_1 and p_2 which are calculated from the current process index pid, and initialises two new global counters $\mathsf{cnt[p_1]}, \mathsf{cnt[p_2]}$. In the case of parallel composition, the entry points of sub-processes P_1, P_2 does contain not only their own counters but also the counter v of the process that spawns $P_1|P_2$, whose index is p. Therefore, the translations of P_1, P_2 store the exit point of the "parent" process (p, v) in variable $parent$ which will be used as an additional guard for prefixing actions of P_1 and of P_2. This however, can be ignored if $v = 1$, meaning that $P_1|P_2$ is not

$$
\begin{aligned}
\mathcal{S}[\![nil]\!]_{\rho}^{k,p,v} &= \emptyset \\
\mathcal{S}[\![\langle\Pi\rangle P]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v} \\
&\quad \texttt{where } \rho' = \rho\{aware \mapsto \rho(aware) \cup \Pi\} \\
\mathcal{S}[\![P_1 + P_2]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P_1]\!]_{\rho}^{k,p,v}; \mathcal{S}[\![P_2]\!]_{\rho}^{k,p,v} \\
\mathcal{S}[\![P_1 | P_2]\!]_{\rho}^{k,p,v} &= \begin{cases} \mathcal{S}[\![P_1]\!]_{\rho}^{k,p,v}; (\downarrow_{\mathrm{g}})\mathcal{S}[\![P_2]\!]_{\rho}^{k,\mathsf{pid},\mathsf{cnt[pid]}} & \text{if } v = 1 \\ (\downarrow_{\mathrm{g}})\mathcal{S}[\![P_1]\!]_{\rho'}^{k,\mathsf{pid},\mathsf{cnt[pid]}}; (\downarrow_{\mathrm{g}})\mathcal{S}[\![P_2]\!]_{\rho'}^{k,\mathsf{pid},\mathsf{cnt[pid]}} & \text{otherwise} \end{cases} \\
&\quad \texttt{where } \downarrow_{\mathrm{g}} = \{\mathsf{pid} := \mathsf{pid} + 1, \mathsf{cnt[pid]} := 1\} \\
&\qquad\qquad \rho' = \rho\{parent \mapsto \rho(parent) \cup (p, v)\}, \\
\mathcal{S}[\![Q|^{m} P]\!]_{\rho}^{k,p,v} &= (\downarrow_{\mathrm{g}})\mathcal{S}[\![Q' | \underbrace{Q' | \ldots | Q'}_{\text{m times}}]\!]_{\rho}^{k,p,v} \\
&\quad \texttt{where } \downarrow_{\mathrm{g}} = \{\mathsf{D_k(Q') := Q}\} \quad \text{for } Q' \text{ fresh} \\
\mathcal{S}[\![K]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v} \\
&\quad \texttt{where } P = \mathsf{D_k}(K), \\
&\qquad\qquad \rho' = \rho\{visit[K_p] \mapsto true, entry[K_p] \mapsto v\} \\
\mathcal{S}[\![\alpha.[\tilde{a} := \tilde{E}]P]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![\alpha.P]\!]_{\rho'}^{k,p,v} \\
&\quad \texttt{where } \rho' = \rho\{upd \mapsto \rho(upd) \cup [\tilde{a} := \tilde{E}]\} \\
\mathcal{S}[\![\alpha.K]\!]_{\rho}^{k,p,v} &= \begin{cases} \mathcal{S}_{rec}\alpha & \text{if } \rho(visit)[K_p] = \text{true} \\ \mathcal{S}_{nom}\alpha; \mathcal{S}[\![K]\!]_{\rho'}^{k,p,\mathsf{cnt[p]}} & \text{otherwise} \end{cases} \\
&\quad \texttt{where } \rho' = \rho\{upd \mapsto \emptyset, aware \mapsto \emptyset, parent \mapsto \emptyset\} \\
\mathcal{S}[\![\alpha.P]\!]_{\rho}^{k,p,v} &= \mathcal{S}_{nom}\alpha; \mathcal{S}[\![P]\!]_{\rho'}^{k,p,\mathsf{cnt[p]}} \\
&\quad \texttt{where } \rho' = \rho\{upd \mapsto \emptyset, aware \mapsto \emptyset, parent \mapsto \emptyset\} \\
\mathcal{S}_{nom}(\tilde{E})@\Pi &\equiv (\downarrow_{g})\mathcal{B}[\![(\tilde{E})\Pi]\!]_{\rho}^{k,p,v,\mathsf{cnt[p]}} \\
&\quad \texttt{where } \downarrow_{g} = \{\mathsf{cnt[p]} := \mathsf{cnt[p]} + 2\} \\
\mathcal{S}_{nom}\Pi(\tilde{x}) &\equiv (\downarrow_{g})\mathcal{B}[\![\Pi(\tilde{x})]\!]_{\rho}^{k,p,v,\mathsf{cnt[p]}} \\
&\quad \texttt{where } \downarrow_{g} = \{\mathsf{cnt[p]} := \mathsf{cnt[p]} + 1\} \\
\mathcal{S}_{rec}(\tilde{E})@\Pi &\equiv (\downarrow_{g})\mathcal{B}[\![(\tilde{E})\Pi]\!]_{\rho}^{k,p,v,\rho(entry)[K_p]} \\
&\quad \texttt{where } \downarrow_{g} = \{\mathsf{cnt[p]} := \mathsf{cnt[p]} + 1\} \\
\mathcal{S}_{rec}\Pi(\tilde{x}) &\equiv \mathcal{B}[\![\Pi(\tilde{x})]\!]_{\rho}^{k,p,v,\rho(entry)[K_p]}
\end{aligned}
$$

FIGURE 4.6: Structural translation of processes: semicolon (;) denotes the completion of the left translation before starting the right one, a global update ($\downarrow_g$) takes place before the follow-up translation, α denotes either an *AbC* input or output action, $\equiv$ denotes a textual expansion of the left hand side

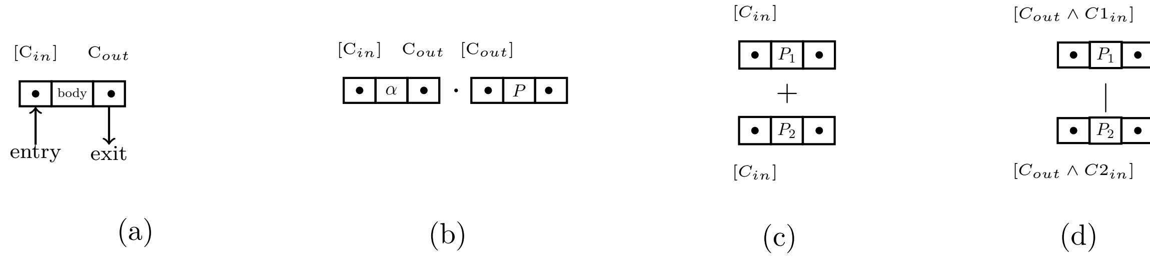

FIGURE 4.7: Structural mapping to combine generated UMC transitions: (a) graphical representation of a transition; (b) an action prefixing process $\alpha.P$ has the entry point of α as $\mathtt{C_{in}}$, and the entry point of P as the exit point of α ($\mathtt{C_{out}}$); (c) a choice process $P_1 + P_2$ has the same entry point on both sub-processes P_1, P_2, which may be the exit point of a previous action (if there any); (d) the entry points of sub-processes P_1, P_2 in a parallel process $P_1|P_2$ contain also the exit point of a previous action (if there any)

prefixed by any action. The translation of $Q|^m P$ is the translation of $Q'|P$ where $Q' \triangleq Q$ and P is a parallel composition of m processes Q'.

Process call. The translation of a process call K first looks up the definition P of K in the set of process declarations $\mathsf{D_k}$, and then returns a translation of P under the new map ρ', in which the entry point of K_p is stored and K_p is marked as visited.

Update. The translation of a update process $\alpha.[\tilde{a} := \tilde{E}]P$ accumulates the update descriptor $[\tilde{a} := \tilde{E}]$ into variable *upd* and returns the translation of $\alpha.P$ under the new map.

Action-prefixing. In an action prefix process, the continuation process can be either a process call K, i.e., a name or a process code P. Separating them out helps the translation in dealing with recursion. In particular, there are two cases:

- The translation of $\alpha.K$ is a “recursive” translation of α if the name K, indexed by p has been visited before, otherwise, it is a “normal” translation of α followed by a translation of K.
- The translation of $\alpha.P$ (where P is not a name) is a “normal” translation of α followed by a translation of P.

In a “normal” translation of α, denoted by $\mathcal{S}_{nom}\alpha$ in Figure 4.6, the global counter $\mathsf{cnt[p]}$ is increased by 2 for an output action and by 1 for an input action. This is because we will need two UMC transitions for modeling an output action and one for an input action. In a “recursive” translation of α, denoted by $\mathcal{S}_{rec}\alpha$ in Figure 4.6, however, the global counter $\mathsf{cnt[p]}$ is increased by 1 for an output action.

The actual UMC code for the α is then generated by the *behavioural translation* $\mathcal{B}$. It has the form of $\mathcal{B}[\![\alpha]\!]_{\rho}^{k,p,c_{in},c_{out}}$, where the parameter set is enriched with an exit point c_{out}. c_{out}, in case of a recursion,

receives the entry point of the recursive process K, i.e., $\rho(entry)[K_p]$, otherwise, the updated value of cnt[p].

Finally, the translation of the continuation process, either K or P is parameterised with a new map ρ' where the previous accumulated information is reset, and a new entry value, calculated from the value of global counter for P - cnt[p], is added.

4.1.2.2 Behavioural mapping.

We now describe the function $\mathcal{B}[\![\alpha]\!]_{\rho}^{k,p,c_{in},c_{out}}$ which generates the actual UMC code for a specific action α according to the information accumulated in the map ρ and the parameter set including component index k, process index p, the entry and exit points associated to this action c_{in} and c_{out}.

Output Action. We model the output action in two steps that are forced to occur in a strict sequence: the sending to self (i.e., the state machine) of the `bcast` event (that will be dispatched to all the parallel components), and the discarding of this very message, as illustrated by the code snippet in Figure 4.8. The global variable `receiving` of the state machine works as a lock, to guarantee the correct ordering of the two transitions. Here the (main) transition is guarded by the following

```
B[[(Ẽ)@Π]]ρ^{k,p,c_in,c_out} =
SYS.Ck.s0 -> Ck.s0 {
    allowsend(i)[i=k & receiving=false & pc[k][p]=c_in & [[ρ(parent)]] & [[ρ(aware)]]]/
    tgt: int[];
    for j in 0..pc.length-1 {
      if ([[Π]]) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    receiving:=true;
    self.bcast(tgt,[[[Ẽ]]],k);
   OUT.send($2,msg);
    [[ρ(upd)]];
    pc[k][p] = c_in +1;
}
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[pc[k][p] = c_in +1]/
    receiving:=false;
    self.allowsend(k);
    pc[k][p] = c_out;
}
```

FIGURE 4.8: Translation of an output action

conditions:

- `i = k` means that the component k gets scheduled;
- `receiving = false` means that there is no other components performing an output action;

- $\mathtt{pc[k][p]} = \mathtt{c_{in}}$ means that the program counter holds the execution point of this action, so the action is ready to be executed;
- and (possibly) conditions on the exit point of the parent process, and the satisfaction of awareness predicate accumulated in the map ρ. Recall that $\rho(parent)$, if not empty, is a pair of index p' and exit point v'. Its translation has the form $\mathtt{pc[k][p']} = \mathtt{v'}$. The translation for awareness predicate is detailed later.

The transition body includes the computation of the set of potential receivers `tgt`, a sending operation in the form of `self.bcast(tgt, Msg, k)` where Msg stands for the result of translating expressions $\tilde{E}$, and the translation of attribute updates stored in $\rho(upd)$. This transition also sets the global variable `receving` to disable "sending" transitions of other components, thus only allowing transitions triggered by `bcast(tgt, msg, k)`.

Finally, the complementary transition follows resets `receiving` and updates the program counter of the current process to $\mathtt{c_{out}}$. In addition, it pushes the signal `allowsend(k)` to the event queue of the state machine.

Input Action. An input action is translated into a transition whose general form is shown in Figure 4.9. The transition is triggered by signal `bcast(tgt, msg, j)` issued from some sender `j`. It is enabled, for a component k (if k is in the target set `tgt` of receivers), when the receiving predicate Π and (possibly) the preceding awareness predicates are satisfied. Variable binding is done by assigning the received message `msg` to vector `bound` indexed by k and p. Similarly to the output action, the transition guard can contain awareness predicates; the transition body can contain update commands.

$\mathcal{B}[\![\Pi(\tilde{x})]\!]_{\rho}^{k,p,c_{in},c_{out}} =$

```
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[tgt[k]=1 & pc[k][p]= c_in & [[ρ(parent)]] & [[ρ(aware)]] & [[Π]]]/
    bound[k][p][0] = msg[0];
    bound[k][p][1] = msg[1];
    ...
    [[ρ(update)]];
    OUT.received(k,msg);
    pc[k][p] = c_out;
}
```

Figure 4.9: Translation of an input action

Other translations. In the above generated transitions, we have smaller translations for specific *AbC* terms such as expressions, predicates. Theses are translated into UMC code by a family of functions $[\![\cdot]\!]$ whose general definitions are given in Figures 4.10, 4.11.

$$
\begin{aligned}
[\![true]\!] &= \mathsf{true} \\
[\![false]\!] &= \mathsf{false} \\
[\![E_1 \bowtie E_2]\!] &= [\![E_1]\!] \bowtie [\![E_2]\!] \\
[\![\Pi_1 \; and \; \Pi_2]\!] &= [\![\Pi_1]\!] \; \mathsf{and} \; [\![\Pi_2]\!] \\
[\![\Pi_1 \; or \; \Pi_2]\!] &= [\![\Pi_1]\!] \; \mathsf{or} \; [\![\Pi_2]\!] \\
[\![not \; \Pi]\!] &= \mathsf{not} \; [\![\Pi]\!]
\end{aligned}
$$

FIGURE 4.10: Translation of predicates

$$
\begin{aligned}
[\![v]\!] &= v \\
[\![x_i]\!] &= \mathsf{msg}[i] \\
[\![\$x_i]\!] &= \mathsf{bound}[k][p][i] \\
[\![a]\!] &= \begin{cases} a[j] & \text{if a appears in interaction predicates} \\ [\![this.a]\!] & \text{otherwise} \end{cases} \\
[\![\text{this}.a]\!] &= a[k] \\
[\![E.hd]\!] &= [\![E]\!].\mathsf{head} \\
[\![E.tl]\!] &= [\![E]\!].\mathsf{tail} \\
[\![E_1[E_2]]\!] &= [\![E_1]\!].[[\![E_2]\!]] \\
[\![|E|]\!] &= |[\![E_2]\!]| \\
[\![E_1 \circ E_2]\!] &= [\![E_1]\!] \circ [\![E_2]\!]
\end{aligned}
$$

FIGURE 4.11: Translation of expressions

Notice the case of an attribute name `a`, the translation depends on the context. If `a` appears in the interaction predicates (send and receive), its value is of a remote component, and thus is translated into `a[j]`. Otherwise, `a` is a local attribute and thus is translated into `a[k]`.

The translation of a variable `x` calculates its position `i` in the vector msg and then returns `msg[i]`. The translation of a value `$x` calculates its position `i` in the vector `bound` and then returns `bound[k][p][i]` where k, p are indexes of the corresponding component and process.

Also note that the translation for operators on sets $in, notin, ++, --$ is more involved since UMC does not support those binary operators. In such cases, the translation makes use of loops and conditional branching (supported by UMC) to properly simulate their semantics. Furthermore, if a designer wishes to use more complex expressions, e.g., functions on sub expressions, that can not be directly represented using our supported syntax, she would need to instrument the generated UMC code herself to encode such expressions (or predicates).

It is easy to see why our translation terminates. The presented translation rules work on syntactic categories of AbC terms, decomposing them and translating each AbC action as a basic unit. Since we do not allow infinite process definitions, the structural mapping function $\mathcal{S}$ on a process stops when it meets either a recursive call ($\mathcal{S}_{rec}$) or a nil process ($\mathcal{S}[\![nil]\!]$).

Example 4.2 This example shows how structural mapping works on a man component in the stable marriage problem, presented in Example 4.1. If we take the first declared component C1

$$\Gamma_0 : \langle P_{init_0}, D_0 \rangle$$

then the set of relevant variables at the beginning is as follows:

- P_{init_0} = M
- D_0 = {M → α_1.u$_1$M1, M1 → α_2.u$_2$M} where α_1 = ('propose')@(id = this.pref.hd), α_2 = (x ='no')(x) and u$_1$ = [partner := pref.hd, pref := pref.tl], u$_2$ = [partner := 0]
- k = 0, pid = 0, cnt[0] = 1
- $\rho = \{visit \rightarrow [], entry \rightarrow [], aware \rightarrow \emptyset, parent \rightarrow \emptyset, update \rightarrow \emptyset\}$

The structural mapping takes P_{init_0}, ρ, the parameters $k = 0, p = \mathsf{pid}, v = \mathsf{cnt}[0]$ and proceeds as:

$$\begin{aligned}
\mathcal{S}[\![M]\!]^{0,0,1}_{\rho} &= \mathcal{S}[\![D_0(M)]\!]^{0,0,1}_{\rho} \\
&= \mathcal{S}[\![\alpha_1.u_1M1]\!]^{0,0,1}_{\rho_0} \text{ where } \rho_0 = \rho\{visit[M_0] \mapsto true, entry[M_0] \mapsto 1\} \\
&= \mathcal{S}[\![\alpha_1.M1]\!]^{0,0,1}_{\rho_1} \texttt{ where } \rho_1 = \rho_0\{upd \mapsto u_1\} \\
&= \mathcal{S}_{nom}\alpha_1; \mathcal{S}[\![M1]\!]^{0,0,\mathsf{cnt}[0]}_{\rho_2} \texttt{ where } \rho_2 = \rho_1\{upd \mapsto \emptyset\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_{nom}\alpha_1 &\equiv (\downarrow_g)\mathcal{B}[\![(\text{`}propose')@(id = this.pref.hd)]\!]^{0,0,1,\mathsf{cnt}[0]}_{\rho_1} \text{ where } \downarrow_g = \{\mathsf{cnt}[0] := \mathsf{cnt}[0] + 2\} \\
&= \mathcal{B}[\![(\text{`}propose')@(id = this.pref.hd)]\!]^{0,0,1,3}_{\rho_1} \qquad /{*}\mathsf{cnt}[0] = 3 \text{ after update } {*}/
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![M1]\!]^{0,0,\mathsf{cnt}[0]}_{\rho_2} &= \mathcal{S}[\![D_0(M1)]\!]^{0,0,3}_{\rho_2} \qquad /{*}\mathsf{cnt}[0] = 3 \text{ after update } {*}/ \\
&= \mathcal{S}[\![\alpha_2.u_2M]\!]^{0,0,3}_{\rho_2} \\
&= \mathcal{S}[\![\alpha_2.M]\!]^{0,0,1}_{\rho_2'} \texttt{ where } \rho_2{}' = \rho_2\{upd \mapsto u_2\} \\
&= \mathcal{S}_{rec}\alpha_2 \qquad /{*}\rho_2'(visit)[M_0] = \text{ true } {*}/ \\
&\equiv \mathcal{B}[\![(x = \text{`}no')(x)]\!]^{0,0,3,\rho_2'(entry)[M_0]}_{\rho_2'} = \mathcal{B}[\![(x = \text{`}no')(x)]\!]^{0,0,3,1}_{\rho_2'}
\end{aligned}$$

The behavioural mapping $\mathcal{B}$ for actions α_1 and α_2 can then produces the actual UMC code in the sequence (Figure 4.12), by using the accumulated information in the parameter set.

Example 4.3 This example shows how structural mapping works on a woman component in the stable marriage problem, presented in Example 4.1. If we take the first declared component C4

$$\Gamma_3 : \langle W, D_3 \rangle$$

$\mathcal{B}[\![(propose, this.id)@(id = this.pref.hd)]\!]^{0,0,1,3}_{\rho_1} =$

```
SYS.C0.s0 -> C0.s0 {
    allowsend(i)[i=0 & receiving=false & pc[0][0]=1]/
    tgt: int[];
    for j in 0..pc.length-1 {
        if (id[j] = pref[0].head) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    receiving:=true;
    self.bcast(tgt,[propose,id[0]],0);
    partner[0] := pref[0].head;
    pref[0] := pref[0].tail;
    pc[0][0] = 2;
}
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[pc[0][0] = 2]/
    receiving:=false;
    self.allowsend(0);
    pc[0][0] = 3;
}
```

$\mathcal{B}[\![(x = no)(x)]\!]^{0,0,3,1}_{\rho'_2} =$

```
SYS.C0.s0 -> C0.s0 {
    bcast(tgt,msg,j)[tgt[0]=1 & pc[0][0]= 3 & msg[0] = no]/
    bound[0][0][0] := msg[0];
    partner[0] := 0;
    pc[0][0] = 1;
}
```

FIGURE 4.12: The generated code for a Man component

then the set of relevant variables at the beginning is as follows:

- $D_3 = \{\mathrm{W} \to W'|W'|W', W' \to \beta.H, H \to a_1\beta_1.u_1 nil + a_2\beta_2.nil\}$ where
  β = (x=‘propose’)(x,y),
  $a_1 = \langle pref[partner] < pref[\$y]\rangle$, $u_1 = [partner := \$y]$ and
  $a_2 = \langle pref[partner] > pref[\$y]\rangle, \beta_2 = (‘no’)@(id = \$y)$.
- $k = 3, \mathsf{pid} = 0, \mathsf{cnt}[0] = 1$
- $\rho = \{visit \to [], entry \to [], aware \to \emptyset, parent \to \emptyset, update \to \emptyset\}$

The structural mapping takes W, ρ, the parameters $k = 3, p = \mathsf{pid}, v = \mathsf{cnt}[0]$ and proceeds as below.

$$
\begin{aligned}
\mathcal{S}[\![W]\!]^{3,0,1}_{\rho} &= \mathcal{S}[\![W']\!]^{3,0,1}_{\rho}; (\downarrow_g)\mathcal{S}[\![W']\!]^{3,\mathsf{pid},\mathsf{cnt[pid]}}_{\rho}; (\downarrow_g)\mathcal{S}[\![W']\!]^{3,\mathsf{pid},\mathsf{cnt[pid]}}_{\rho} \\
&\qquad \texttt{where } \downarrow_g = \{\mathsf{pid} := \mathsf{pid}+1, \mathsf{cnt[pid]} := 1\} \\
&= \mathcal{S}[\![W']\!]^{3,0,1}_{\rho}; \mathcal{S}[\![W']\!]^{3,1,1}_{\rho}; \mathcal{S}[\![W']\!]^{3,2,1}_{\rho} \\
\mathcal{S}[\![W']\!]^{3,0,1}_{\rho} &= \mathcal{S}[\![\beta.H]\!]^{3,0,1}_{\rho} \\
&= (\downarrow_g)\mathcal{B}[\![(x = \text{`}propose\text{'})(x,y)]\!]^{3,0,1,\mathsf{cnt[0]}}_{\rho}; \mathcal{S}[\![H]\!]^{3,0,\mathsf{cnt[0]}}_{\rho} \ \texttt{where } \downarrow_g = \{\mathsf{cnt[0]} := \mathsf{cnt[0]}+1\} \\
&= \mathcal{B}[\![(x = \text{`}propose\text{'})(x,y)]\!]^{3,0,1,2}_{\rho}; \mathcal{S}[\![H]\!]^{3,0,2}_{\rho} \\
\mathcal{S}[\![H]\!]^{3,0,2}_{\rho} &= \mathcal{S}[\![P1]\!]^{3,0,2}_{\rho}; \mathcal{S}[\![P2]\!]^{3,0,2}_{\rho} \\
&= \mathcal{S}[\![\beta_1.nil]\!]^{3,0,2}_{\rho_1}; \mathcal{S}[\![\beta_2.nil]\!]^{3,0,2}_{\rho_2} \\
&\qquad \texttt{where } \rho_1 = \rho\{aware \mapsto a_1, upd \mapsto u_1\}, \rho_2 = \rho\{aware \mapsto a_2\} \\
&= (\downarrow_g)\mathcal{B}[\![(\text{`}no\text{'})@(id = this.partner)]\!]^{3,0,2,\mathsf{cnt[0]}}_{\rho_1}; (\downarrow_g)\mathcal{B}[\![(\text{`}no\text{'})@(id = \$y)]\!]^{3,0,2,\mathsf{cnt[0]}}_{\rho_2} \\
&\qquad \texttt{where } \downarrow_g = \{\mathsf{cnt[0]} := \mathsf{cnt[0]}+2\} \\
&= \mathcal{B}[\![(\text{`}no\text{'})@(id = this.partner)]\!]^{3,0,2,4}_{\rho_1}; \mathcal{B}[\![(\text{`}no\text{'})@(id = \$y)]\!]^{3,0,2,6}_{\rho_2} \\
\mathcal{S}[\![W']\!]^{3,1,1}_{\rho} &= \ldots \qquad \text{/* a similar translation of this instance */} \\
\mathcal{S}[\![W']\!]^{3,2,1}_{\rho} &= \ldots \qquad \text{/* another similar translation */}
\end{aligned}
$$

Here, we omit the translation details for other parallel instances of the code W' and the generated UMC code by the behavioural translation $\mathcal{B}$.

The above examples illustrated the use of structural mapping in dealing with prefixing action, process recursion, non-deterministic choice and parallel composition. Figure 4.13 shows entry and exit points of the generated transitions. They are characterized by corresponding actions α, β and are connected via a match on the exit point of one action with the entry point of other action. We subscript execution points in Figure 4.13 (b) to denote the process of index 0 of a Woman component. The other two instances have the similar values on their transitions. However, their executions would be independent due to the difference in process indexes.

To conclude this section, we briefly explain the other translation parts in Figure 4.5. The function $\mathcal{A}$ simply aggregates all attribute names of all components and output empty attribute vectors, one for each attribute (for declaration purpose). The function $\mathcal{V}$ instead declares attribute vectors and associated initial values, obtained from components instantiations. The code of a component k is therefore able to access its local attribute values by using its index k. Finally, the program counter pc is a vector of 1's, one for each component. For a component k with the maximum process index pid, $pc[k]$ has the length of $\mathsf{pid} + 1$.

FIGURE 4.13: Transitions are glued by matching entry and exit points: (a) transitions of a man component; (b) transitions of a woman component (for one process instance)

4.1.3 Implementation and Optimizations

We have developed a tool implementing the above translation rules in Erlang. The tool [1] contains about 1000 lines of Erlang code, in addition to 200 lines of grammar rules for parsing AbC-like specifications. It is worth mentioning that we have also added some optimizations to the basic translation scheme. These include:

- Eliminating dead variables: Every process uses a vector $bound[k][p]$ to store the received messages. This vector can be reset to its initial value, i.e., 0 in the "last" translated transition, whenever the translation encounters a *nil* or a recursion.
- Handling empty send: Associated examples come with the AbC calculus [21, 22] often use the construct $[\tilde{a} := \tilde{E}]()$@(false) in order to model updating attributes $\tilde{a}$. In this case, the translation can create one transition (instead of two) that performs only updates. Likewise, in case of a normal send in which the computed set of potential receivers (i.e., tgt) is empty, the main transition of an output can set the program counter to be the exit point in order to skip the second complementary transition.
- Scheduling parallel instances: the use of $|^n$ creates parallel processes executing the same code. For example, P := Q |^(n-1) P generates n copies of Q. This implies that it does not matter which copy Q_i performs the first action. Our implementation sets an extra guard on the first action of these parallel processes, so that a specific instance Q_j can move only if a "previous" instance Q_{j-1} has already moved.

4.2 Case studies

We apply our verification methodology to algorithmic solutions of the two case studies, namely stable marriage and graph coloring problems in order to check a selection of properties of interest. For each solution, we provide a short informal description along with the resulting formal specifications in AbC.

Similarly, we present a number of properties first informally and then as a precise logical property of the state machines generated from the formal specifications. We show how to instrument these state machines for property checking. As we go along, we also consider a few additional program-specific properties that we used as a guidance to refine the formal specifications themselves.

4.2.1 Stable Marriage

The Stable Marriage Problem (SMP) [65] is a well studied problem that has applications in a variety of real-world situations, such as assigning students to colleges or appointing graduating medical students to their first hospital. SMP that has been initially formulated in terms of peers that make offer to potential partners by taking into account a preference list is easily adaptable to a context in which partners are selected according to their attributes. Indeed, due to its simple formulation and its intrinsically concurrent nature, SMP has been already used to show the advantages of AbC as a very high-level formalism to describe complex systems [46, 47]. In this section, we use it to show how our framework can be used to reason about properties of attribute-based systems.

Our first algorithm implements the classical solution, where preferences are represented as complete ordered lists of identifiers. Initiators and responders are programmed as individual processes that interact using their local preference lists in a point-to-point fashion using their identity. The AbC system for this algorithm has been given in the Example 4.1.

The other two programs adapt the classical solution to the context of attribute-based communication, where partners are selected by considering predicates over the attributes of the potential partners. The two new solutions differ for the way initiators choose their potential partners. They can start by either making proposals to the responder they prefer most and then relax their expectations or making proposals with the lowest requirements, to make sure to get a partner, and gradually increase their expectations.

Top-down Strategy. In this case preferences are expressed as predicates over the attributes of partners rather than as lists of people. For example, a person might be interested in finding a partner from a specific country who speaks a specific language. A suitable communication predicate would be $country = \text{this}.favcountry \ \wedge \ language = \text{this}.favlanguage$, where $language$ and $country$ are two attributes of initiators and responders, and $favcountry$ and $favlanguage$ are used to express preferences.

In the solution to this variant of SMP, the initiator starts by making offers to responders that satisfy its highest requirements, i.e., have all wanted attributes. In case nobody satisfy these requirements, the initiator retries after weakening the predicate by eliminating one of the preferred attributes and waits for a reaction. The system then evolves as follows.

```
component Man
  attributes: id, m1, m2, pw1, pw2, partner, bl, send, c, counter, n, n2, n3, n4
  observables: partner
  behaviour:
        let {
        Q1 := ...
        Q2 := ...
        Q3 := ...
        Q4 := ...
        P := ...
        A := ...
        R := ...
           }
        init  Q1 | Q2 | Q3 | Q4 | P | R | A
end
```

FIGURE 4.14: A template for Man component

A single initiator that receives a *yes* considers himself engaged and sends out a *confirm* message; it keeps proposing if a *no* is received. An engaged initiator that receives a *yes* notifies the interested responder that meanwhile another partner has been found by sending a *toolate* message. An engaged initiator dropped by its current partner with a *bye* message restarts immediately proposing.

An engaged responder reacts upon receiving a proposal by comparing the new initiator with the current partner. If the new proposer is not better, it will receive a *no* message. Otherwise, the responder sends a *yes* to notify the proposer her availability, and waits for a reply. Upon receiving a *confirm*, the responder changes partner and sends *bye* to the ex partner; in case a *toolate* message is received the responder continues without changes.

Essentially this is the matching algorithm presented in Section 2.2.1, Chapter 2. Here we will encode it using our template.

AbC Specification. We model a scenario where each agent exposes two characteristics besides their identifiers: $\mathtt{id}, \mathtt{m1}, \mathtt{m2}$ for proposers and $\mathtt{id}, \mathtt{w1}, \mathtt{w2}$ for responders. Furthermore, participants have their own preferences on which are modeled by $\mathtt{pw1}, \mathtt{pw2}$ and $\mathtt{pm1}, \mathtt{pm2}$. The set of preferences for each agent is also the set of predicates it has over partner characteristics. In particular, we will consider 4 predicates to express the matching intention. For example, the set of predicates that an initiator uses to find partner is derived in the following order: $\Pi_1 = (\mathtt{pw1} = \mathtt{w1} \text{ and } \mathtt{pw2} = \mathtt{w2})$, $\Pi_2 = (\mathtt{pw1} = \mathtt{w1})$, $\Pi_3 = (\mathtt{pw2} = \mathtt{w2})$, $\Pi_4 = \mathtt{true}$. Likewise, each responder uses a similar mechanism to compare two any proposers in order to select a better one.

In our template, the `Man` component has the form as shown in Figure 4.14. The set of attributes include in addition: the current `partner`, initially set to 0 which is also made observable for later analysis, `bl` is a set of responders that a proposer does not want to contact, initially the empty set. `send` is used to control making proposals. `c` is the number of responders who are willing to answer the 'propose' message (for each preference), $\mathtt{counter}, \mathtt{n}, \mathtt{n2}, \mathtt{n3}, \mathtt{n4}$ are helper variables used to calculate

```
Q1 := <counter = 0 and c = |bl|>
       ('q1', this.id, this.pw1, this.pw2)@(id notin this.bl).
       [counter := counter + |bl|]Q11

Q11 := <counter < n>(x = 'interest')(x).
        [c := c + 1, counter := counter + 1]Q11
        +
       <counter < n>(x = 'uninterest')(x).
        [counter := counter + 1]Q11

Q2 :=  <counter = n and c = |bl|>
        ('q2', this.id, this.pw1, this.pw2)@(id notin this.bl).
        [counter := counter + |bl|]Q22

Q22 := <counter < n2>(x = 'interest')(x).
        [c := c + 1, counter:= counter + 1]Q22
        +
       <counter < n2>(x = 'uninterest')(x).
        [counter := counter + 1]Q22

Q3 := <counter = n2 and c = |bl|>
       ('q3', this.id, this.pw1, this.pw2)@(id notin this.bl).
       [counter := counter + |bl|]Q33

Q33 := <counter < n3>(x = 'interest')(x).
        [c := c + 1, counter:= counter + 1]Q33
        +
       <counter < n3>(x = 'uninterest')(x).
        [counter := counter + 1]Q33

Q4 := <counter = n3 and c = |bl|>()@(false).
       [counter := counter + 1]nil
```

FIGURE 4.15: Querying processes

c. Initially, counter $= 0$, n is the problem size (i.e., the total number of responders) while n2, n3, n4 stands for values $n*2, n*3$ and $n*3+1$[1], respectively.

The initial behaviour, as indicated in the code is a parallel composition of processes Q(s) for querying a given matching intention, process P used to make proposals, and processes R, A used for handling replies.

We first describe querying processes Q used for querying the number of interesting women (responders) in the system. Their code share the same structure, as reported in Figure 4.15. A query message is sent to all women (n) except those in bl, triggered by conditions on counter and c. Replies messages are expected of either 'interest' or 'uninterest' for which counter is increased while c only counts the number of 'interest' messages. The number of replies for a given query would be n $-$ |bl|, thus counter is also increased by the size of the set bl. For example, Q_1 sends a message tagged with 'q1' denoting the current level of preferences that the proposer is asking for. Other messages elements are proposer *id* (so that a responder can send reply), and his two preferences $pw1, pw2$ (so that a responder can answer if its characteristics match this requirement ('interest') or not ('uninterest')).

[1]UMC does not allow arithmetic calculations inside transition guards

```
P := <counter = n and c > |bl| and partner = 0 and send = 0>
      ('propose', this.id, this.m1, this.m2)@(w1 = this.pw1 and w2 = this.pw2 and id
   notin this.bl).[send := 1]P
    +
      <counter = n2 and c > |bl| and partner = 0 and send = 0>
      ('propose', this.id, this.m1, this.m2)@(w1 = this.pw1 and id notin this.bl).[send
    := 1]P
    +
    <counter = n3 and c > |bl| and partner = 0 and send = 0>
     ('propose', this.id, this.m1, this.m2)@(w2 = this.pw2 and id notin this.bl).[send
   := 1]P
    +
    <counter = n4 and n > |bl| and partner = 0 and send = 0>
     ('propose', this.id, this.m1, this.m2)@(id notin this.bl).[send := 1]P
```

FIGURE 4.16: Proposing process

```
A := (x = 'yes')(x,y).H |^(n-1) A
H := <partner = 0>('confirm')@(id = $y).[partner := $y]A
     +
     <partner > 0>('toolate')@(id = $y).A

R := (x = 'bye')(x,y).[partner := 0, bl := bl ++ [$y], send := 0]R
     +
     (x = 'no')(x,y).[bl := bl ++ [$y], send := 0]R
```

FIGURE 4.17: Handling replies processes

After querying for the requirement 'q1', c is the number of interesting responders which will be used by the proposing process P (presented below). A new query messages with a lower level, e.g, 'q2' will be sent if $c = |bl|$, which means the proposer could not find any partner at the level 'q1'.

Process Q4 deserves a little more explanation. It handles the last requirement in which all the specific requirements over partners attributes have been considered, yet the proposer is still single (i.e., the condition counter = n3 and c = |bl| holds). In this case, there is no query message is sent out. The attribute counter is increased by 1, making its value into $n4$, so that P can exploit this fact for making proposals with no requirement (i.e., a true predicate).

Process P (Figure 4.16), guarded by two conditions partner = 0 and send = 0, becomes active when a single proposer has not yet sent a proposal. After that, it sets the flag send and continues as P. To model the adaptive behaviour of proposers needed to relax their preferences, we need to enrich P with the condition c > |bl| where bl is a set of responders that the proposer does not want to contact. This set is updated when a proposer receives a 'no' or a 'bye' message. This allows them to know when to relax their requirements.

A proposer may receive multiple replies and takes care of them according to the message type. In particular, process A handles 'yes' messages while R handles 'bye' and 'no' messages, as shown in Figure 4.17. A is composed by n parallel instances of (x = 'yes')(x, y).H. Process R, due to containing one single action and its recursive behaviour, needs no parallel instances. On receiving a ′yes' message,

```
component Woman
    attributes: id, w1, w2, pm1, pm2, partner, cm1, cm2, bl, lock, n
    observables: partner
    behaviour:
        let {
            P1 := (x = 'q1')(x,y,z,t).H1 |^(n-1) P1
            P2 := (x = 'q2')(x,y,z,t).H2 |^(n-1) P2
            P3 := (x = 'q3')(x,y,z,t).H3 |^(n-1) P3
            W :=
            }
         init P1 | P2 |  P3 | W
end
```

FIGURE 4.18: A template for the component Woman

```
H1 := <$z = w1 and $t = w2>('interest')@(id = $y).nil
        +
      <not ($z = w1 and $t = w2)>('uninterest')@(id = $y).nil

H2 := <$z = w1>('interest')@(id = $y).nil
        +
      <not ($z = w1)>('uninterest')@(id = $y).nil

H3 := <$t = w2>('interest')@(id = $y).nil
        +
      <not ($t = w2)>('uninterest')@(id = $y).nil
```

FIGURE 4.19: Responding to queries

a single proposer sends back a 'confirm' and updates the new partner, otherwise he sends a 'toolate' message. On receiving a 'bye' or a 'no', a proposer adds the id of the sender to bl because there is no way he could get this sender as partner. The attribute send is then reset to possibly trigger the sending of a 'propose' message in P. Furthermore, in case of receiving a 'bye', the proposer becomes single and thus the attribute partner is reset to 0.

The definition of a component Woman, representing for responders has the general form, presented in Figure 4.18. The set of attributes for a responder includes its id, characteristics w1, w2, preferences over partner characteristics pm1, pm2; in addition to the current partner and characteristics cm1, cm2. Attribute bl is a set storing the identities of proposers from which a responder does not want to consider 'propose' messages. Attribute lock is used to sequentialize the processing of parallel 'propose' messages. Initially, bl is an empty set, $\texttt{partner} = 0, \texttt{lock} = 0, \texttt{cm1} = \texttt{cm2} = -1$, n is the problem size, i.e, the number of proposers in the system.

The initial behaviour of a woman, including handling querying messages in parallel with answering proposal messages. First of all, it is necessary to have three processes P(s) to handle three different types of querying messages that may come from proposers. As shown in Figure 4.18, these processes input messages whose the first elements are 'q1', 'q2', 'q3', respectively. Each of them then proceeds with a corresponding behaviour Hs whose definitions are elaborated in Figure 4.19.

```
W := (x = 'propose' and id notin this.bl)(x,y,z,t).
      [bl := bl ++ [$y]](A + R) |^(n-1) W
A := <(partner = 0 or
        ($z = pm1 and cm1 /= pm1) or ($z = cm1 and $t = pm2 and cm2 /= pm2))
        and lock = 0>
       ('yes',this.id)@(id = $y).[lock := 1]A1
A1 := (x = 'confirm')(x).('bye',this.id)@(id = this.partner).
       [partner := $y, cm1 := $z, cm2 := $t, lock := 0]W
        +
        (x = 'toolate')(x).[lock := 0, bl := bl -- [$y]]W
R := <not (partner = 0 or
       ($z = pm1 and cm1 /= pm1) or ($z = cm1 and $t = pm2 and cm2 /= pm2))>
        ('no', this.id)@(id = $y).W
```

FIGURE 4.20: The main behaviour of a woman component

Basically, messages to be answered by processes `H(s)` may be of '`interest`' or '`uninterest`', depending on the query type and on the satisfaction of their awareness predicates.

The main behaviour of a responder is specified by `W` (Figure 4.20). A responder is willing to listen to '`propose`' messages from people who are not in the set `bl`, as indicated in the receiving predicate of `W`. On receving such a message, the sender `id` bound to variable `y` is added to the set `bl`. The parallel composition with the same process $|\hat{}(n-1)$ `W` guarantees that other '`propose`' messages would not be lost. Then, for each message, a responder can proceed either like `A` (accept) or `R` (reject).

In the process `A`, the awareness predicate encodes the condition that the new comer is better than the current partner by comparing their attributes with respect to the responder's preferences. If this is the case and if there is no other pending processing, i.e., `lock` = `0`, the responder replies with a '`yes`', setting the `lock` to 1 to wait for a decision from the same proposer.

The continuation `A1` encodes this behaviour. If the proposer decides to match by sending a '`confirm`', the responder rejects the current `partner`, updates the proposer as the new partner (together with his characteristic) and gets rid of the critical region (i.e., via setting `lock` = `0`). In the other case where the proposer replies a '`toolate`' message, the responder returns the `lock` and remove the proposer from `bl` to allow him to propose again later.

In the process `R`, the new comer is not better than the current partner and thus is sent a '`no`' message. Both `A` and `R` recursively call `W` after their processing so to reuse the behaviour of `W`[2].

Bottom-up Stable Marriage. We have also experimented with another approach to attribute-based stable marriage, where men start proposing to the women they like less, and try to incrementally improve their level of satisfaction by continuously proposing themselves even after finding a partner in the attempt to find someone they like more then their current partner. In this case both men and women can be dropped by their current partner if a more desirable option shows up.

[2]One alternative is to terminate them with a nil process. In this case, it would be necessary to specify the number of parallel instances of W as n*(n-1) - the maximum number of 'propose' messages can arrive in a system of n proposers.

This strategy assumes that lower expectations of men include the higher ones. This is indeed the case of our relaxation scheme, in which a strong preference $\Pi_{\{a_1,a_2\}}$ requires matching with exactly two attributes $\{a_1, a_2\}$, its lower versions can be $\Pi_{\{a_2\}}$, $\Pi_{\{a_2\}}, \Pi_{true}$. This assumption leads to a "bottom up" matching idea in that a man starts issuing a proposal with lower expectations and then increase the preference only when he got a partner. Compared to the previous solutions, in this solution the men can exchange their current partners for better ones, and women should handle the case of being actively dropped. The protocol is summarized as follows: *Initiator*

- When a man with no partner receives a *yes* he sends a $confirm$ to the partner and then issues a new proposal with higher expectation
- When a man with a partner receives a *yes* from somebody he prefers to his current partner, he sends a *bye* to the current partner, and a $confirm$ to the new partner. Otherwise he sends back a *toolate* message.
- When a man receives a *bye* he resets his partnership status and restarts issuing a proposal starting with the lowest expectations

Responder

- When a woman with no partner receives proposal she replies *yes*.
- When a woman with a partner receives a proposal from somebody she prefers to her current partner, she sends *yes* to the new potential partner.
- When a woman with no partner receives a $confirm$ she updates her partnership status.
- When a woman with a partner receives a $confirm$ from a partner not better than the current one she sends a *bye* to the new proposer.
- When a woman receives a *toolate* she continues without changes.
- When a woman receives a *bye* she resets her partnership status and continue.

AbC Specification. We have implemented this protocol in *AbC* using a slightly different approach. We used an extra process in both components Man and Woman that plays the role of a message queue manager. This process appends every incoming messages to the tail of queue, while another process implementing the main behaviour retrieves messages from the queue and processes them sequentially. The resulting specification mainly uses choice processes for cases analysis, as can be read from the above matching protocol. Here we omit the presentation of this specification. The interested reader can refer to the appendix section B.1 for full specification of this approach. In any case, all the artifacts relevant to this chapter are also available at [1].

4.2.2 Graph coloring

Given an undirected graph, we would like to assign colors to vertices of the graph such that vertices that share an edge do not receive the same color. In AbC terms, a graph is modeled naturally by a set of AbC components, one for each vertex, with attributes id representing a unique vertex id, and nei representing a set of neighbors ids that the vertex connects to.

A distributed algorithm encapsulated in a AbC specification for the graph coloring problem was proposed in [24]. Vertices communicate with their neighbors to achieve a proper coloring. Each vertex executes a coloring procedure, which takes place in rounds, independently from each other. A sending predicate has the form of $this.id \in nei$, used by a vertex id to send messages to other vertices whose attributes nei containing id. There are two types of messages exchanged between neighboring vertices. A try message of the form (‘try’, c, r) is sent to inform others that the sending vertex wants to attain color c at round r. A done message of the form (‘done’, c, r + 1) is sent to inform others that the color c has been chosen at the end of round r. Vertices may propose the same color at the same round. This conflict is resolved by using the unique identifiers of vertices. Only the vertex with greatest id among its neighbors has the right to attain its proposed color.

To select a color for a ‘try’ message, each vertex keeps a copy of the set *used* colors, initially an empty set and gradually updated via exchanging ‘done’ messages. A color is selected as the first available color which has not been used by any neighbors. In other words, $min_i\{i \notin used\}$ returns the smallest positive number that does not appear in the set *used*.

On the other hand, by collecting both ‘try’ and ‘done’ messages, a vertex can keep track of the coloring status of its neighbors. The vertex uses these information to decide if it can issue a ‘done’ message, or it would need to try a new color, i.e., starting a new round. A new round is started at a vertex, if it receives a message associated with a greater round than its current round. More details on vertex operations are elaborated in what follows.

AbC Specification. Chapter 2 discussed two slightly different specifications. One is originally presented in [24]. The other version contains a modified vertex behaviour. Here we refer them as Col Fix (modified) and Col (original), and present their specifications in our template in sequence.

There is one type of component Vertex for all vertices of an input graph. Private attributes involved to local computations of vertices include: `counter` - the number of neighbors who are trying a color at a given `round`, `done` - the number of neighbors who have done their coloring, *used* - the set of colors which have already used, *constraints* - the set of colors proposed by neighbors with greater ids. In addition, `color` is the current proposed color, `assigned` is a boolean indicating the completion of the vertex in coloring procedure. `send` is used to control when to make a color proposal.

The initial values for private attributes are the following. `color` = `undefined`, `assigned` = `ff`, `send` = `tt`, `round` = `0`, `used` = `constraints` = [], `done` = `counter` = `0`.

Col Fix. The vertex behaviour is a combination of 4 independent processes `F`, `T`, `D`, `A`. Process `F` sends try messages when able to do so (`send` = `tt`) and if it have not reached to a final decision (`assigned` = `ff`). The output action sends a message of three elements, in which the second one - '`min_color`' is simply a place holder. Actually, this place holder stands for the function $min_i\{i \notin \mathit{this.used}\}$ that selects the first available color in the set *used*. Currently expressions of this kind can not be expressed directly in our template. To get around this problem, we put such a place holder for it, and later instrument the output model to get the right semantics of the function (see the Appendix section C.1).

```
component Vertex
  attributes: id, nei, color, counter, done, constraints, used, send,
              round, assigned
  observables: color, assigned
  behaviour:
  let {
      F := <send = tt and assigned = ff>
           ('try', 'min_color', this.round)@(this.id in nei).
           [send := ff, color := 'min_color']F

      T := (x = 'try' and this.id > id and this.round = z)(x,y,z).
           [counter := counter + 1]T
          +
           (x = 'try' and this.id < id and this.round = z)(x,y,z).
           [counter := counter + 1, constraints := constraints ++ [$y]]T
          +
           (x = 'try' and this.id > id and this.round < z)(x,y,z).
           [round := $z, send := tt, counter := 1, constraints := []]T
          +
           (x = 'try' and this.id < id and this.round < z)(x,y,z).
           [round := $z, send := tt, counter := 1, constraints := [$y]]T

     D := (x = 'done' and this.round >= z)(x,y,z).
          [done := done + 1, used := used ++ [$y]]D
          +
          (x = 'done' and this.round < z)(x,y,z).
          [round := $z, done := done + 1, send := tt,
           counter := 0, used := used ++ [$y], constraints := []]D

     A := <(|nei| = counter + done) and color /= 'undefined'
           and (color notin constraints ++ used)>
          ('done', this.color, this.round + 1)@(this.id in nei).
          [assigned := tt]nil
    }
    init F | T | D | A
end
```

FIGURE 4.21: The Vertex component

Process `T` collects '`try`' messages according to the message contents. First, messages that come from neighbors who are executing the same round as `this.round` cause the attribute `counter` increased by 1. Also, the try colors (bound to $\$y$) of neighbors with `id` greater than `this.id` are stored in the set `constraints` to avoid conflict. Second, messages that come from neighbors who are executing a greater round than `this.round` causes the triggering of a new round. For this purpose, relevant variables are updated, i.e., `send` = `tt`, `round` := \$z, `counter` := `1`. The set `constraints` is instead

updated with a value depending on the sender's *id*, which is either an empty set (if id $<$ this.id) or a singleton set of the try color (if id $>$ this.id).

Process D collects 'done' messages according to the message contents, in which attribute done is increased by 1 for each incoming message and used accumulated the used color ($\$y$). Similarly to process T, messages associated with a greater round number z >= this.round trigger the startup of a new round. For this, attributes round, send, counter, constraints are updated accordingly.

Process A waits for reporting the completion of coloring, so to set the attribute assigned to true and terminates. A vertex can take this.color as it final color if it has all neighbors information |nei| = counter + done and if its current color is a valid one, i.e., color/ = 'undefined' and there is no conflict color notin constraints ++ used.

Col. Next, we provide the specification for the original AbC version [24] of graph coloring in our template. We would like to analyse of this specification in order to uncover the issues mentioned in Chapter 2.

Figure 4.22 presents the definition of the vertex component, called Vertex1. The difference between this specification with the one reported in Figure 4.21 is at the definitions of process F1, for making color proposals. Other details such as the set of attributes and their initial values, the definitions of T, D, A are the same.

```
component Vertex1
  attributes: id, nei, color, counter, done, constraints, used, send,
              round, assigned
  observables: color, assigned
  behaviour:
  let {
      F1 := <send = tt and assigned = ff>()@(false).[color := 'min_color']
           ('try', this.color, this.round)@(this.id in nei).[send := 1]F

      T := ... \*same as before*\

      D := ... \*same as before*\

      A := ... \*same as before*\
    }
  init  F1 | T | D | A
end
```

FIGURE 4.22: The Vertex1 Component

We can notice that F1 has two separate actions: the first one exploits an empty send to select a *color*; the second one sends a try message associated with the updated color and sets the attribute send to 1. On the contrary, process F (Figure 4.21) combined these two actions into one single action.

4.3 Formal analysis and verification

We have used the developed tool to translate the *AbC* specifications for case studies into UMC models. The number of generated UMC code lines varies depending on specification and on the input instances. For example, speaking only for a component type in case of classical stable marriage, the number of UMC lines is constant for a component `Man`, while it increases proportionally with the size of the problem for a component `Woman` due to the use of operator $|^n$. On the other hand, the translated code lines for graph coloring, being a constant since a component `Vertex` contains a constant number of processes (4), regardless of the graph order.

The input UMC models used in the following analysis are actually composed by two objects: the generated `SYS` object, triggered by a `start`($<$ `inputdata` $>$ event, modelling the behaviour of the *AbC* system with the given `inputdata`, and an object `Driver` which generates all the possible input data and activates the `SYS` object. For checking the generic (i.e for all inputs) validity of a formula ϕ we therefore evaluate the formula A[{not *start*} W {*start*} ϕ], which says that ϕ holds in the initial state of any of the possible scenarios. The number of generated system states reported in the rest of this section refers to the cumulative data over the whole input domain. For conducting experiments, we use a machine with an Intel Core i5 2.6 GHz, 8GB RAM, running OS X and UMC v4.4.

4.3.1 Stable marriage

UMC models and annotations

In order to verify the following properties of interests, we have annotated the generated UMC models with abstraction rules to make observable labels on states and actions.

```
Abstractions {
  -- Auto generated
  Action sending($1,$2) -> send($1,$2)
  Action received($1,$2) -> received($1,$2)
  State SYS.partner[0]=$2 ->  has_partner(C1,$2)
  State SYS.partner[1]=$2 ->  has_partner(C2,$2)
  ...
  -- Manual instruments
  Action man_level($1,$2,$3) -> man_level($1,$2,$3)
  Action woman_level($1,$2,$3) -> woman_level($1,$2,$3)
}
```

Here rules starting with `States` expose labels *has_partner*($1, $2) in all system states, where $1 is the component name and $2 is the value of its attribute partner. We assume that the identifiers of initiators and responders are in the ranges $[1 \dots n]$ and $[n + 1 \dots 2n]$ respectively, with n being the problem size. Rules starts with `Actions` instead expose `send` and `receive` labels on all transitions denoting attribute send and receive actions.

We have additionally instrumented the models with more involved annotations. In particular, we store the current level of satisfaction of people, computed when a component updates its partner. In classical SMP, the level of satisfaction of initiators and responders is determined by the position of the current partner in the preference list. In the attribute-based variant, this number is calculated based on the similarity between one's own preferences and the characteristics of partners. The procedure issues a signal `man_level`($1, $2, $3) (and similarly `woman_level`($*)) where $1 is the component name, $2 is a previous value of satisfaction level and $3 is the value to be updated.

Properties Verification

We now consider the following properties of interest that we shall verify for all the algorithmic solutions. These include:

F_1 (*convergence*) The system converges to final states:
AF FINAL [3]

F_{2a} (*completeness of matching*) Everybody has a partner:
AF (FINAL and not *has_partner*(*,0))

F_{2b} (*uniqueness of matching*) There exists only one final matching:
AG (((EF(FINAL and *has_partner*(C1,4))) implies AF (FINAL and *has_partner*(C1,4)))
and ((EF(FINAL and *has_partner*(C1,5))) implies AF(FINAL and *has_partner*(C1,5)))
and ((EF(FINAL and *has_partner*(C1,6))) implies AF(FINAL and *has_partner*(C1,6))))

F_{2c} (*symmetry of matching*) The matchings are symmetric:
AG (FINAL implies ((*has_partner*(C1,4) implies *has_partner*(C4,1))
and (*has_partner*(C1,5) implies *has_partner*(C5,1))
and (*has_partner*(C1,6) implies *has_partner*(C6,1)))

F_3 *(satisf. of responders)* The level of satisfaction of responders always increases:
AG[*man_level*($1,$2,$3)] (%2 $\geq$ %3)

F_4 *(satisf. of proposers)* The level of satisfaction of proposers always increases:
AG[*woman_level*($1,$2,$3)] (%2 $\geq$ %3)

We performed the analysis for the three proposed solutions on the whole input space. For the classical case, we considered the systems of 6 agents (i.e., three proposers and three responders), making up of 12 parallel processes. For the attribute-based variants we considered systems of 4 agents, making up of 32 parallel processes.

[3]FINAL is a shortcut for "not EX {true} true"

In the classical solution, each agent is characterised by its preference list and thus the input space has $6^6 = 46656$ configurations. In the attribute-based variant of stable marriage, each agent has four attributes (two for expressing their preferences about partners, and two for modelling their features), each having two possible values. The input space therefore has $16^4 = 65536$ configurations.

The results of our verification are reported in Table 4.1. A [✓] means that the formula is satisfied by all possible inputs, while a [×] means that the formula does not hold for at least one input.

property	F_1	F_{2a}	F_{2b}	F_{2c}	F_3	F_4
Classical	✓	✓	✓	✓	✓	×
Top-down	✓	✓	×	✓	✓	×
Bottom-up	×	×	×	✓	×	×

TABLE 4.1: Verification results of three algorithmic solutions

By looking at these results we can attempt some considerations:

Classical. Formulae $F_1, F_{2a}, F_{2b}, F_{2c}$ do hold, confirming that the classical algorithm always returns a unique and complete matching. The fact that formula F_3 holds while F_4 does not hold further reflects that a responder keeps trading up its partners for better ones, while proposers can be dropped at any time.

Top-down. Since formula F_{2a} does hold and F_{2b} does not, we can conclude that the top-down strategy will in general return multiple but complete matchings. This is not surprising, since attribute-based stable marriage is a general case of stable marriage with ties and incomplete list (SMTI) [47], and it is known that one instance of SMTI may have multiple matchings [96]. When verifying F_3 and F_4, we obtain the same results of the classical case.

Bottom-up. F_1 does not hold indicating that this approach is not guaranteed to converge. This happens in any configuration containing a cycle in the preferences which makes partners chasing each other. Formula F_{2b} does not hold because there might be two proposers competing for one responder w.r.t. their lowest requirements, thus one of the two remains single. We also verified that both formulas F_3, and F_4 do not hold. This reflects that the satisfaction levels of components may decrease because partners from both sides may drop them for better ones at any moment.

In addition to previous properties, we also considered a few protocol-related properties to increase to double check the correctness of the specifications derived from informal requirements. In particular, we verified the following property of the classical solution:

F_5. After a proposer receives a *no*, it will eventually send a new proposal[4]:
AG ([received($1,*no*)] AF {send(%1,*propose*)} true)

As expected, UMC answered true when verifying F_5. This guarantees that the proposer will send a proposal again, thus confirming that our specification in that regard meets the informal requirements.

As we have specified a communication protocol for matching entities, the following properties of the top-down solution are important to determine whether the implementation conforms to the requirements:

F_6. If a proposer receives a *bye*, it will always eventually send a new proposal:
AG([received($1, *bye*)] AF{send(%1, *propose*)} true)

F_7. If a responder sends *yes* it will eventually receive a *toolate* or *confirm*:
AG[send($1, *yes*)] AF{received(%1, *confirm*) or received(%1, *toolate*)}

F_8. After sending a proposal an initiator does not send further proposals until it receives a *no*:
AG[send($1, *propose*)] A[{*not* send(%1, *propose*)} W{received(%1, *no*)}]

By verifying the above properties, we have found out that F_7 holds, while F_6 and F_8 do not. Formula F_8 can be false, because, after sending a proposal, an initiator may receive a *yes*, and then a *bye* message which forces it to send a new proposal. F_6 does not always hold because a initiator after receiving a *bye* message from his partner, may immediately receive a *yes* message from another responder. In this case, it can confirm the new responder without the needs of sending new proposals.

Notice that the informal description of the top-down strategy is not quite rigorous. We have two statements somewhat in contrast, one statement saying that after a *yes* an initiator without a partner should send a *confirm*, and another statement saying that after a *bye* an initiator should send a new proposal. When a *bye* and a *confirm* arrive in sequence, the informal description is not clear in describing the intended behaviour. The formalisation of this requirement in terms of a logical formula, its verification w.r.t. the formal specification of the system, and the observation of the generated counter-example has allowed us to detect and understand this kind of ambiguities.

State Space. We comment on the state spaces of different solutions. The top-down solution requires the largest number of states with about 49 millions, compared with 0.5 and 2.7 millions of states of the classical and of the bottom-up solution, respectively.

The complexity of the top-down specification is the main reason for its state explosion, which stems from the use of many state variables (attributes) and parallel processes in components. Specifically, proposers and responders consist of parallel processes performing more actions than their classical counterparts, mostly for exchanging extra acknowledgment messages: in a querying phase, a proposer

[4] $*id* and %*id* are used to match the identities of the sending and receiving components.

would need to collect answers from all responders; in a proposing phase, 'confirm' messages are needed for selecting one partner out of many. This greatly increases the interleaving of actions of processes and thus the whole state space.

4.3.2 Graph Coloring

UMC models and annotations

We have annotated the generated UMC models for graph coloring specifications with the following abstraction rules.

```
Abstractions {
   -- Auto generated
   Action sending($1,$2) -> send($1,$2)
   Action received($1,$2) -> received($1,$2)
   State SYS.color[0]=$2 ->  has_color(C1,$2)
   State SYS.assigned[0]=$2 ->  has_assigned(C1,$2)
   ...
   -- Manual instruments
   -- Soundness
   State SYS.matrix[0][1] > 0 and SYS.color[0] = SYS.color[1] ->  not_sound
   State SYS.matrix[0][2] > 0 and SYS.color[0] = SYS.color[2] ->  not_sound
   ...
   -- \delta + 1 algorithm
   State SYS.maxt < SYS.color[0] -> bad_alg
   State SYS.maxt < SYS.color[1] -> bad_alg
   ...
}
```

The rules enable some aspects of the underlying L2TS model that we want to observe. First it is the attributes `color` and `assigned` of each vertex. Generally, a component with a name C exposes a label on the system states as `has_attrname(C,attrval)` where *attrname* is some `observable` attribute, declared in the specification and *attrval* is the corresponding value. Second, we label any state as `not_sound` if the `color` of two any adjacent vertices is the same. This is achieved by relying on a variable storing the adjacency `matrix` of the graph under verification. Finally, a label `bad_alg` is made exposed to states if the maximum degree + 1 (stored in a variable `maxt`) is smaller than the color value of any vertex. Note that color numbers starts from 1.

Proprerties verification

In this case study, we are interested in checking the following properties:

G_1 (*Termination*) The system converges to final states:
AF FINAL

G_2 (*completeness of coloring*) Every vertex has a valid color :
AF (FINAL and not *has_color*(*,undefined))

G_3 (*soundness of coloring*) Adjacent vertices do not share the same color:
AF (FINAL and not *not_sound*)

G_4 *(not bad algorithm)* The algorithm is $(\Delta + 1)$ - coloring:
AF (FINAL and not *bad_alg*)

G_1 states that the algorithm terminates while G_2 say every vertex gets assigned with a valid color. The property G_3 assesses the soundness of the algorithm by saying that no adjacent vertices share the same color. Finally G_4 concerns with the total number colors in use, which asserts that it should not exceed the maximum degree of the graph plus 1.

We have verified these properties for both specification Col and Col Fix considering the input space of graphs of sizes 2-5 . In order to generate all possible graphs for a given size n, we first used a tool called nauty [100] that can generate all non-isomorphic graphs in the form of adjacency matrices. For example, a graph of 5 vertices has 34 possible adjacency matrices. After that, all graphs of size n is generated by permuting the set of unique ids in the range $(1 \ldots n)$, assigning them to vertices, one permutation for each adjacency matrix.

The verification results are presented in Figure 4.2. UMC model checker verified that that all the above properties do hold for our Col Fix specification for all possible graphs of the considered sizes. The number of states generated when verifying the property AF FINAL is also included, giving an idea of the state space for each graph size.

property	G_1	G_2	G_3	G_4
Col	✓	×	n/a	n/a
Col Fix	✓	✓	✓	✓
graph size	2	3	4	5
state space	85	1469	42980	2549301

Table 4.2: Verification results of graph coloring scenario

When verifying the specification Col, property G1 holds showing that Col terminated. However, as expected the property G2 do not hold, meaning that vertices eventually do not have `assigned` as true. To see why this is case, we used UMC to generate counter examples for graph size of 2, as reported in Figure 4.23. The execution path reveled that G2 is false in a connected graph of two vertices. After getting assigned a color (vertex id 2), a vertex still sends a try message afterward, which would affect the counter value of neighbors (vertex id 1). This makes the termination condition of those neighbors, i.e., $|\texttt{nei}| = \texttt{done} + \texttt{counter}$ never happened. This is because in the Col specification, process `F1` has separated the action of color selection from the action of sending a try message. Thus there are chances for these two actions to be interleaved with process `A1`.

```
AF (FINAL and not has_assigned(*,false)) is FOUND_FALSE in State C1
This happens because:
  ...
  C13 --> C68   {} /* nei:=[[2],[1]]; matrix:=[[-1,1],[1,-1]]; */
  C68 --> C69   {} /* color[0]:=1; */
  C69 --> C70   {send(C1,[tryc,1,0])} /* send[0]:=false; */
  C70 --> C71   {receive(C2,[tryc,1,0])} /* counter[1]:=1; */
  C71 --> C228   {} /* color[1]:=1; */
  C228 --> C238   {send(C2,[donec,1,1])} /* assigned[1]:=true; */
  C238 --> C239   {receive(C1,[donec,1,1])} /* round[0]:=1; done[0]:=1;
                  send[0]:=true; counter[0]:=0; used[0]:=[1]; constraints[0]:=[]; */
  C239 --> C240   {} /* color[0]:=2;  */
  C240 --> C241   {send(C1,[tryc,2,1])} /* send[0]:=false; */
  C241 --> C242   {receive(C2,[tryc,2,1])} /* send[1]:=true; counter[1]:=1;
                  constraints[1]:=[]; */
  C242 --> C245   {send(C2,[tryc,1,1])} /* send[1]:=false;  */
  C245 --> C246   {receive(C1,[tryc,1,1])} /* counter[0]:=1; constraints[0]:=[1]; */
(C246 is final)
```

Figure 4.23: Counter example for property G2

4.4 Concluding remarks

We have presented a model-checking approach to the verification of attribute-based communication systems. Starting from informal requirements, we have devised formal specifications in AbC. We have then shown how to systematically translate these into verifiable models accepted by the UMC model checker. The main idea has been published in [45]. There, we considered the verification for a fragment of another version of the AbC calculus [21]. Here, we have refined our work to consider the most recent version of AbC (Chapter 2).

In a previous work [46], we have exploited the strong relation between AbC and UMC specification language for modelling the stable marriage example with attributes. However the modelling was based on a manual translation only where a common center broker class is employed for keeping attribute environments of all components. Instead in [45] we provided a systematic way to handle all AbC constructs, together with tool support. In this chapter, the translation is presented in greater detail and a new case study is added. We have also refined the specifications for the stable marriage case study. The verification has been also re conducted. In the previous work [45], the top down strategy for the stable marriage problem was specified with the help of a global awareness operator whose original ideas was borrowed from Linda coordination language. Using global awareness, men does not need to send querying messages to learn about the availability of interested partners; he can directly get to know them. However, for a uniform modeling approach and reasoning over case studies used through out this thesis, we omitted the use of global awareness.

The translation proposed in this chapter, although has been implemented thanks to its implementable semantics, currently still lacks a correctness treatment. We consider this as a shortcoming of our approach and will work on this issue in near future. In particular, we anticipate that a formal relationship between the generated UML state machine and an AbC system can be established by focusing on the following steps.

1. prove syntactically that there exists one parallel state machine for one AbC specification by applying our translation rules.
2. formalizing the operational semantics of parallel state machine in UMC with focuses on event queue, enabling conditions on transitions and the behaviour of concurrent regions.
3. show how a total order broadcast among concurrent regions is achieved from the uses of the global lock *receiving* and *allowsend* events.
4. show how AbC interleaving semantics among concurrent regions is achieved from the use of program counter pc

Chapter 5

AErlang - Extending Erlang with attribute-based communication

In this chapter we present an approach to programming with attribute-based communication. Differently from Chapter 3, we are not concerned much with mimicking the *AbC* syntax or semantics, but rather taking the notion of attribute-based communication in a general sense.

The communication model of *AbC* is based on message passing in which sending is non-blocking while receiving is not. Semantically speaking, in *AbC* message delivery is instantaneous in which input actions wait to synchronize on the available messages sent by asynchronous output actions. However, when considering at the level of implementation, having different components receiving a message at the same time is not possible and mechanisms are needed to guarantee causality of actions. In the previous chapter, we have seen how to relax this "instantaneous" assumption by considering message delivery occurring in synchronous communication rounds. In this way, a component can not send a message until a previous message has been delivered to all other components. As a result, the system evolves in a lock-step fashion in that only one sending component is allowed to send the message at each step.

We have implemented this system semantics by enforcing a total order on message delivery. Although the implementation exposes some degree of asynchrony, imposing a total order is too strict and may be inefficient. On the other hand, many interaction protocols can work without needing a total order. For example, in two-phase commit protocol, a coordinator first sends out a vote request to its managed sites and then waits for their votes in order to decide committing a transaction. Clearly, the votes can arrive at the coordinator in any order without affecting the scenario. This is because the coordinator only cares about the *number* of votes, not their arrival order. Similarly, in our example of graph colouring presented before, each vertex waits to collect messages sent from its neighbours without requiring any specific ordering of messages.

For this reason, we think that while the *AbC* instantaneous semantics gives rise to nice algebraic properties that are of theoretical interest, a reasonable implementation should be asynchronous to be useful in practice. In this chapter we will take this standpoint to enrich *Erlang* - a concurrency oriented programming language [29] with attribute based communication.

In fact, any general-purpose programming language can make attribute-based communication possible by providing the following key ingredients:

- *Attribute Environment*: to provide a collection of attributes whose values represent the status of a component and influence its run-time behaviour.
- *Attribute-based operations*: to establish the communication links between different components according to specific predicates over their attributes. In particular, there are:
  - $send(\tilde{v})@\pi$ that is used to send a sequence of values $\tilde{v}$ to the components whose attributes satisfy predicate π;
  - $receive(\tilde{x})@\pi$ that binds to the variables in the tuple $\tilde{x}$ the message received from any component whose attributes satisfy the receiving predicate π.
- *Attribute update operation*: to change the values of the attributes according to contextual conditions and to adapt the component's behaviour.

We have chosen *Erlang* for its light-weight concurrency model, and for its set of standard libraries which have been battle-tested through years of its use in large-scale systems, e.g., [57, 91, 104]. Processes in *Erlang* are widely regarded as actors in the actor model [14] although they were developed independently[1]. *Erlang* processes are inherently concurrent entities which share no states and communicate to one another by asynchronous point-to-point message passing. Processes encapsulate local states, have unique names while messaging is realized by copying. Incoming messages are buffered in processes mailboxes and processed one at time, thus in principles prevents problems such as race conditions [103], atomicity violations [94]. In response to a message received, a process can send a finite number of messages, create new processes and possibly change its state. This actor-based model represents a different school of thoughts in programming distributed systems, in which the concurrency model is *unified* with the underlying programming model, encouraging thinking of the application domain as a set of autonomous processes. It is argued that actors makes distributed programming easier than the dominant thread-based model present in other languages. In practice, the model has gained traction in the last decade with the arrival of many languages and frameworks, e.g.,[4, 6, 7, 10, 87].

Our proposed prototype, namely *AErlang* lifts *Erlang*'s send and receive communication primitives to attribute-based reasoning. In *Erlang*, for exchanging messages, the send primitive requires an explicit

[1] Joe Armstrong never mentioned actors in [29], but in [32] he writes that the *Erlang* views of the world are "broadly similar" to that of actors.

destination address, such as a registered name or a process identifier. In contrast, *AErlang* processes are not aware of the presence and identity of each other, and communicate by using predicates over attributes. *AErlang* aims at relieving programmers from the burden of working out details such as the explicit handling of attributes, the evaluation of predicates, and so on, while at the same time preserving *Erlang*'s excellent scalability.

Our prototype is implemented as a centralized middleware that plays the role of global process registry and takes charge of forwarding messages from senders to receivers. Although being centralized, the components process registry and message broker serve processes requests by spawning handlers on the fly, potentially exploit parallelism.

Unlike *ABEL*'s approach which aims at preserving the semantics of the *AbC* calculus, *AErlang* focuses on efficient message exchanges while its programming style follows the actor-based model of *Erlang*. Consequently, the current drawback of *AErlang* is the lack of a formal semantics which would be helpful to prove the correctness and to facilitate automated verification of *AErlang* programs. However, the relevant texts in this section still provide informal semantics of the supported programming constructs.

We also demonstrate the use of *AErlang* through a number of case studies and provide a performance evaluation of our prototype in terms of efficiency and scalability. In particular, we assess the effectiveness of our prototype by using it to program a solution to stable marriage that aims at matching members according to their preferences. For this problem, we consider implementations of different variants in different languages. Namely, we first consider a variant explicitly based on (predicates over) attributes and provide an implementation in *AErlang*. Then, we derive preference lists from the predicates and implement the classical algorithm in *AErlang*, *Erlang*, and X10, a language specifically designed to scale with the number of cores. The different implementations are instrumental to compare performances of our solutions.

5.1 AErlang support for attribute-based communication

AErlang's programming interface is presented in Figure 5.1. The commands for initialising the system, registering or de-registering processes, and handling process environments and attributes are shown in the leftmost column of the figure. The attribute-based communication system is initialised by invoking function `start` with an operating `Mode` to select the strategy for inter-process message dispatching. Our prototype provides the `broadcast` operating mode, in which a sent messages is forwarded to other processes whose exposed attributes appears in the sender's predicate. Additionally, it provides three alternative operating modes (`push`, `pull`, and `pushpull`) to improve the efficiency on some classes of systems (see Sect. 5.3 for details).

An *Erlang* process joins the *AErlang* system by invoking the function `register` and declares its own attribute environment `Env` in form of a list or a map, i.e., sets of attribute name-value pairs. The

identifiers of all the exposed process attributes are assumed to be public among all processes, so that they can specify predicates over attributes of each other. In *AErlang*, exposed attributes are those appearing in the portion of attribute environment when a process registers. After registering, processes can manipulate their local environments by using the `setAtt` and `getAtt` functions. Processes leaving the system may actively `unregister`, and when a process unregisters, then it is no longer able to use attribute-based communication.

```
% initialization
aerl:start(Mode)

% join and leave
aerl:register(Env)
aerl:unregister()

% environment handling
aerl:setAtt(List)
aerl:getAtt(List)
```

```
% attribute-based
% send and receive
to(Pred) ! Msg

from(Pred),
receive
    Pattern_1 -> Expression_1;
    ...
    Pattern_n -> Expression_n
end
```

```
% attribute-based
% send and receive with counting
Count = to(Pred) ! Msg

from(Pred,Count),
receive
    Pattern_1 -> Expression_1;
    ...
    Pattern_n -> Expression_n
end
```

FIGURE 5.1: *AErlang* programming interface.

Registered *AErlang* processes interact via attribute-based commands, as shown in the mid column of Fig. 5.1. Differently from standard *Erlang*, this pair of communication primitives replace source and destination identifiers with arbitrary conditions expressed by predicates over the declared attributes.

Predicates are strings containing Boolean expressions that can refer to attribute names (*Erlang* atoms), constants (identified by a leading underscore, e.g., `_constant`), process-local references to attributes (such as `this.a`), and process-local variables representing values (for instance `$X`). Predicate terms can be combined with comparison operators, logical connectives, and arithmetic operators such as $+, *, /, -$. Furthermore, predicates can contain the operator `in`, which denotes the membership relation between an element and a list, and user-defined functions.

The attribute-based send operation `to(Pred) ! Msg` has the effect of sending message `Msg` to all processes whose attributes satisfy predicate `Pred`. At the other end, similarly, the attribute-based receive operation `from(Pred)` is invoked to receive messages from all senders satisfying predicate `Pred`. In particular, a receive operation has the effect of retrieving from the mailbox of the receiver any message from senders whose attributes match the given predicate. The message content is instead filtered by patterns declared in the receive clause in the *Erlang* style.

Our prototype also implements counting-based variants of the attribute-based communication primitives, so that processes can be aware of how many peers they are trying to interact with (see the rightmost column of Fig. 5.1). These primitives are not part of the original *AbC* calculus. A sending operation `Count = to(Pred) ! Msg` returns the number of selected receivers (whom our middleware forwards the message to) at communication time. A matching multi-receive operation `from(Pred, Count)` can later be invoked within the same process to receive up to that amount of messages.

Intuitively, counting is helpful for processes to react to groups of messages from sets of peers, rather than single messages from specific peers. We observe that one of the main interaction patterns in group-based systems is that one component/agent makes decision by collecting a number of messages from its peers. This is useful for instance when a process, having contacted multiple peers with a request, needs to consider all their answers before moving on. This ability has proved to be useful in many coordination schemes [64, 115] and has been the main motivation of a broadcast calculus with counting [84]. Without the support of constructs of this kind, programmers have to then encode the pattern by themselves, mixing computation and communication patterns which often result in programs that are error-prone and hard to reason.

Some ingenuity is required to prevent multi-receive operations from blocking when the number of incoming messages does not reach the count. Depending on the operating mode, the outbound messages may be further filtered at the receiver's end during a send operation. The actual number of targeted receivers, and thus the triggered answers to expect from them later, would then be less than the original count. Such a circumstance is not a problem as *AErlang* makes use of implicit acknowledge messages to automatically compensate for this mismatch. However, this still does not guarantee that a multi-receive operation will never block. In the general case to guarantee a proper counting mechanism, it may be necessary to adopt adequate interaction protocols between the involved peers. As a rule of thumb every process receiving a message should always send back an answer. More sophisticated protocols may additionally be required depending on the specific application. An example of using attribute-based primitives with counting is shown in Figure 5.3 and explained in Section 5.5.1. A discussion on the benefits, in terms of language expressiveness, of the counting-based versions (within the attribute-based communication primitives) can be found at the end of Section 5.5.3.

5.2 Prototype Architecture

There are two main components in *AErlang*: a *message broker* (MB) that undertakes the delivery of outbound messages, and a *process registry* that keeps track of all process-related details that are used by the message broker to deliver messages. The message broker and the process registry jointly access a distributed database to handle all the information needed for running the system. Each *AErlang* process also has its own local data store. The actual interplay between the above components, the handling of the data structures, and the local behaviour of the involved communication endpoints, depend on the specific operating mode. In the rest of this section, we only describe the general architecture of our prototype. The different operating modes are described in Section 5.3. Further technical details of our prototype implementation are provided in Section 5.4.

The process registry handles all process-related data. It stores on the main database the process identifiers and the attributes to be used by the message broker for delivering messages. When a process joins the system, a service request to the process registry is performed to insert the identifier

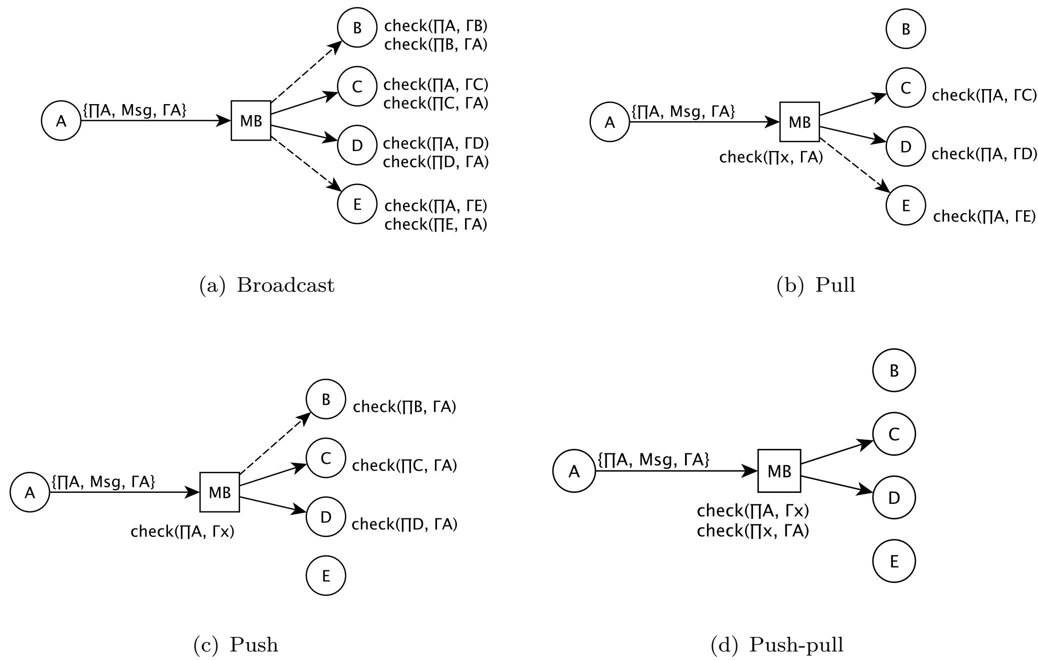

Figure 5.2: Operating modes of *AErlang*.

of the process, along with its attribute environment (if the chosen operating mode requires it) into the main database. Depending on the operating mode, the process registry also stores other details, such as the receiving predicates of the processes interested in receiving messages, or even their full environments. The environment of a process is also stored locally. All the above information is removed when the unregistering procedure is invoked.

The message broker listens for requests of attribute-based send operations. When receiving a send request, it selects from the database the set of targeted receivers and forwards the message to them.

Depending on the operating mode, the message broker can also be involved in filtering the messages according to the communication predicates.

5.3 Forwarding Strategies

This section presents in detail the operating modes supported by our prototype implementation of attribute-based communication in *Erlang*. The operating modes take into account the exposed attributes of registered processes to make sure that a message is only forwarded to those who are interested in the interaction. This treatment realizes the role of interaction interfaces of AbC components. Otherwise stated, we will assume that processes have the same set of exposed attributes. Forwarding strategies do not impose any order on forwarded messages.

Broadcast. *AErlang*'s default operating mode, broadcast, is similar to the semantics of *AbC* output action. The message broker immediately forwards all outbound messages to every other process than the sender, regardless of any interest at the other end in receiving the forwarded message. The burden of filtering messages is postponed to receiving time.

As illustrated in Figure 5.2(a), process A sends message *Msg* along with its predicate Π_A and its own environment Γ_A. Every process, other than the sender, then receives the content forwarded by the message broker and decides whether to keep the message on the basis of the sending predicate Π_A, its own receiving predicate, and the environments of both the involved communication endpoints. Please notice that the sender's environment Γ_A is included in the message as it may be needed to evaluate the predicates at the receiver's end (more details on this are reported in Sect. 5.4).

An advantage of broadcast is that it keeps the system design simple, because the required global information is essentially limited to the identity of registered processes. The main disadvantages of broadcast are the high overhead and the limited scalability. The number of forwarded messages at each send operation depends, in fact, on the size of the system rather than on the specific communication predicates. Communication-intensive tasks in a large system would therefore likely result in huge quantities of messages being unnecessarily forwarded. The transmission and the subsequent predicate evaluations to discard those messages would, in turn, cause extra communication and computational costs. Distribution, or slow transmission, and complex predicates would further exacerbate this issue.

We introduced different operating modes to address the above shortcomings. The intuition is to reduce the number of unnecessarily forwarded messages. We replicate in the process registry (part of) the information needed to compute the targeted communication endpoints. We then use this information to filter the messages to be forwarded. This improves efficiency but comes at the cost of deviating from the semantics of attribute-based communication actions. Specifically, mismatches between process-local information and their replica in the process registry can cause reception of unwanted messages, or loss of expected ones.

Pull. The pull operating mode can reduce broadcast overhead by considering the possible interests of the receivers to filter outbound messages up front, in a publish-subscribe fashion [34]. The system keeps track of the receiving predicates of all processes willing to receive messages, and uses such predicates to pre-select the receivers. This early server-side check avoids forwarding messages to processes that are not interested in interacting with some senders, or even in performing a receive operation at all.

Figure 5.2(b) sketches this operating mode. The message broker checks the receiving predicates of processes B, C, D and E (collectively represented in the figure as Π_X) against the environment Γ_A of the sender. As an effect of this check, no message is forwarded to process B. Note the difference from broadcast (see Fig. 5.2(a)), where B gets a message anyway.

The pull operating mode requires the receiving predicates to be replicated in the process registry. Therefore, it is prone to message loss or reception of unsolicited messages. Tolerance to this problem

depends on the specific application, for example in wireless sensor networks message loss is somehow expected. This forwarding strategy tends to be effective when the receiving predicates do not change frequently.

The pull mode is appropriate to model topic-based and publish-subscribe systems, where the receiving predicates correspond to subscriptions and the sending actions to publications. In attribute-based communication, a publisher (sender) will expose an attribute to denote the topic of the provided content, whereas subscribers (receivers) will express the topics of interest via receiving predicates. In this scenario, unnecessarily forwarded information is reduced because topics are filtered upfront based on the interests of the subscribers. At the same time, changes in the subscriptions presumably will not occur frequently, in which case the problem of lost or unsolicited messages can be considered negligible.

Push. The push operating mode, similarly to pull, attempts to reduce the broadcast overhead by avoiding unnecessarily forwarding outbound messages to some receivers. The filtering in this case is done by considering the sending predicate and the attribute environments of the potential receivers. A global replica of the attribute environments of all processes is maintained in the process registry, and process-local attribute update operations are notified to it.

Figure 5.2(c) shows that, given the availability of the attribute environments, sending predicate Π_{A} is checked against the environment of each potential receiver, reported in the figure as Γ_X, where X represents the receiving processes, i.e., B, C, D, and E. The check needed to select the sender is left to the receivers.

When updating attributes, problems might arise due to the interval of time passing from the local update of the environment of a process and the update of the corresponding environment in the system. As in pull mode, there may be lost or unsolicited messages. The push operating mode is suitable to scenarios where processes' attributes do not change frequently, or message loss is acceptable. For instance, in social networks, users' attributes (e.g., age, spoken languages, and education) are updated only rarely, which makes this policy convenient.

Push-pull. In the push-pull operating mode the sending and the receiving predicates are both checked by the message broker, so that communication only happens between interested endpoints.

This mode, shown in Figure 5.2(d), minimises the overall number of messages, and thus the system overhead. Another advantage is that the push-pull mode facilitates the implementation of counting-based communication patterns (see the rightmost column of Fig. 5.1 and Sect. 5.1), because the count of targeted receivers returned at the moment of sending and the actual number of receivers are usually the same, or it is quite close.

As in the case of push and pull, the main drawback of the push-pull operating mode is in the possible inconsistency between the local environment of a process and its global replica stored in the process registry. Also, the workload may increase significantly for the message broker, as it has to perform

predicate checking and handle environment update notifications for the whole system. The impact of this issue on the efficiency and scalability of our prototype can be somehow limited by parallelising the message broker, and spawning dynamically independent lightweight processes to handle incoming messages (see Sect. 5.2). Indeed, the push-pull operating mode provides fast attribute-based communication at the cost of an increased risk of losing or receiving unsolicited messages. Therefore, it can be convenient when aiming at high-speed rather than at fully-reliable systems.

Selection criteria for forwarding strategies. Table 5.1 serves as a first guidance for choosing an appropriate forwarding strategy for a given system.

In highly-dynamic scenarios where attributes and predicates change frequently, *broadcast* is the policy to go for. In *pull* mode, the system keeps track of the receiving predicates, thus this strategy is usually suitable to applications that rarely update them. In *push* mode, the attribute environments of the potential receivers are considered for enabling end-to-end communication; this strategy is thus suitable to applications where the attributes of the communicating entities are not updated frequently. Finally, the *push-pull* mode relies on verifying upfront both the attributes and the predicates, hence it can be used in applications where none of them changes frequently.

Attributes \ Predicates	dynamic	static
dynamic	*broadcast*	*pull*
static	*push*	*push-pull*

Table 5.1: Selection of forwarding strategies driven by features of attributes and predicates.

Note that the alternative policies can be applied beyond the suggested cases, as long as they are complemented with additional checks that modify the interaction protocol to handle the possibility of receiving unsolicited messages or not receiving expected ones.

Let us consider, as an example, a swarm of robots collaborating for rescuing victims in a disaster area. Once a robot finds a victim, it looks for other robots supporting the mission in the surrounding area. The sending predicate can be targeting robots whose location is within a certain zone. However, locations of robots tend to change rapidly, and, because of this, some robots may receive unwanted messages, while others may miss some. To handle this situation, a programmer can slightly alter the local behaviour so that senders periodically re-scan their neighbourhood, or receivers have the possibility of double-checking the locations. A more detailed example on how to effectively take advantage of the alternative operating modes is reported in Sect. 5.5.3.

5.4 Implementation

In this section we discuss *AErlang* implementation details. Both the process registry and the message broker implement *Erlang*'s generic server behaviour, accepting service requests by *AErlang* processes

when they join the system, perform an attribute update, or send a message. Our prototype uses Mnesia, *Erlang*'s built-in database, as the main backend for distributed storage. The process registry uses this database to store process identifiers and attributes. Attributes are kept in separate columns for increased performance and at the expense of some extra memory. Table records are indexed by process identifier. The message broker uses this information for delivering messages. The attribute environment of a process is also stored locally into the *Erlang* built-in term storage. A reference to this table is stored into the process dictionary.

We seamlessly integrate the *AErlang* primitives (Fig. 5.1) into *Erlang*'s syntax via source-to-source translation. This functionality is already implemented in the host language by the built-in parse transformation feature, i.e., essentially a function that takes as input an *Erlang* module and performs pattern-based substitutions on the abstract representation of the input, returning as output the transformed module. We use parse transformation to expand all invocations to *AErlang* primitives into fragments of *Erlang* code. In what follows, we elaborate on how translated code works in agreement with the message broker in order to achieve the desirable behaviour.

Attribute-based Send. An attribute-based sending operation `to(Pred) ! Msg`, where a message `Msg` targets a group of receivers specified by the sending predicate `Pred`, is translated as follows:

```
1  Envs = aerl:getEnv(),
2  Ps = aerl:evallp(Pred, Envs),
3  gen_server:cast(aerl_broker, {self(),Ps,Msg,Envs})
```

The sender retrieves its attribute environment `Envs` (line 1) to be used in a local evaluation of the sending predicate. The purpose of the call to `evallp` (line 2) is to provide all local information needed for a remote evaluation of the sending predicate `Pred`, either by the message broker or by some receiver. The function replaces every reference to local attributes (such as `this.a`) with the corresponding values from the environment. If the sending predicate contains references to variables (of the form `$X`), `evallp` also builds a list of bindings for them. Eventually, the sender sends a message containing its identifier, the parsed predicate, the actual message content and the environment to the message broker via a `gen_server` cast (line 3).

Message Broker. The message broker spawns a separate handler to process an incoming sending request operation. A sending request has the form `{Pid, Ps, Msg, Envs}`, and contains the process identifier, the sending predicate, the message, and the sender's environment. The handler computes the set of targeted receivers, and forwards the message to them. In `broadcast` mode, the message broker selects all registered processes except the sender, and forwards the message `{Ps, Msg, Envs}` to them. Each receiver then checks both the sending and receiving predicates. Note that, unlike *Erlang* but similarly to traditional actors, *AErlang* does not guarantee the message ordering due to the generation of a new handler for each sending request.

In push mode, the message broker translates the sending predicate Ps into a database query in the form of a matchspecification. The query includes an additional condition to exclude the process identifier Pid of the sender. The result is a list of process identifiers (potential receivers), to which the message broker forwards the message {true, Msg, Envs}. The first element of the message prevents the receiver from checking the sending predicate a second time.

In pull mode, the message broker filters the set of processes whose receiving predicates are satisfied in the sender's environment. This is accomplished by first checking the receiving predicates against the environment of the sender, and then forwarding the message {Ps, Msg, ok}. The last element prevents the receivers from checking the receiving predicate a second time.

In pushpull mode, the message broker selects the set of potential receivers and performs filtering according to the receivers' viewpoint. A second query retrieves the list of receiving predicates. The message {true, Msg, ok} is then forwarded only to receivers whose receiving predicates are satisfied in the sender's environment. The first and third elements of the message prevent further predicate checks.

Note that, depending on the operating mode, predicates can be evaluated at the broker or at the receiver's end. If local variables are used in a predicate, then the binding with the actual value of these variables is included as part of the partially-evaluated predicate. This enables remote predicate evaluation without replicating process-local variables at the message broker. Moreover, the message forwarded is also decorated with an extra field, e.g., sender's identifier, which may be used at the receiver side for point-to-point acknowledgement. Acknowledgement is needed when the message was sent by a process invoking the construct send with counting.

Attribute-based Receive. An attribute-based receive from(Pred), receive Pattern1 → Exp1; ...end is expanded into the following *Erlang* code:

```
1  Envr = aerl:getEnv(),
2  Mode = aerl:get_mode(),
3  Pr = aerl:evallp(Pred,Envr,Mode),
4  F = fun(Foo)
5   receive
6     {Ps, Pattern1, Envs} ->
7       case aerl:check(Ps, Envr) andalso aerl:check(Pr, Envs) of
8         true -> Exp1;
9         false -> aerl:handle_false(...), Foo(Foo)
10      end;
11    // code for other patterns
12  end
13 end, F(F)
```

Similarly to a sending action, the receiving predicate is first partially evaluated under the local environment of the receiving process (line 3). In the case of pull and push − pull modes, the parsed

predicate is also automatically updated in the registry. A function `F` is introduced (line 4) to wrap the user-defined receive construct with checks for the predicates. In the body of this function, the message patterns are extended with additional components to capture the messages forwarded by the broker (line 6), where `Ps` and `Envs` denote the sending predicate and the sender's environment, respectively. If the predicate checks (line 7) are satisfied, the corresponding user-defined expressions (line 8) are executed. Otherwise, the receiver first invokes the function `handle_false` and then keeps waiting for new messages (line 9). `handle_false` is responsible for the situation where the discarded message is originated from those senders who use the send construct with counting. In such a case the sender needs an explicit response; `handle_false` detects those senders, e.g., by using extra information attached in the message, and sends a special acknowledgment message to them.

The `check` functions (line 7) do verify the truth of the exposed predicates in the given environment while taking into account the forwarding strategy. Some of the checks are ignored; for example, in `push` mode, the check on the sending predicate `Ps` always returns true. Similarly, in `pull` mode, the check for the receiving predicate `Pr` is satisfied regardless of the sender's environment.

Send and Receive with counting As mentioned before, *AErlang* provides the counting variants of send and receive (see the third column of Fig. 5.1) to express useful communication and synchronization patterns between processes. A send with counting returns the number of receivers that the message broker has forwarded the sent message to. In the operating modes such as push and push-pull in which the broker can access the attributes environment of other processes, this number is calculated by evaluating the sending predicate against their attribute environments. In broadcast and pull modes, however, the broker simply returns the number of processes whose attribute names appear in the sending predicate. A receive construct supplied with a *count* collects such messages from senders whose attributes satisfy the receiving predicate. In the body of this receive, a process can react to each message received, or stores them in some attributes for later processing.

These programming abstractions allows a process collecting a number of answers from other peers before making a decision. This ability of responding to a number of messages has been demonstrated useful in some coordination schemes [64, 106, 115]. For example, in a voting protocol [69], an agent sends an inquiry message and waits for a certain number of replies in order to decide a final result on behalf of the population. This behaviour can be emulated by using a pair of send and receive with counting: a requester sends an inquiry message and obtains the number of potential receivers; a receive construct parameterized with this number is used afterward to handle answers. However, it can happen that the number of actual replies does not match the number that a process is expecting. This is because the requester of an inquiry message may be not interested by the some receivers. We handle this problem by using implicit acknowledgements: the process identifier of a requester is included in the sent message, denoting for the need of acknowledgement. Upon checking, any process sends an ack message to the sender, even if discarding the message.

The receive with counting specifies the number of messages `Count` it intends to receive from senders satisfying the receiving predicate. The generated code for this construct has the following form:

```
1  Envr = aerl:getEnv(),
2  Pr = aerl:evallp(Pred,Envr),
3  F = fun Foo(0) -> ok;
4   Foo(_Cnt) ->
5   receive
6     {Ps,Pattern_1,Envs} ->
7       case aerl:check(Ps,Envr) andalso aerl:check(Pr,Envs) of
8         true -> Expression_1,
9                 Foo(_Cnt-1)
10        false -> aerl:handle_false(...) % handle false for sender
11                 Foo(_Cnt)
12      end;
13
14    // code for other receiving patterns ...
15
16    % receive an explicit acknowledge for the sent message
17     ack_msg_discarded -> Foo(_Cnt-1)
18   end
19 end, F(Count)
```

As opposite to the receive without counting, the generated function F takes the `Count` parameter (line 19) and enters a receive loop. The body of this receive includes a pattern to discard the acknowledgment message (line 17), meaning that the inquiry message is discarded by some process. The value `_Cnt` is decreased when a message is accepted (line 9) or an acknowledgment message of the sent message is received (line 17).

5.5 Case Studies

In this section we show how *AErlang* can be used for attribute-based programming. A first case study is the well-known problem of Stable Marriage (SM), and we describe a program that implements the classical solution for this problem, and then consider progressively more elaborate variants. The purpose is to show the convenience of relying on attributes for interactions. We also discuss how the proposed approach can be generalised to model realistic examples of collective adaptive systems, such as social networks.

As a second case study, we consider an interactive market where users aim at trading goods. We present a possible interaction protocol and an *AErlang* program that implements it. Moreover, we discuss the counting mechanism for modelling more complex interaction patterns that cannot be easily expressed otherwise.

Finally, we describe the program for the graph coloring problem (see Section 2.2.2) in *AErlang*. We use it to highlight the differences between the two programming models, i.e., *AErlang* and *ABEL*.

SM consists in finding a matching between sets of men and women, where each person has a preference list of members of the opposite gender [65]. A *matching* is a set of one-to-one assignments between men and women. Each assignment is denoted by a *pair* (m, w), where m and w indicate the two matched *partners*. A pair is *blocking* if, according to their respective preference lists, both the matched man and woman prefer someone else to their partners. A matching is *stable* if there is no blocking pair. A matching is *complete* when everybody is matched, *incomplete* otherwise.

SM was originally used to match hospitals to resident doctors [108], but it has many other practical applications [79]. Recently, it has been adopted for public school admissions [13], transplant of human organs [110], matching of sailors to ships [92], or primary school students to secondary schools [116]. Other applications include computer networks, for global load balancing in content delivery networks [95], and even economics, e.g., for market trading [109]. SM has been intensively studied in the literature, together with its variants [85]. In the classical form, the preference lists are strictly-ordered and complete. For this, Gale and Shapley gave an efficient algorithm to find a stable matching [65]. It can be informally summarized as follows. Each man actively proposes himself to his most favourite woman according to his preference list. Whenever a man is rejected, he tries again with the next woman in the list. On the other hand, each woman continuously waits for incoming proposals. A woman without a partner immediately accepts any proposal, otherwise she compares the proposer with her current partner. She then rejects the man whom she likes less, according to her preference list. The algorithm terminates when every man has a partner.

Variations of this algorithm are based on different kinds of preference lists: incomplete (SMI), with ties (SMT), or both (SMTI). While the first two variants can be solved similarly to the classical case, SMTI is hard [86]. In this section we investigate a new variant of the algorithm where the matching happens by taking into account the mutual interests of partners characteristics (attributes), rather than preference lists of identifiers. We call this variant stable marriage with attributes (SMA).

5.5.1 Stable Marriage with preference list

In this section we consider the variant of SM known as SMTI [86], in which the preference list is incomplete and partially ordered, i.e., a man or a woman may like several people at the same level. The preference list is thus a list of sets rather than single elements and we refer to such sets as *ties*.

We model this problem in *AErlang* by introducing an attribute id to represent people identifiers and predicates over them to specify the preferences. As an example, Table 5.2 shows the predicate lists induced from a SMTI instance (on the left) where ties are enclosed by parentheses. To implement preference list we use predicates over the attribute id, and model ties as disjunctions of comparisons. We refer to the newly derived lists shown in Table 5.2 (on the right) as predicate lists.

id	preference list
m_1	w_1, w_2
m_2	w_1, w_2
w_1	(m_1, m_2)
w_2	m_1

id	predicate list
m_1	$id = w_1$, $id = w_2$
m_2	$id = w_1$, $id = w_2$
w_1	$id = m_1 \vee id = m_2$
w_2	$id = m_1$

TABLE 5.2: Correspondence between preference lists and predicate lists.

```
1  man(Prefs,Id) ->
2   [H|T]=Prefs,
3   Count=to("id in $H") ! {propose,Id},
4   from("id in $H",Count),
5   receive
6    {yes,W} ->
7      case aerl:getA(partner) of
8        none ->
9              to("id=$W") ! confirm,
10             aerl:setAtt(partner,W);
11          _ ->
12             to("id=$W") ! busy
13       end
14  end,
15  case aerl:getA(partner) of
16      none -> man(T,Id);
17       _ -> from("id=this.partner"),
18            receive
19              goodbye ->
20                  aerl:setAtt(partner,none),
21                  man(Prefs,Id)
22            end
23   end.
```

```
24 woman(Prefs,Id,P) ->
25  from("bof($Prefs,$P,id)"),
26  receive
27     {propose,M} ->
28         to("id=$M") ! {yes,Id},
29         from("id=$M"),
30         receive
31             confirm ->
32                 to("id=$P") ! goodbye,
33                 woman(Prefs,Id,M)
34             busy ->
35                 woman(Prefs,Id,P)
36         end
37  end.
```

FIGURE 5.3: Stable Marriage with preference lists in *AErlang* (SMTI-aerl).

We then solve the problem under this new representation of preferences by using a similar solution to the classical Gale-Shapley algorithm described above, but using a slightly different protocol and relying on message-passing.

The *AErlang* program for STMI is shown in Figure 5.3. Function man() takes as arguments the preference list Prefs of a man and his identifier Id. The first element in Prefs is bound to variable H by pattern matching on list (line 2). A man goes through a proposing phase from lines 3 to 14. First, he sends a propose message using “id in $H“ as the sending predicate (line 3) which has the effect of contacting all women whose id belongs to the list H. He then waits for enough answers from the women he contacted using the attribute-based receive construct with counting (line 4), with the same predicate used when sending. Inside the body of this receive operation, the man is only interested in yes messages. He becomes aware of his status by checking attribute partner (line 7) to take a decision. If he has no partner, he sends a confirm message to the first woman who said yes by using her identifier W attached in the reply message. He then considers this woman as his current partner (line 10), and informs any other interested women that he is no longer available by sending them a busy message (line 12).

After the proposing phase, a man can either be alone or engaged (checked by line 15). In the first

case, he does not consider any woman in the current predicate H and tries to propose himself again to the women in the remaining part of his preference list (line 16). In the second case, he takes no action unless he receives a goodbye message from his partner (lines 17-19), in which case he tries proposing himself again using his current predicate unchanged (line 21). The man keeps the predicate unchanged as it may include other women.

Function woman() takes as arguments a preference list Prefs, an identifier Id, and the partner's identifier P. A woman always waits for proposals from men who are better than her current partner. This comparison is performed with the bof function (line 25) that checks if a proposer precedes the current partner P in the woman's preference list. If this is the case, then the woman sends back a yes message and waits for a confirm message from the new man M. After M confirms to her, the woman gets engaged to him by keeping M in the recursive call (line 33), after rejecting her current partner P. Otherwise, she keeps listening for other proposals (line 35).

5.5.2 Stable Marriage with attributes

In this variant each person has a set of attributes describing their own characteristics and some preferences over the attributes of their potential partners. Each attribute has a finite domain, while preferences are represented by logical expressions over the attributes of the partners. For simplicity, in this section we only consider simple predicates where preferences are conjunctions of equality comparisons.

Table 5.3 shows an example of SMA instance of size four where each person has two attributes, which in turn have two possible values. This example points out the expressive power of attribute-based communication. In fact, our program for SMA (see Fig. 5.4) is very similar to the program proposed in previous the section (see Fig. 5.3), and the differences are mostly accommodated by altering the predicates. In addition, men can progressively adapt their preferences to increase the chances to find a partner. For example, no woman in Table 5.3 satisfies the requirements of man m1, hence he looks for partners partially matching his initial preferences. This adaptive behaviour is achieved by transforming preferences into predicate lists, as shown in Table 5.4. For example, when man m1 relaxes his preferences and looks for women with amber eyes only, then there are women w1 and w2 satisfying such predicate. We assume that the ordering of attributes within a predicate indicates their priority.

Figure 5.4 shows a possible *AErlang* implementation for SMA. Function man() takes as arguments the predicates list Prefs of a man, his Id and characteristics Atts. The first element in Prefs (i.e., the most demanding predicate) is bound to variable H by pattern matching on list (line 2). The proposing phase of a man is implemented via lines 3-14 and follows the same behaviour described in previous section.

id	wealth	body	preference
m_1	rich	strong	eyes=amber ∧ hair=red
m_2	rich	weak	eyes=green ∧ hair=dark
m_3	poor	strong	eyes=green ∧ hair=red
m_4	poor	weak	eyes=amber ∧ hair=red

id	eyes	hair	preference
w_1	amber	dark	wealth=poor ∧ body=weak
w_2	amber	dark	wealth=rich ∧ body=strong
w_3	green	red	wealth=rich ∧ body=strong
w_4	green	dark	wealth=rich ∧ body=weak

TABLE 5.3: Attributes and preferences for men and women.

id	predicate list		
m_1	eyes=amber ∧ hair=red	eyes=amber	hair=red
m_2	eyes=green ∧ hair=dark	eyes=green	hair=dark
m_3	eyes=green ∧ hair=red	eyes=green	hair=red
m_4	eyes=amber ∧ hair=red	eyes=amber	hair=red

TABLE 5.4: Predicate lists with relaxed preferences for men.

```
1  man(Prefs,Id,Atts) ->
2    [H|T]=Prefs,
3    Count=to(H) ! {propose,Id,Atts},
4    from(H,Count),
5    receive
6      {yes,W} ->
7          case aerl:getA(partner) of
8            none ->
9              to("id=$W") ! confirm,
10             aerl:setA(partner,W);
11           _ ->
12             to("id=$W") ! busy
13         end
14   end,
15   case aerl:getA(partner) of
16     none -> man(T,Id,Atts);
17     _ -> from("id=this.partner"),
18          receive
19            goodbye ->
20              aerl:setA(partner,none),
21              man(Prefs,Id,Atts)
22          end
23   end.
```

```
24 woman(Prefs,Id,P,PA) ->
25   from("bof($Prefs,$PA,wealth,body)"),
26   receive
27     {propose,M,MA} ->
28        to("id=$M") ! {yes,Id},
29        from("id=$M"),
30        receive
31          confirm ->
32            to("id=$P") ! goodbye,
33            woman(Prefs,Id,M,MA)
34          busy ->
35            woman(Prefs,Id,P,PA)
36        end
37   end.
```

FIGURE 5.4: Stable Marriage with attributes in *AErlang* (SMA-aerl).

Function woman() takes as arguments the preferences Prefs, an identifier Id, in addition to arguments P and PA to keep the current partner's information. A woman waits for proposals from men whose attributes are better than her current partner. This comparison is performed with the bof boolean function (line 25) that checks if a proposer is characterized by attributes wealth and body, better than the partner P characterized by the variable PA. If this function provides true as output, then the woman sends a yes message back and waits for an acknowledge message confirm from this man M. If M confirms to her, the woman gets engaged to him by keeping M and his characteristics MA in the recursive call (line 33), after rejecting her current partner P. Otherwise, she keeps listening for other

proposals (line 35).

5.5.3 An interactive market

We consider the case of a decentralised market where users intend to trade goods. Everyone has an initial budget and a list of items that they either own and are willing to sell, or are seeking to buy, at a given price. To promote the activities, users regularly advertise their sales. At the same time, they also consider advertisement from the others, reacting accordingly whenever possible.

A system of this kind can be programmed quite naturally in *AErlang*. Users can be implemented as separate *AErlang* processes with attributes to represent their *balance*, the list of items on sale with their ask price (or *sales list*), and the list of wanted items along with their bid price (*wish list*). The exchange of items is simply modelled with appropriate predicates. Depending on the specific interaction protocol, the communication predicates may be more or less complex, and some of the attributes may not have to be publicly exposed. In any case, the idea is to use predicates to take into account the interests of the users, so to increase the overall efficiency of the system.

Interaction protocol. Alice and Bob are two users of our interactive market system acting as a seller and a buyer, respectively. Initially, Alice advertises her sales by sending to every potential buyer the type of the item, the price, and her contact details. She then waits for incoming answers from any interested user, or advertisements about other items.

When Bob receives the sales advertisement, he first checks his wish list to compare the sales price with the price he had in mind. Then if he is still interested and can afford it, he informs Alice of his interest in buying and sends his contact details to her. Bob then waits for Alice to confirm or to cancel the sale (if in the meantime she has sold the item to someone else).

At the other end, Alice receives Bob's message and double checks whether the item is still in her sales list. If the item is available, she immediately notifies Bob and updates her sales list and account balance accordingly. Otherwise, Alice tells Bob that the item is no longer available. She then keeps advertising her sales and answering to incoming messages. Bob adds the new item to his sales list and adjusts his account balance if he receives a confirmation message from Alice[2]. Otherwise, he moves on to advertising his own items and answering to incoming messages in the same way as Alice.

Also in this case, the advantage of using attributes is evident. A buyer only involves those users that are offering the goods of interest and to start does not need to know their identities. Moreover users can seamlessly join and leave.

[2]It might sound strange that one puts back on sale immediately an item he has just acquired. We took this design choice to increase the number of interactions and thus more informative performance evaluations.

Implementation In a realistic implementation of our interactive market, a service that limits the overall number of messages is preferable over a more reliable one that uses a lot of bandwidth. On the one hand, choosing to run the service in *push-pull* mode rather than *broadcast* would make it easier to fulfil this requirement. On the other hand, in *push-pull* mode any delayed update of the attribute environments or of the receiving predicates at the message broker can lead to lost and unsolicited messages (see Sect. 5.3). To avoid this kind of problems, we introduce a few local variables to keep track of the previously contacted sellers and the items already booked, as described below.

An *AErlang* program for our interactive market is shown in Figure 5.5. Here, attribute id expresses the identity of the user, slist the sales list, and wlist the wish list. For simplicity, we use individual attributes item1, . . . , itemn associated with the price of an item. Moreover, we use local variables Balance, Sellers, and Booked to represent the account balance, the identifiers of the contacted sellers, and the identifiers of the items that a user is interested in buying, respectively. We also store the list Advertised of items previously advertised, to make sure users advertise every time a different item, and that each item is advertised at most once.

```
1  loop(Balance, Sellers, Booked, Advertised) ->
2    Adv = case aerl:getAtt(slist) -- Advertised of
3      L when length(L) > 0 ->
4        SItem = lists:nth(rand:uniform(length(L)), L),
5        to("$SItem in wlist") ! {adver,SItem,aerl:getAtt(SItem),aerl:getAtt(id)},
6        [SItem];
7      [] -> []
8    end,
9    Advertised2 = Advertised ++ Adv,
10   from("match(slist,this.wlist) or id in $Sellers"),
11   receive
12     {adver,Item,Price,Seller} ->
13       case not lists:member(Item,Booked) and Price =< aerl:getAtt(Item) of
14         true ->
15           to("id = $Seller") ! {interest,Item,aerl:getAtt(id)},
16           loop(Balance - Price, Sellers ++ [Seller], Booked ++ [Item], Advertised2);
17         false ->
18           loop(Balance, Sellers, Booked, Advertised2);
19       end;
20     {interest,Item,Buyer} ->
21       case lists:member(Item, aerl:getAtt(slist)) of
22         true ->
23           to("id = $Buyer") ! {sold,Item,aerl:getAtt(Item),aerl:getAtt(id)},
24           aerl:setAtt(slist, aerl:getAtt(slist) -- [Item]),
25           loop(Balance + aerl:getAtt(Item), Sellers, Booked, Advertised2);
26         false ->
27           to("id = $Buyer") ! {cancel,Item,aerl:getAtt(Item),aerl:getAtt(id)},
28           loop(Balance, Sellers, Booked, Advertised2)
29       end;
30     {sold,Item,Price,Seller} ->
31       aerl:setAtt(wlist, aerl:getAtt(wlist) -- [Item]),
32       aerl:setAtt(slist, aerl:getAtt(slist) ++ [Item]),
33       loop(Balance, Sellers -- [Seller], Booked);
34     {cancel,Item,Price,Seller} ->
35       loop(Balance+Price, Sellers -- [Seller], Booked -- [Item], Advertised2);
36   after 0 ->
37     loop(Balance, Sellers, Booked, Advertised2)
38   end.
```

Figure 5.5: User behaviour for the interactive market.

Advertisement is performed in lines 2 to 8. A user extracts a random item from the sales list and sends an advertisement message to everybody whose wish list contains that item.

When receiving an advertisement message (line 12), a user checks whether he can afford it and he had not booked the item already. In this case, he sends an interest message to the seller and adds the item to the Booked list. This makes the interaction protocol more robust in *push-pull* mode. To see why, suppose that a user has just bought an item, but the update of his wish list at the message broker is delayed. In such a circumstance, further (unsolicited) advertisement messages from other sellers about the same item would still make it through the check at line 10. Without double-checking the list of booked items (line 13), the user would buy the item again. To avoid that, the user keeps track of the item (line 16), so that the check at line 13 can only be satisfied once for the same item.

When a user receives an interest message for an item, he checks whether the item is still in his sale list (line 21). If so, he confirms the sale and updates his sale list and balance accordingly (lines 23-25).

Upon receiving a sold message that confirms a successful sale, a user moves the item from the wish list to the sales list. If the seller cancels the deal, the user removes the item from his booked list so that he can try again.

Expressiveness of counting. With particular regard to the case study considered in this section, it may be worth making a few considerations about how counting can indeed support intuitive design of complex interaction patterns. Specifically, the counting supports scenarios where the behaviour of the involved entities evolves according to gathered groups of answers rather then replies from single peers (see Sect. 5.1 for a more general discussion, and program 5.3 from Sect. 5.5.1 for an example).

To make the interactive market more attractive to advertisers, for instance, one might think to give to sellers the opportunity to sell at the highest possible price. Roughly speaking, to add this feature, one could slightly change the program in Fig. 5.5 in such a way that an interested buyer discloses his bid price when notifying his interest in buying. The seller could then use counting to collect all the answers from the users reached by his advertisement, and sell to the highest bidder.

Similarly, let us consider the case of a seller looking for a quick sale, and for this reason prepared to lower the ask price after a given number of unsuccessful attempts to sell an item. Obviously, at the same time the seller wants to minimise the chances of lowering the sale price while somebody is still interested in buying at the current price. Without counting, the seller is not able to decide when it is reasonable to stop looking for interested potential buyers, and thus the described variation would not be obvious to implement.

5.5.4 Graph Colouring

Consider the graph colouring problem in a message passing model where each vertex is implemented as a process. Vertices exchange messages asynchronously in order to reach a final color assignment such that all vertices eventually get assigned and adjacent vertices are colored differently. Furthermore, the number of colors used should not exceed the maximum degree of the input graph plus one.

A solution to graph colouring in *AErlang* is similar to the one presented in Chapter 2, in which vertices use their unique identifiers to resolve conflicts (that arise when two neighboring vertices propose the same color). Each vertex iterates through a number of rounds. In each round, vertices exchange color proposal messages with their neighbors; only the vertex with greatest id wins the conflict color. This means that after sending a propose message, a vertex should consider all the messages from neighbors in order to make a decision. We will use a receive with counting to program this behaviour.

Any vertex that wins a color then sends a message 'done' to its neighbors and terminates, other vertices instead send a message 'not_done' to inform their status anyway. The 'not_done' vertices also collect the coloring status from their neighbors in order to update the palette. This behaviour is again easily captured by using a receive with counting of *AErlang*.

More concretely, we will enrich processes with the following attributes to model the graph colouring scenario in *AErlang*:

- id: the unique vertex identifier
- nlist: the list containing ids of neighboring vertices
- maximal: the attribute recording if the executing process has the greatest id among unfinished neighbors in a given round
- done: counts the number of neighbors which has already finished their coloring
- used: the set containing color values already taken
- round: the round number

The behaviour of a vertex is given in Figure 5.6. For convenience in accessing id and round, we keep them as the parameters of the function. When starting a round, a vertex chooses the first color which does not appear in the set *used* (line 2) and sends a 'try' message to neighbors (line 5). It then collects a number of try messages from its neighbors, except those that have completed the coloring (line 6). The attribute *maximal* is set to true if the vertex has the greatest id among neighbors. This implies that the vertex will send a 'done' message (line 15). Otherwise, the vertex can not have the proposed color, in which case it sends a 'not_done' message (line 17). It then waits for a number of replies by using another receive construct with counting. This second multi-receive is necessary because

```
1  vertex(Id, Round) ->
2     C = min_color(aerl:getAtt(used)),
3     N = length(aerl:getAtt(nlist)),
4     aerl:setAtt(maximal,true),
5     to("this.id in nlist") ! {'try', Id, C},
6     from("id in this.nlist", N - aerl:getAtt(done)),
7     receive
8        {'try', SId, C} when SId > Id ->
9           aerl:setAtt(maximal, false);
10       {tryc, _, _} ->
11          ok
12    end,
13    case aerl:getAtt(maximal) of
14       true ->
15          to("this.id in nlist") ! {done, C};
16       false ->
17          to("this.id in nlist") ! {not_done, Id},
18          from("id in this.nlist", N - aerl:getAtt(done)),
19          receive
20             {done, SColor} ->
21                aerl:setAtt(done, aerl:getAtt(done) + 1),
22                aerl:setAtt(used, sets:add_element(SColor, aerl:getAtt(used)));
23             {not_done, _} ->
24                ok
25          end,
26          vertex(Id, Round + 1)
27    end.
```

Figure 5.6: Vertex program.

there may be more than one 'done' message sent from neighbors. Any color $SColor$ associated in an incoming 'done' message is then recorded as *used* while other messages are discarded. Finally the vertex proceeds with the next round (line 26).

5.6 AErlang Performance Evaluation

In this section we discuss the performance evaluation of *AErlang*; the prototype is publicly available [3] to reproduce the experimental results. Our experimentation focuses on the following aspects:

1. efficiency in terms of run-time overhead – Sect. 5.6.1;
2. scalability in terms of size of the instances and hardware resources – Sect. 5.6.2;
3. comparison of the broadcast and push forwarding strategies – Sect. 5.6.3;
4. comparison of the broadcast and push-pull forwarding strategies – Sect. 5.6.4;
5. comparison of *ABEL* and *AErlang* broadcast – Sect. 5.6.5.

Multiple case studies are used for the evaluation of these aspects. Specifically, the first three items above are evaluated by relying on the stable marriage case study. The fourth aspect considers the interactive market, and for the fifth point we make use of the distributed graph coloring problem.

	size				
	100	200	300	400	500
SMTI-aerl / SMTI-erl	**2.99**	1.73	1.92	1.98	2.20
SMA-aerl / SMTI-erl	2.21	1.36	1.36	1.43	1.65
SMA-aerl / SMTI-aerl	0.73	0.71	0.72	0.72	0.75

TABLE 5.5: *AErlang* to *Erlang* runtime ratios.

5.6.1 Efficiency

To evaluate the efficiency of *AErlang*, we compare the runtime performance of SMA-aerl, SMTI-aerl (Fig. 5.3), and SMTI-erl. The latter being an *Erlang* program implementing the same matching protocol used in SMTI-aerl and reported in the Appendix (Fig. A.1). The hardware environment used for our experimentation is a machine consisting of 4 CPUs AMD Opteron 6376 2.3 GHz, 2MB Cache, 64GB RAM. The versions of OS and *Erlang* are Linux 4.4.0-62-generic and 19.1, respectively.

We generated multiple random input instances by considering problem sizes from 100 to 500. We considered two attributes for women and two for men, each attribute having a domain of two values (like in Table 5.3), with a probability of occurrence ranging from 0.1 to 0.9. We used the same ranges for preferences. We selected 24 different combinations in the given probability ranges, and generated 10 instances for each combination. Since SMTI-erl and SMTI-aerl take preference lists as input, we have also converted the problem instances to use preference lists. In fact, SMA can always be cast into SM by converting preferences over attributes to preferences over identifiers. This can be done by assigning a weight to each attribute and summing up the weights of all the attributes exposed by the identifiers to obtain the preference list. Finally, we ran each instance 10 times and took the average execution times.

For these experiments we rely on the push operating mode since in SMA predicates are dynamic and attribute are static (see Sect. 5.3). The upper part of Table 5.5 reports the runtime ratio of the SMTI-aerl and SMA-aerl programs with respect to SMTI-erl. Here, columns list the instance size, whereas rows enumerate the compared variants. We observe that the ratio is always within the same order of magnitude, more precisely we found a maximum ratio of 2.99 (observed for SMTI-aerl vs SMTI-erl with 100 instance size), as highlighted by the bold entry in Table 5.5, and a minimum one of 1.36 (observed for SMA-aerl vs SMTI-erl with 200 and 300 instance size). This suggests that the new programming abstractions introduce an acceptable performance overhead (always within the same order of magnitude) which is minimized when attributes are considered for predicate evaluation. In fact, in Table 5.5 we can notice that SMA-aerl always shows lower ratios with respect to SMTI-aerl. This is not affected by the instance size, i.e., with larger instance sizes the ratio remains within the min-max values observed for rather small instance sizes.

It is worth noticing that the SMA-aerl variant always outperforms SMTI-aerl, as showed in the last row of Table 5.5. This is due to the different cost of predicate evaluation, in fact the former uses sending

predicates whose complexity is independent from the input size (e.g., "`hair` = `blonde` `and` `eye` = `amber`") whereas the corresponding predicates of the latter need to check membership of identifiers within ties and therefore may be as large as the size of the tie itself (e.g., "`id` = `w1` `or` `id` = `w2` `or`..."). Note that this also holds at the receiver side.

5.6.2 Scalability

The scalability of our prototype is evaluated by increasing: (i) the size of the input instances from 1k to 5k and comparing *AErlang* with AS-X10; (ii) the number of cores from 2 to 48 and comparing *AErlang* with its *Erlang* counterpart. Similarly to previous section, in these experiments we used the push operating mode. The hardware environment used for these experiments is an idle local workstation equipped with 128 GB of memory, a dual Intel Xeon processor E5-2643 v3 (12 physical cores in total) clocked at 3.40 GHz. The operating system is a 64-bit generic Linux kernel version 4.4.0, the used software is *Erlang*/OTP version 19.1, and X10 version 2.4.2.

Comparison with AS-X10

In [102], the authors proposed adaptive search as an efficient approach to the solution of the SMTI and SMI problems. They model SMTI as a permutation problem and try to resolve blocking pairs until an acceptable size of the matching is achieved. Their framework, namely AS-X10 implemented in the X10 programming language [41], can handle instances up to the size of 1000 pairs with good performance and scalability on a large number of cores thanks to a fine-tuned cooperation mechanism between many parallel solvers.

SMTI aims at finding a matching of maximum size, since there might exist different matchings that represent a solution for a SMTI instance. In [102] there is a reset mechanism to guide the search out of sub-optimal solutions and try to achieve perfect matchings, i.e., when the matching size coincides with the problem size. Since there is no guarantee that a perfect matching exists, the search stops after a fixed number of steps.

To improve the matching size and guarantee a fair comparison with AS-X10, we adopted the local approximation algorithm proposed in [88] and modified accordingly our SMTI-aerl program (see Fig. 5.3). The idea is that an unmatched man, once his preference list is exhausted, can start afresh once more. On this second attempt, a woman can prefer him to her current partner if they both have the same priority on the woman's preference list. This heuristic resulted in very good solutions, i.e., perfect matchings for all the problem instances used when comparing the performance of SMTI-aerl with AS-X10.

In this experiment we used the inputs originally described in [72], which are generated by using the tool [8] that takes three parameters as input; namely, size of the instance (n), probability of

incompleteness (p_1), probability of ties (p_2). The generator has also been used in empirical studies of SMP and its variants, for example, in the local search approach [67] and in constraint programming [37]. We generated two classes of instances while considering instance sizes up to 5k pairs of elements and the following sets of parameters: (i) 80% of incompleteness and no-ties instances (i.e., $p_1 = 0.8$, $p_2 = 0$); (ii) 95% of incompleteness and 80% of no-ties instances (i.e., $p_1 = 0.95$, $p_2 = 0.8$). These parameters were intentionally selected to be in line with those chosen in the evaluation of the adaptive search approach, for a fair comparison [102].

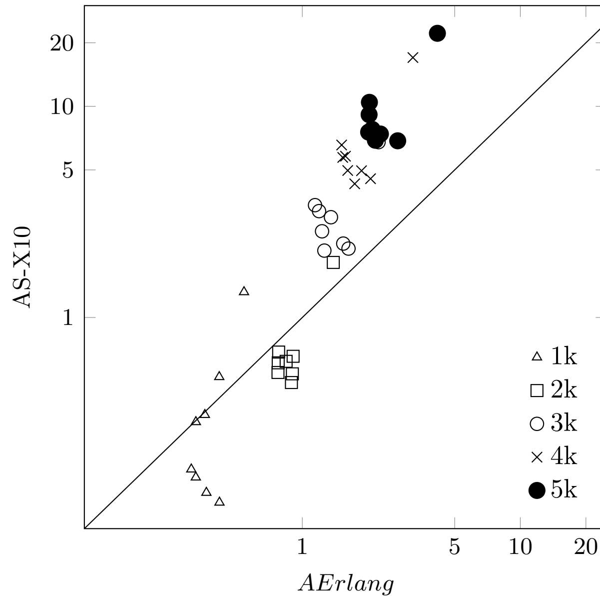

FIGURE 5.7: *AErlang* vs AS-X10: scalability over problem size.

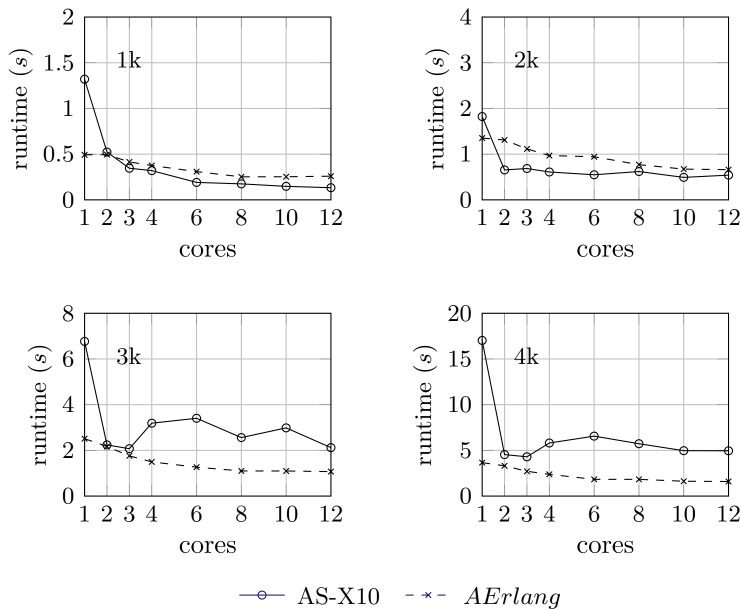

FIGURE 5.8: *AErlang* vs AS-X10: scalability over cores.

Figure 5.7 compares the performance of *AErlang* and X10 in solving multiple instances of SMTI, while varying the problem size from one to five thousands. Data points represent the average runtime ratio over one hundred runs between the two approaches. We can notice that X10 performs faster than *AErlang* only on small instances with one thousand pairs of elements. However we do notice that when increasing the size of the instances the *AErlang* program scales considerably better. This gap tends to increase with size, making the *AErlang* program very competitive on larger instances.

Figure 5.8 compares the performance of *AErlang* and X10 while varying the number of cores from 1 to 12. For each problem size, a graph shows on the y-axis the runtime overhead expressed in seconds, and the number of cores is reported on the x-axis. For small instances, X10 exhibits very good scalability, but for larger instances, this no longer holds. On the contrary, *AErlang*'s scalability appears to be quite modest but more consistent across all the considered sizes.

Comparison with *Erlang*

We implemented an *Erlang* program for the classical algorithm by Gale-Shapley, and used it to compare runtime performance with the *AErlang* program for SMTI and SMI. We also used the same input generator to generate problem instances for both SMTI and SMI problems, while considering instance sizes up to 10k pairs of elements and the following set of parameters: 80% of incompleteness and no-ties instances (i.e., $p_1 = 0.8$, $p_2 = 0$).

We ran the *AErlang* program for SMTI (see Fig. 5.3) and the *Erlang* program for SMI (see Fig. A.2) to safely exclude any hidden complexity due to the management of the ties. The size of the instances is fixed to the largest available option, i.e., 10 thousands of pairs of elements, and by ranging the number of cores from 2 to 48. We ran 10 instances, 10 times each, and collected the average execution times. This experiment was performed on a computing cluster [114] where we had access to nodes with 64 Intel CPUs clocked at 2.3 GHz and 110 GB of memory running a scientific Linux distribution.

The results are presented in Figure 5.9, where the x-axis denotes the number of cores and the y-axis reports the execution time in seconds on a logarithmic scale. Interestingly, the pronounced fluctuations in the running times are consistent for both *AErlang* and *Erlang* programs. This suggests that performance glitches within the *Erlang* subsystem end up affecting our *AErlang* prototype too.

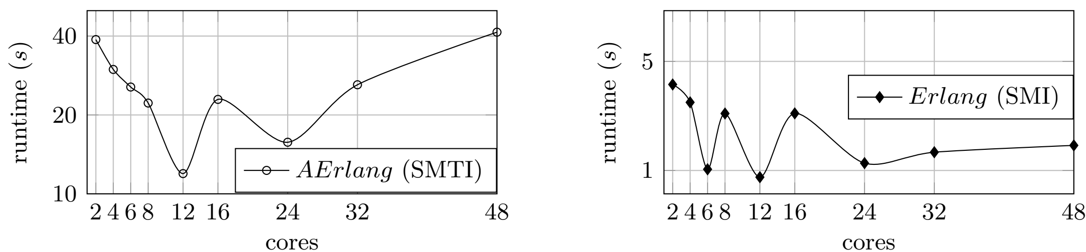

FIGURE 5.9: *AErlang* vs *Erlang*: scalability over cores.

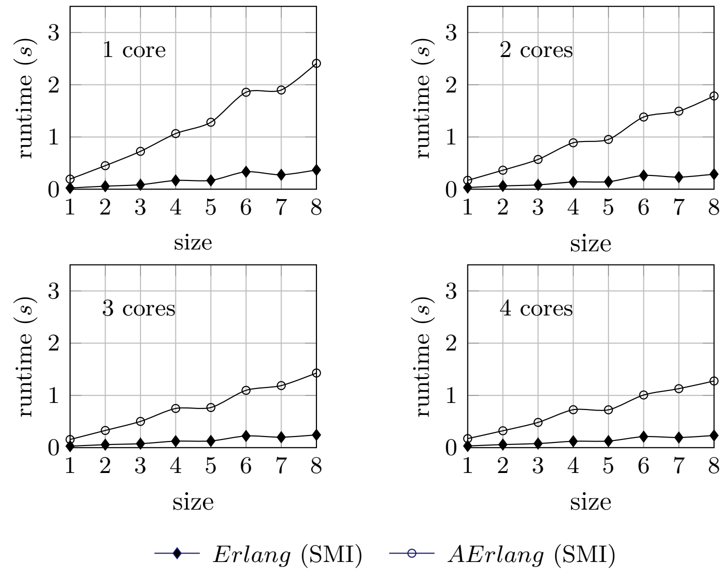

FIGURE 5.10: *AErlang* vs *Erlang*: scalability over problem sizes.

We also ran the *Erlang* and *AErlang* programs for the SMI problem, while ranging the size of the instances from 1 to 8 thousands and the number of cores from 1 to 4. Figure 5.10 shows the execution times of the two programs (in seconds) where each data point is the average of one hundred runs. We notice that the performance gap between *AErlang* and *Erlang* reduces when increasing the number of cores.

Summarizing, we can conclude that introducing the attribute-based programming abstraction leads to a performance overhead that can be considered acceptable for some application scenarios, especially if one considers the prototype nature of our *AErlang* implementation. The experimental results confirm that nevertheless the scalability provided by the underlying runtime system is not significantly affected.

5.6.3 Broadcasting vs. Pushing

In this section we report the experimental results aimed to compare the performance of the push and broadcast operating modes. We used SMA instances with size varying from 100 to 500, and we ran the *AErlang* programs one hundred times for each size, and then we reported the average values.

The hardware environment used for these experiments is an idle local workstation equipped with 128 GB of memory, a dual Intel Xeon processor E5-2643 v3 (12 physical cores in total) clocked at 3.40 GHz. The operating system is a 64-bit generic Linux kernel version 4.4.0, the used software is *Erlang*/OTP version 19.1.

Figure 5.11 shows the performance runtime (expressed in seconds on a logarithmic scale) of the broadcast and push forwarding policies while varying the problem size. We observe a large performance gap between the two policies. The execution time of the push strategy grows linearly with the problem size,

whereas the broadcast explodes. In particular, with problem sizes of 100, 200 and 300, the time ratios between the two policies are roughly 15, 190 and 750, respectively. For problem instances larger than 300, broadcast frequently timed out after 1000 seconds. By contrast, the push strategy can handle the rest of the experimental dataset with a runtime overhead slightly larger than 3 seconds.

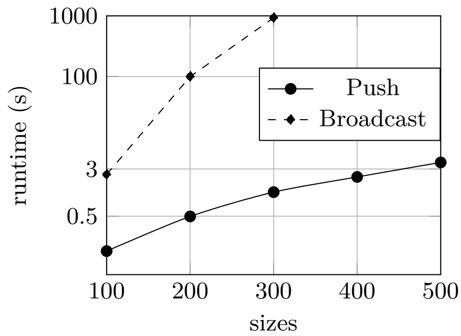

FIGURE 5.11: Varying problem sizes.

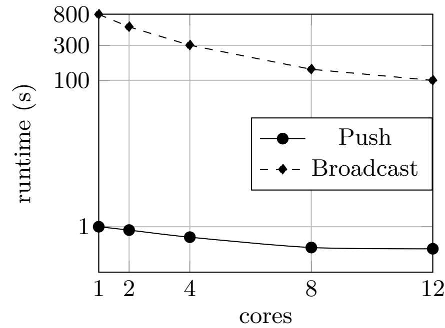

FIGURE 5.12: Varying CPU cores.

A further experiment was conducted by fixing the problem size at 200 pairs of communicating entities and varying the number of cores. The experimental results are shown in Figure 5.12 where the performance runtime is reported on a logarithmic scale, similarly to Figure 5.11. We can notice that both policies follow the same trend as they are both able to achieve a certain speed up when the number of cores increases. A difference is that the push strategy seems to stabilize when considering a number of cores varying from 8 to 12, whereas broadcast keeps a decreasing trend. Similarly to Figure 5.11, the runtime ratio between the two policies is quite large, in fact it goes from a speedup factor of 191 on 12 cores (as shown in Fig. 5.11), to 794 on a single core.

To analyse the factors behind runtime overhead, we instrumented the *AErlang* program to collect further details. Table 5.6 reports the information extracted during the execution of the *AErlang* program on problem sizes varying in the interval 100-300. In particular, the columns in the mid-section of Table 5.6 show the average execution times in seconds, the number of messages, the acknowledgement messages due to counting, the filtering operations, and the total size of messages. The results attest both communication and computational overhead. The difference between the number of exchanged messages (and thus the size of messages) is indeed not negligible. Moreover, exchanging more messages has the effect of increasing the local computational load, because processes have more messages to filter.

On the other hand, as pointed out in Section 3.2, using modes other than broadcast may result in an increased likelihood of inconsistencies between the local environment of a process and the replica stored globally. To further investigate this aspect, we have compared the quality of the output under the two operating modes. In particular we measured the size of the matching and the level of satisfaction of partners, i.e., the similarity between one's own preferences and the attributes of partners calculated as a weighted sum. As reported in the two rightmost columns of Table 5.6, we observe no significant gap between push and broadcast. This confirms that the push operating mode is definitely more

size	mode	time (s)	msgs (k)	acks (k)	filtering ops (k)	msgs size (MB)	matching size	satisf. level
100	push	0.13	13	4	4	3	88.6	496.8
	broadcast	2.04	1149	20	368	352	88.6	496.9
200	push	0.49	50	17	17	11	176.0	988.8
	broadcast	99.84	8919	79	3051	2835	175.8	988.2
300	push	1.25	113	37	37	24	265.0	1494.0
	broadcast	929.10	30600	181704	10510	9503	265.1	1493.6

Table 5.6: Push vs. Broadcast comparison details.

appropriate han broadcast for the considered scenario, since it yields considerable performance speedup without compromising the quality of the computed solution.

5.6.4 Broadcasting vs. Push-Pulling

The goal of the experimentation described in this section is to compare the efficiency of the push-pull and broadcast operating modes on the interactive market case study. We generate the datasets according to the following parameters. We considered systems of 20 users, 200 overall different items to exchange with randomly generated prices between 10 and 50, size of the wish list 20 or 40, size of the sale list 20. For each user, we allocate an initial budget sufficient to buy all items of interest. We performed this experiment on the same machine as Sec. 5.6.3.

We let the system run up to a *market time*, i.e., essentially a timeout on the next successful sale, of 10s and 20s. We stop taking the measures either after the given market time, or when the wish list of even a single user gets empty. This last condition avoids bias against broadcast, because when the wish list of a user is empty the system keeps generating useless advertisement messages.

The main cause of concern when running a system in push-pull mode is in the possible delays when updating an attribute or a receiving predicate in the message broker (see Sect. 5.3). Such a circumstance, however, would be unlikely when using a single machine for experiments, due to the very short time window occurring between the local update and the update in the message broker. To reproduce more realistic scenarios, we simulated network delays by artificially slowing down any communication among the processes themselves, and between processes and the message broker. We considered delays of 0ms, 2ms, 20ms, and 40ms [74]. For each combination of the above parameters we generated ten instances, and run each instance ten times.

To evaluate the broadcast and push-pull operating modes in a realistic setting, we compared them in terms of *cost* and *quality* of the service. Intuitively, a service provider implementing a platform for the interactive market may want to cut on communication costs by limiting the overall number of exchanged messages. On the other hand, to be motivated to use the service, a seller would like an advertisement to lead to a successful sale, whenever possible. Under the considered operating modes,

we thus measured: (a) the number of messages per successful sale, and (b) the number of successful sales.

Tables 5.7 and 5.8 report the average measures considering the size of the wish list equal to 20 and 40, respectively. We report the number of exchanged messages (expressed in thousands), the number of sales, and the number of messages per sale. We note that mode push-pull requires less (almost one order of magnitude) exchanged messages per sale with respect to broadcast, while at the same time not affecting the overall number of successful sales. Interestingly, the number of sales under push-pull mode is actually slightly greater than in broadcast, due to the fact that in broadcast mode processes spend some time in filtering inbound messages instead of focusing on the market activity. The above considerations are consistent over the considered network delays.

t=10s	delay											
	0ms			2ms			20ms			40ms		
mode	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s
push-pull	2.5	161.7	15.7	2.6	165.5	15.6	2.6	162.7	15.7	2.5	161.2	15.7
broadcast	21.4	154.3	138.6	21.8	158.6	137.1	21.9	159.5	137.3	21.9	159.9	137.2

t=20s	delay											
	0ms			2ms			20ms			40ms		
mode	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s
push-pull	2.5	163.3	15.6	2.6	165.9	15.5	2.6	163.0	15.7	2.5	161.4	15.7
broadcast	21.5	156.7	137.2	21.7	158.8	137.0	21.9	160.0	137.2	21.9	160.0	137.3

Table 5.7: Experimental results (wish list size = 20).

t=10s	delay											
	0ms			2ms			20ms			40ms		
mode	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s
push-pull	5.7	314.2	18.3	5.8	310.8	18.6	57.9	309.8	18.7	5.7	306.7	18.7
broadcast	39.4	285.6	138.0	40.7	298.0	136.6	40.9	300.2	136.4	40.9	300.5	136.3

t=20s	delay											
	0ms			2ms			20ms			40ms		
mode	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s	msg	sales	m/s
push-pull	5.7	314.1	18.3	5.8	311.7	18.6	5.8	308.5	18.7	5.7	307.1	18.7
broadcast	39.2	283.2	138.4	40.6	297.7	136.4	40.9	300.7	136.1	41.0	301.2	136.0

Table 5.8: Experimental results (wish list size = 40).

5.6.5 AErlang broadcast vs. *ABEL* broadcast

In this section we compare *ABEL* with *AErlang* operating in broadcast mode. This is because *ABEL* always broadcasts messages. The difference between the two is that *AErlang* uses one single broker (featuring parallel message forwarding), whereas *ABEL* parameterizes with the number of brokers (i.e., tree nodes).

We take the same set of input graphs reported in Section 3.4.2 (see Chapter 3) and ran the *AErlang* program (see Figure 5.6) to color each graph. Table 5.9 shows the execution times, the number of colors, the number of rounds, the number of messages, and the total size of messages. All the numbers reported in Table 5.9 are the average over 50 runs.

Graph	Time (s)	#C	#R	msg (in milli.)	msg size (in MB)
flat300_28_0	2.6	48	48	2.8	6,752
dsjc500.1	1.79	20	20	3.1	3,099
will199GPIA	2.71	9	19	5.7	2,971
dsjc1000.1	25.3	32	32	19.9	35,044

Table 5.9: The results of graph coloring in AErlang.

For the comparison, we also report the original results of *ABEL* in Table 5.10 where the execution times are shown for 1 and 31 brokers.

Graph	Times (in seconds)		#C	#R	msg (in milli.)	msg size (in MB)
	T = 1	T = 31				
flat300_28_0	37.4	13.6	47.7	151.7	4.67	22,229
dsjc500.1	29	9	19.6	69.7	5.58	10,449
will199GPIA	71.9	20.4	9.1	165.2	15.9	14,734
dsjc1000.1	372.1	98.8	31.5	144	46.3	156,966

Table 5.10: The results of graph coloring in ABEL.

We can see that the performance of *ABEL* is about one order of magnitude worse than *AErlang* broadcast, even when it uses up to 31 brokers. About the other metrics, *AErlang* program uses less messages and bandwidth to color the same set of input graphs, but at the same time returns the similar number of colors. Furthermore, the number of execution rounds and colors used by *AErlang* are the same for all runs of each input, while *ABEL* varies. This implies that the proposed algorithm, implemented in *AErlang* is deterministic. Indeed, *AErlang*'s synchronization constructs enabled vertices to agree on the common coloring status (i.e., the colors taken, the number of neighbors that completed the coloring process) with their neighbors, before proceeding with the next round. Moreover, the starting of a new round is autonomous in *AErlang* processes. By contrast, in the *ABEL* program, vertices indirectly influence the starting of new rounds at others.

Nevertheless, with the application of some optimization strategies, *ABEL* performance can be comparable to that of *AErlang*. Table 5.11 reports the results of *ABEL* program (using 31 brokers) where we disabled the message delivery at 'done' vertices, plus constraining the interface to eliminate the inclusion of neighbors lists in every message (see Section 3.4.2, Chapter 3).

Graph	Time (s)	#C	#R	msg (in milli.)	msg size (in MB)
flat300_28_0	5.9	47.5	150.5	4.72	1,444
dsjc500.1	4.7	19.5	72.3	5.69	1,781
will199GPIA	12.4	9	166.2	16.3	5,089
dsjc1000.1	35.6	31.4	144.6	46.2	14,452

Table 5.11: Optimizing the results of graph coloring in ABEL.

In comparison with the best results of *ABEL*, *AErlang* broadcast still performs about 2 to 4 times better, despite the larger consumption of the bandwidth. The fact that *AErlang* exposes a larger message size is because it considered registered attributes as processes interfaces. In the considered case study, this means *AErlang* has included the neighbors lists of vertices in every messages.

The difference between *AErlang* and *ABEL* performances can also be inferred by looking at their implementation details. In *ABEL*, components features parallel processes which relies on a local coordinator for interleaved executions. In the meanwhile, message passing occurs via two steps: a request of a fresh id for the message to be sent, and the actual send of the message to the connected broker. While requesting fresh ids are independent between components, message passing in *ABEL* is sequentialized. In *AErlang*, message passing is inherently asynchronous and there is no need of fresh ids.

5.7 Concluding Remarks

Our prototype language extension, *AErlang*, is a middleware enabling attribute-based communication among *Erlang* processes, with the aim of preserving *Erlang*'s scalability. *AErlang* plays the role of global process registry which allows processes to register and update their attributes. It also takes charge of forwarding messages from senders to receivers by evaluating the predicates they supply. Currently, *AErlang* implements four message forwarding strategies with different performance and reliability: broadcast, pull, push, and push-pull. In broadcast mode, the system immediately broadcasts any outbound message from any sender; at the other end, each receiver individually evaluates both the sending and the receiving predicates to discard irrelevant messages. In pull mode, the system performs an early check on the receiving predicates of the interested receivers and only forwards outbound messages from selected senders; the messages are then further filtered by the receivers according to the sending predicate. In push mode, the system checks the sending predicate and only forwards the outbound message to selected receivers; the messages are then further filtered by each receiver separately according to the receiving predicates. In push-pull mode, the system checks both the sending and the receiving predicates, then forwards the message to the appropriate peers.

Unlike the approach in Chapter 3 that aims at having a one to one correspondence between *ABEL* and *AbC* constructs, an *AErlang* process is an *Erlang* process with support for multicast and synchronization. A program written using *AErlang* may deviate from an *AbC* specification for two main reasons: components have a single thread of control and messages sent to a group of receivers are not ordered. For the former, programmers have to rely on process mailbox and the reactivity nature of processes to properly code the desired behaviour. Because of the latter, programmers have to mix synchronization code with functionality code when processing delivered messages in mailboxes. From *AbC* point of view, an interaction protocol that works can not be directly rendered as an *AErlang* program. Rather, it is required to reconsider the actual interactions among *AErlang* processes.

We have evaluated the efficiency and scalability of our *AErlang* approach with a number of experiments that compared the runtime performance of actor-based implementations for a known solution to a hard matching problem. We have shown that the overhead resulting from the new communication primitives is acceptable, and our prototype successfully preserves *Erlang*'s efficiency and scalability.

We have also implemented a variant of the above matching problem that requires a more involved interaction pattern and have compared it with an ad-hoc parallel version based on adaptive search implemented in X10 that can scale very well when increasing the number of cores. The experimental results have shown that our prototype does not currently scale well when increasing the number of cores. However, *AErlang* does indeed scale considerably well on large instances, whereas these turn out to be progressively out of reach for the algorithm based on adaptive search implemented in X10. We have also studied the impact of using different strategies for exchanging messages and evaluating the sending and receiving predicates that determine information exchange.

In our view, the choice of the forwarding strategy does depend on the specific class of problems under consideration. Generally, the broadcast mode is better suited for modelling highly-dynamic systems where attributes and predicates change frequently, but it requires a high number of message exchanges, and thus may not be suitable to large systems. On the other hand, the push mode is appropriate for systems whose attributes do not change frequently, while push-pull works well when also the predicates are quite static. In our experiments, broadcast turns out to be affected by a considerable computational and communication overhead that makes it not suitable to large-sized systems. In contrast, we have seen that, for some classes of systems, alternative operating modes (such as push or push-pull) can efficiently compute solutions of comparable quality, and are therefore definitely more appropriate.

As a final remark, we think that the proposed approach of *AErlang* can be applied to other concurrent languages such as Scala, Elixir as these share same concurrency model with *Erlang*. Specifically, the extension of communication primitives to their attribute-based version can be done for these languages. On the other hand, the design of message dispatching policies are quite general. It would be interesting to see if enriching *ABEL* with some messaging policies can improve its performance.

Chapter 6

Related Works

The general theme of our work has touched on several lines of researches, particularly related to other communication models with mechanisms for establishing group communication among processes and to other implementations of the AbC calculus.

6.1 Communication and coordination models.

Separating the functional part from the coordination mechanisms has been proposed to model complex interactions in some classes of multi-agent systems. Coordination models such as [28, 69, 112] allow the programmer to (compositionally) describe the communication patterns without considering the local behaviour of the components. In these approaches, the focuses of programming is at identifying communication patterns between participating agents and to define the aggregated and conditional behavior for each communication. Thus, communication is treated as first class citizens which can be manipulated by programming constructs. In comparison, $AErlang$ and $ABEL$ (being based on attributes) embrace the relationship between the exposed features of the components and their possible interactions. The communication patterns are naturally described by logical predicates over the attributes, thereby the behaviour of the system is the result of the interaction between individuals.

The ability of a process to reacts based on a set of messages was proposed in [64]. They introduced the notion of activators and receptionists. An activator is a command that blocks the executing process until a certain set of messages satisfying a boolean condition arrives. Receptionists are entities communicated in an inquiry message, and used by the activator command as the reply destinations for responses. In [70] these two concepts are merged in a single command, namely multicall, that explicitly handles the creation of the activator and the receptionists for the caller, in addition the command allows aggregating the received messages. In both models, the process may unblock without having to wait for responses from all the recipients of the inquiry message. The counting constructs of

AErlang share a similar idea, i.e., they allows a process to multicast an inquiry and collects responses from a set of processes. However, a process on waiting for a number of replies can also react to each response. The unblocking condition is based on counting the number of response instead of the satisfaction of the boolean condition over messages contents.

ActorSpace [15] is a programming paradigm which extends actor-based point-to-point communication to group-based communication. The model utilizes pattern matching on actors attributes to select groups of receivers. A sender can specify such groups by patterns represented as regular expressions over static actors attributes. The pattern is interpreted at run time to determine the intended receivers. However, only senders can select their partners; receivers have no control on incoming messages. In addition, ActorSpace provides primitives for explicit control of passive containers of actors called spaces that represent contexts for matching patterns. Spaces are first class entities that can be created and destroyed by programmers and may be nested or overlap. By contrast, attribute-based communication paradigm abstracts from the notion of explicit groups. Interacting partners are selected dynamically depending on the evaluation of the send and receive predicates.

Linda [68] is one of the first coordination languages for programming distributed systems. Processes communicate via a shared memory called tuple space, where any process can add, read or withdraw data objects. The communication model is time and space decoupling, and allows processes to exchange information regardless of the availability of interacting entities in the interim. Although simple and intuitive, tuple spaces have not been widely used. A comprehensive comparison between tuple spaces implementations can be found in [38]. Attribute-based communication, instead is categorized as a message passing model. Information exchange thus only happens at the communication time, and requires the presence of the involved agents.

6.2 Implementations of AbC

There has been other work aiming at extending well established languages to enrich them with the primitives of the *AbC* calculus and thus to enable programmers to directly use attribute-based communication.

AbaCus [26] is based on Java and, like *AErlang*, relies on a centralised broker that keeps track of all components, intercepts every message and forwards it to all registered components. It is then the responsibility of each component to decide whether to receive or discard the message. In contrast, in *AErlang* (depending on the operating mode), to avoid broadcast, the broker relies on an attribute registry, where components store their attribute values for message filtering.

GoAT [19] extends Go [9]; it relies on an attribute-based programming interface that is parametric with respect to different brokering distributed infrastructures for message exchange to mediate the interaction between components. The distributed infrastructures, originally proposed in [20] and

therein proved correct with respect to the original centralised semantics, are instrumental to avoid the bottleneck of a single centralised broker. It would be interesting to experiment with the distributed implementation of the broker also in *AErlang* and to study the effect of combining message filtering and distributed broker.

ABEL and *GoAT* shares some similarity in terms of implementation as both replicated the idea of the distributed coordination infrastructure [20]. *GoAt* provides the ring and the cluster shapes, in addition to the tree-based infrastructure. At the programming levels, non deterministic choices are (as traditionally) implemented as if then else constructs, process recursion is implemented as infinite loop in both *AbaCus* and *GoAt*. In this respect, *ABEL* APIs is closer to the calculus than that of *GoAT* and *AbaCus*. This fact could make program transformations easier such as automatic translating *AbC* specifications into *ABEL* programs, or develop domain specific languages for *AbC* on top of *ABEL*.

jRESP [5] adds attribute-based communication to Java but takes the SCEL paradigm [51] as starting point rather than *AbC* and is more oriented towards programming autonomic and adaptive systems. *jRESP* designates ports with specific roles at nodes (or components) for communication. Nodes agreeing to interact via a port can use any of the communication protocols (such as broadcast via a central server, multicast or point-to-point) supported by the chosen port. *jRESP* also requires that all messages are delivered to all components; this choice simplifies the design and the implementation of the message broker but introduces significant communication overhead, especially in large systems.

Erlang has been used as the host language for incorporating domain specific abstractions to deal with multi-agents and self-adaptive systems [55, 90, 111]. Among these, we touch on ContextErlang [111], an extension of *Erlang* for context-oriented programming [80]. ContextErlang extends *Erlang*'s gen_server behaviour with context_agent whose callback functions can be overridden by (functions implementing) variations at runtime. During operation, a context change triggers the activation of the corresponding variations, which leads to changing the behaviour of context_agents. The difference from our approach is that we exploit exposed attributes, thus processes can adapt their behaviour implicitly using predicate-based message passing. In practice, we can model context-awareness by updating attributes when receiving information from appropriate sensors. Extended communication primitives for *Erlang* have been considered also in other works. The most closed one is JErlang [106] that extends *Erlang*'s receive operations into a receive-like join construct inspired from Join-Calculus [61]. Differently from *AErlang*, entirely based on source-to-source translations, JErlang embeds the new receive primitive with low-level optimisations within an altered version of *Erlang*'s VM.

6.3 Verification

Analysing concurrent systems by relying on process algebraic specifications has been considered in [76]. Other works, [97, 113] have taken an approach similar to ours and perform verification by translating a specification formalism into a verifiable one that could make use of existing model checkers. There are a few works towards the verification of ensemble-based systems which are of some relevance. In [50], the authors translated a simplified version of SCEL into Promela, the input language of the SPIN model checker. Similarly, [89] translated HELENA specifications into Promela specification and proved that the translation preserves satisfaction of LTL formulae. Both tools exploited the SPIN model checker for verifying liveness properties and deadlock freedom. The SCEL operational semantics has also been implemented in Maude [33] for assessing qualitative and quantitative properties.

Symbolic verification of systems expressed in a variant of AbC has been considered in [44]. The basic idea is to translate the system specifications into non-deterministic C programs to be analysed with existing general-purpose SAT-based techniques.

Chapter 7

Conclusions and Future Works

In this thesis, we studied the merits of attribute-based communication with a focus on programming and verification. The general benefits of using predicate-based message passing as a novel paradigm are by now rather clear; it has indeed been shown in a number of papers [21, 23, 93] that components interactions benefits significantly from the use of predicates. However, implementability, scalability, performance and proof of correctness of systems specified by resorting to the new communication primitives are still a concern.

For programming and experimenting with attribute-based communication, we have initially instantiated the novel programming abstractions on top of *Erlang*. This language was chosen for its lightweight concurrency model and its advanced features encompassed in battle-tested standard libraries. Our work considered two different approaches in providing language supports for the novel paradigm.

First, starting from the semantics of *AbC* - a kernel calculus that formalizes attribute-based communication, we designed and implemented a domain-specific framework for *AbC* in *Erlang*, namely *ABEL*. Its programming constructs are designed to mimic *AbC* primitives. Thus, programming in *ABEL* is similar to specifying models in *AbC*. Furthermore the direct correspondence between *AbC* specifications and *ABEL* programs paved the way to formal verification of programs via model checking the original *AbC* specification. Indeed, the explicit-state model checker [81] helped us to verify early designs of the case studies and to come up with correct specifications. From the verified specifications we have then derived the *ABEL* programs introduced Chapter 3.

To implement the broadcasting parallel operator among *AbC* components while guaranteeing the expected synchronous semantics we resorted to a sequencer based protocol that takes care of delivering messages according to a total ordering [20]. The parallel operator among processes inside components is instead implemented by using a reactive state machines associated to the component that coordinates input and output actions of parallel processes. We have provided a number of experiments to analyse

the performance of *ABEL* programs. These experiments showed that the cost of running programs that fully respect *AbC* semantics is quite high.

In the second approach, we sacrificed the semantics of *AbC* calculus and more simply extended *Erlang* processes with programming abstractions for attribute-based communication. We thus proposed *AErlang*, a prototype that enables attribute-based communication among *Erlang* processes. *AErlang* conveniently combines the benefits of the novel paradigm with the efficiency and scalability of *Erlang*; in addition, it supports several policies for arbitrating messages exchanges which depending on the nature of attributes and predicates changes can cut down communication and computational overheads of broadcast. Our experiments with *AErlang* showed that the approach copes well with the main sources of complexity of collective adaptive systems, such as anonymity, open-endedness, adaptivity, and their large size. Thus, it allows programmers to concentrate on the essence of the system to implement, by relieving them from the burden of working out low-level details to be approached on a case-by-case basis.

We also considered the issue of automatic verification and based the reasoning about *AbC* specifications on the UMC model checker [81]. UMC relies on doubly-labelled transition systems as a formal model and on a customized temporal, branching time logic called UCTL. We have proposed a structural translation that brides the gap between the high-level syntax categories of *AbC* and the UMC modeling language. Specifically, the specification of a given *AbC* system is mapped into a unique UML-like state machine, where interesting component attributes and communication actions are made observable. We have showed that our approach can be helpful for identifying emerging properties and unwanted behaviours on a number of non-trivial case studies.

Future Works

Much work remains to be done to make our prototype implementations more attractive in practice. For example, an extensive evaluation on arbitrarily large instances that use complex predicates and frequently changing attributes would be useful to assess the overall robustness. An in-depth performance evaluation to understand whether the large size of the system stresses the underlying scheduling mechanisms would be very helpful. A systematic evaluation of the cost of predicate handling would be very useful to improve efficiency. Indeed, since predicates can have an arbitrary complexity, their evaluation may add a significant overhead, and efficient predicate evaluation is known to be non-trivial [60]; looking for more efficient ways to handle it is thus very important. A comprehensive experimentation by varying the number of attributes, the size of the domains, the frequency of their updates, and their probability distribution would be very useful to devise different handling strategies according to a finer-grained classification of the attributes.

We think that attribute-based communication can be implemented in other concurrent languages by following our *Erlang* approaches, in particular, actor languages such as Scala, Elixir. Generally, extending the paradigm across different programming environments would certainly allow a deeper investigation on the effectiveness of attribute-based communication.

For improving *AErlang*, we notice that some optimization strategies can be applied. First, *AErlang* predicates are represented in form of strings, which requires parsing every time a communication action taking place. We may gain some efficiency by representing predicates as anonymous functions (similar to *ABEL*). In addition, *AErlang*, by default considers registered attribute names as processes interfaces, and thus includes this portion of the environment in every messages. It is not always the case that receiving processes would use this portion of sender's environment. We would like to study to what extent we can reduce the message size based on remote attribute names appearing in receiving predicates (e.g., using static analysis). Second, a more serious treatment of the declared interfaces is helpful to classify processes into groups. Currently, *AErlang* keeps all attributes in a single, flat table. The cost of scanning the whole table for selecting partners can be huge. Registered processes however, can be grouped based on their public interface. Each group can be taken care by different process registries and message brokers. Message forwarding among processes would then be able to work out if a sent message is internal to a group or should be directed to another group, possibly in a way similar to [39]. Along this direction, message forwarding policies could be re-designed and re-evaluated. Moreover, language abstractions that support for expressing useful communication and synchronization patterns [18, 61, 63] can be added. Last but not least, we would like to study a formal semantics for *AErlang*, possibly separating out its operating modes. This would general increase the confidence of writing and reasoning about *AErlang* programs. One may provide a formal semantics for *AErlang*, e.g., following the approach in [111] which distills a formal semantics for ContextErlang from an implementation. In our case, this task would be simpler when considering separately *AErlang* operational modes. On the other hand, we would like to study the approach in [71] which provides a semantics of a communication layer, independently from actors computations. The main advantages is that their approach separates functional concerns from communication concerns (which is usually complex), and allows composing simple communication patterns into more expressive ones.

For improving *ABEL*, we plan to equip it with an asynchronous semantics to relax the total ordering of message delivery. We would like also to automatically generate *ABEL* code from the verified *AbC* specifications. This would facilitate a model-driven development approach for the *AbC* paradigm, similar to the study [105] done for the Spi-calculus and its implementations in Java.

For automatic verification, we would like to port our approach to other model checkers, different from UMC. The tool selection depends on some major factors such as the expressiveness of modeling language, the support for property definition and verification performance. A recent survey [99] for seven well-known verification frameworks suggested us that nuSMV [42], mCRL2 [75], CADP [66] provide best support in terms of property definition language (i.e., both CTL and LTL), while the

nuSMV tool outperforms the others in terms of performance. It thus is interesting to provide mappings from *AbC* into these selected toolset. Furthermore, the program verification approach used in [44] for a variant of *AbC* is promising and worth to considered.

Alternatively, since *AErlang* is automatically translated to *Erlang* (Sect. 5.4), any general-purpose verification technique for *Erlang* can in principle be applied to *AErlang*. Existing verification techniques for *Erlang* are based on traditional model checking, and work either directly on the program's source code [62, 78], or on process-algebraic specifications obtained by abstracting from the program [77]. A more sophisticated technique combines infinite-state model checking and static analysis via abstract interpretation, but assumes a restricted version of the language and only considers safety properties [56].

Further evaluations can be conducted by extending *AbC* with a probabilistic-based specification, i.e., the communication among interacting entities may be regulated by probability distribution functions expressing the probability of success in matching exposed users' attributes, similarly to the preferences studied in [35]. This leads to probabilistic reasoning of *AbC* and it is possible to analyze the stochastic interactions between communicating entities according to the probability of matching predicates with attributes. Properties of interest in this setting can be the probability of reaching a matching within a certain number of rounds, or the expected number of rounds to reach a stable matching. In this direction, we would like to also consider techniques developed for modeling and analysing probabilistic systems [16, 17, 118].

Finally, we would like to remark that the synchronous semantics of *AbC* appears to be too demanding for many applications and to be a serious limitation when large scale distributed systems are considered. In fact, some of the case studies showed that a total ordering of the exchanged message in a multicast is not necessary. For example, vertices in our graph colouring scenario can send messages asynchronously without changing the final outcome and in the stable marriage scenario, the result is the same irrespectively of the order women process messages. In a forthcoming paper, we will consider an asynchronous semantics of *AbC*, that by relaxing the *total ordering* requirement will give rise to more efficient computation.

Appendix A

Erlang programming

A.1 Erlang program for SMTI problem

The *Erlang* program that implements the matching algorithm for the SMTI problem (see Section 5.5.1) is reported in Figure A.1. This program was used for the experiments that lead to the results reported in Sect. 5.6.1 (see Table 5.5).

In comparison with SMTI-aerl (see Figure 5.3), SMTI-erl is more intricate and longer (i.e., 37 vs 47 lines of code). Moreover, the *Erlang* program only works on the input where identifiers are provided, in fact we use the global module for registering agents identifiers and for sending messages. In case the input is in attribute format, and in general in case of more complex interaction predicates, there is no obvious way to encode the attribute-based interaction directly in *Erlang*.

A.2 Erlang program for SMI problem

Figure A.2 presents the *Erlang* program that was used as a basis for the experiments about scalability whose outcome is reported in Sect. 5.6.2 (see Figures 5.9 and 5.10).

```
1  man(Prefs,Id,P) ->
2    [H|T] = Prefs,
3    Cnt = [global:send(X,{propose,Id}) ||
4           X <- H],
5    NP = receive_count(Id,length(Cnt),P),
6    case NP of
7      none ->
8        man(T,Id,P);
9      _ ->
10       receive
11         {goodbye,NP} ->
12             man(Prefs,Id,none)
13       end
14   end.
15
16 receive_count(_,0,P) -> P;
17 receive_count(Id,Cnt,P) ->
18   receive
19     {yes,W} ->
20       case P of
21         none ->
22           global:send(W,{confirm,Id}),
23           receive_count(Id,Cnt-1,W);
24         _ ->
25           global:send(W,{busy,Id}),
26           receive_count(Id,Cnt-1,P)
27       end;
28     no ->
29       receive_count(Id,Cnt-1,P)
30   end.
```

```
1  woman(Prefs,Id,P) ->
2    receive
3      {propose, M}  ->
4        case bof(Prefs,P,M) of
5          true ->
6            global:send(M,{yes,  Id}),
7            receive
8              {confirm,M} ->
9                global:send(P, {goodbye,Id}),
10               woman(Prefs,Id,M);
11             {busy,M} ->
12               woman(Prefs,Id,P);
13         false ->
14           global:send(M,no),
15           woman(Prefs,Id,P)
16       end
17   end.
```

FIGURE A.1: Stable Marriage with preference lists in *Erlang* (SMTI-erl).

```
1  man(Prefs,Id) ->
2    [H|T] = Prefs,
3    global:send(H,{propose,Id}),
4    receive
5      no ->
6          man(T,Id)
7    end.
```

```
1  woman(Prefs,Id,Partner) ->
2   receive
3    {propose, Man}  ->
4      case bof(Prefs,Partner,Man) of
5        true ->
6          global:send(Partner,no),
7          woman(Prefs,Id,Man);
8        false ->
9          global:send(Man,no),
10         woman(Prefs,Id,Partner);
11     end
12  end.
```

FIGURE A.2: Classical Stable Marriage with preference lists in *Erlang* (SMI-erl).

Appendix B

An *AbC* specification

We provide the specification for the bottom up solution of the stable marriage problem mentioned in Chapter 4.

B.1 Bottom-up Stable marriage

```
component Man
 attributes: id, partner, m1, m2, pw1, pw2, cw1, cw2, queue
 observables: partner
 behaviour:
   let {
     Q := (x = ‘yes’)(x,y,z,t).[queue := queue+[[$x,$y,$z,$t]]]Q

     H := (’propose’,this.id,this.m1,this.m2)@(pw2 = this.pw2).HF
     HF :=
  % accept current partner even if not best
          <queue /= [] and partner = 0>
          (‘confirm’,this.id,this.m1,this.m2)@(id = this.queue.hd[1]).
          [partner:= queue.hd[1], cw1 := queue.hd[2], cw2 := queue.hd[3],
          queue := queue.tl]HF
            +
  % if a better match is also arrived prepare send bye and swap
         (<queue /= [] and partner /= 0 and (queue.hd[3] = pw2 and cw2 /= pw2)>
         (‘bye’,this.id)@(id = this.partner).
         [partner := 0,cw1 := 0, cw2 := 0]HS)
            +
  % notify toolate for delayed replies (not better then current match)
         (<queue /= [] and partner /= 0 and (queue.hd[3] /= pw2 or cw2 = pw2)>
         (‘toolate’,this.id)@(id = this.queue.hd[1]).
         [queue := queue.tl]HF)
            +
  % if abandoned by current partner, but queue not empty continue
         <queue /= []>(x = ‘bye’ and y = this.partner)(x,y).
         [partner := 0,cw1 := 0, cw2 := 0]HF
            +
```

```
% if abandoned by current partner, and empty queue restart with low expectations
      <queue = []>(x = 'bye' and y = this.partner)(x,y).
      [partner := 0, cw1 := 0, cw2 := 0]H
        +
% if empty queue but not best make better proposal
      <queue = [] and partner /= 0 and ch /= ph>
      ('propose',this.id,this.m1,this.m2)@(w1 = this.pw1 and w2 = this.pw2).HS

  HS :=
% accept current partner (just swapped, is surely the best)
      <queue /= [] and partner = 0>
      ('confirm', this.id,this.m1,this.m2)@(id = this.queue.hd[1]).
      [partner:= queue.hd[1], cw1 := queue.hd[2], cw2 := queue.hd[3],
      queue := queue.tl]HS
       +
% notify toolate for all other queued replies (cannot be better)
     (<queue /= [] and partner /= 0 and (queue.hd[3] /= pw2 or cw2 = pw2)>
     ('toolate', this.id)@(id = this.queue.hd[1]).
     [queue := queue.tl]HS)
        +
% receive first reply of this stronger requirment, bye ex "lower" partner
     <queue /= [] and partner /= 0 and (cw2 /= pw2 and queue.hd[3] = pw2)>
     ('bye', this.id)@(id = this.partner).
     [partner:= 0, cw1 := 0, cw2 := 0]HS
       +
% if abandoned by current partner, but queue not empty continue
    <queue /= []>(x = 'bye' and y = this.partner)(x,y).
     [partner := 0, cw1 := 0, cw2 := 0]HF
       +
% if abandoned by current partner and empty queue, then restart with low expectations
    <queue = []>(x = 'bye' and y = this.partner)(x,y).
     [partner := 0, cw1 := 0, cw2 := 0]H
  }
  init Q | H
end
```

```
component Woman
 attributes: id, partner, w1, w2, pm1, pm2, cm1, cm2,  queue
 observables: partner
 behaviour:
   let {
     Q := (x = 'propose')(x,y,z,t).[queue := queue+[[$x,$y,$z,$t]]]Q

     H :=
% accepts when being single
     <queue /= [] and partner=0>
     ('yes',this.id,this.w1,this.w2)@(id = this.queue.hd[1]).
        (
          (x = 'confirm')(x,y,z,t).
          [partner := queue.hd[1],cw := queue.hd[2],cb := queue.hd[3],
          queue := queue.tl]H
           +
          (x = 'toolate')(x,y).[queue := queue.tl]H
        )
      +
```

```
% compare two men, new man is better
     <queue /= [] and partner /= 0 and ((cm1 /= pm1 and queue.hd[2] = pm1)
      or ( cm1 = queue.hd[2] and cm2 /= pm2 and queue.hd[3] = pm2))>
      ('yes', this.id,this.w1,this.w2)@(id = this.queue.hd[1]).
             (
         % wait confirmation or toolate notice
              (x = 'confirm')(x,y,z,t).
                (
                 ('bye',this.id)@(id = this.partner).
                  [partner := queue.hd[1], cm1 := queue.hd[2], cm2 := queue.hd[3],
                  queue := queue.tl]H
                  +
                  % abandoned by now ex partner
                 (x = 'bye' and y = this.queue.hd[1])(x,y).[queue := queue.tl]H
                )
              +
              (x ='toolate')(x,y).[queue := queue.tl]H
             )
        +
% if the new man is not better, discard message
     <queue /= [] and partner /= 0 and (( cm1 = pm1 and queue.hd[2] /= pm1)
             or (cm1 = queue.hd[2] and pm2 /= queue.hd[3] and cm2 = pm2)
             or (cm1 = queue.hd[2] and cm2 = queue.hd[3]))>
             ()@(false).[queue := queue.tl]H
       +
% abandoned by current partner, reset status and continute
      (x = 'bye' and y = this.partner)(x,y).
      [partner := 0, cm1 := 0, cm2 := 0]H
    }
  init Q | H
end
```

Appendix C

A generated UMC model

We provide the generated UMC code for graph coloring scenario in order to show specific parts that required user manipulations.

C.1 UMC Model for Graph Coloring

Our implementation in practice translates for each component type. The specific code for components is then replicated from their type with supplied indexes. In the code below, $1 is the variable holding a component index, $2 is the variable holding a component name. User code is commented accordingly in the first transition of component.

The purpose of this section, apart from presenting a generated UMC model, is to highlight specific parts of the model that needed user manipulation. Since expressions and predicates which can be arbitrarily complex, currently there is no uniform way of encoding and thus enable translating them automatically.

```
Class System with niceparallelism is
Signals:
        allowsend(i:int);
        broadcast(tgt,msg,j:int);
Vars:
        RANDOMQUEUE;
        receiving:bool := false;
        pc :int[];
        bound:int[];
        -- attributes
        assigned;
        color;
        constraints;
        counter;
```

```
        done;
        id;
        nbr;
        round;
        send;
        used;
State Top Defers allowsend(i)
init -> SYS {-/
        pc := [[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]];
        for i in 0..pc.length-1 {
                self.allowsend(i);
        }}
 ----- Send -----
SYS.$2.s0 -> $2.s0 {
       allowsend(i)[receiving = false and i = $1 and
       (send[$1] = true and assigned[$1] = false) and pc[$1][0] = 1]/
        -- begin user code (implementing the min_color function)
        temp: int = 0;
        for v3 in 1..used[$1].length {
         if v3 < used[$1][v3-1] and temp = 0 then {
            temp = 1;
            color[$1] = v3;
         };
        };
        if temp = 0 then {
           color[$1] = used[$1].length + 1;
        };
        -- end user code
        target:int[];
        for j in 0..pc.length-1 {
        tumccounter : bool = false;
        for z in 0..nbr[$1].length-1 {
             if not tumccounter1 and (id[j]=nbr[$1][z]) then {tumccounter := true;}
        };
        if (tumccounter1) then
                { target[j] := 1} else {target[j]:=0;}
        };
        receiving=true;
        self.broadcast(target,[try,color[$1],round[$1]],$1 );
       OUT.sending($2,[try,color[$1],round[$1]]);
        --- attr update ---
        send[$1] := false;
        -- color[$1] := min;
        pc[$1][0] = 2;
}
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[pc[$1][0] = 2]/
        receiving=false;
```

```
          self.allowsend($1);
          pc[$1][0] = 1;
          bound[$1][0] := 0;
}
 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
          broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = try and
          id[$1] > id[j] and round[$1] = msg[2]) and pc[$1][1] = 1]/
          bound[$1][1][0] = msg[0];
          bound[$1][1][1] = msg[1];
          bound[$1][1][2] = msg[2];
          counter[$1] := counter[$1]+1;
          OUT.received($2,msg);
          pc[$1][1] = 1;
          bound[$1][1] := 0;
}

 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
          broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = try and
          id[$1] < id[j] and round[$1] = msg[2]) and pc[$1][1] = 1]/
          bound[$1][1][0] = msg[0];
          bound[$1][1][1] = msg[1];
          bound[$1][1][2] = msg[2];
          counter[$1] := counter[$1]+1;
          constraints[$1] := constraints[$1]+[bound[$1][1][1]];
          if constraints[$1].length > 1 then {
          for v1 in 0..constraints[$1].length - 1 {
          for v2 in v1+1..constraints[$1].length-1 {
          if constraints[$1][v1] > constraints[$1][v2] then {
                  temp:int := constraints[$1][v1];
                  constraints[$1][v1]:=constraints[$1][v2];
                  constraints[$1][v2]:=temp;}}
          };
          umctemp : int := 0;
          umctemplength : int := constraints[$1].length;
          umctempvec : obj;
          for v1 in 0..umctemplength - 2 {
          if constraints[$1][v1] /= constraints[$1][v1+1] then {
          umctempvec[umctemp] = constraints[$1][v1];
          umctemp = umctemp + 1;};
          umctempvec[umctemp] = constraints[$1][umctemplength-1];};
          constraints[$1]:= umctempvec;
          };
          OUT.received($2,msg);
          pc[$1][1] = 1;
          bound[$1][1] := 0;
}
```

```
 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = try and id[$1] > id[j]
        and round[$1] < msg[2]) and pc[$1][1] = 1]/
        bound[$1][1][0] = msg[0];
        bound[$1][1][1] = msg[1];
        bound[$1][1][2] = msg[2];
        round[$1] := msg[2];
        send[$1] := true;
        counter[$1] := 1;
        constraints[$1] := [];
        OUT.received($2,msg);
        pc[$1][1] = 1;
        bound[$1][1] := 0;
}

 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = try and id[$1] < id[j]
        and round[$1] < msg[2]) and pc[$1][1] = 1]/
        bound[$1][1][0] = msg[0];
        bound[$1][1][1] = msg[1];
        bound[$1][1][2] = msg[2];
        round[$1] := bound[$1][1][2];
        send[$1] := true;
        counter[$1] := 1;
        constraints[$1] := [bound[$1][1][1]];
        OUT.received($2,msg);
        pc[$1][1] = 1;
        bound[$1][1] := 0;
}

 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = donec and
        round[$1] >=msg[2]) and pc[$1][2] = 1]/
        bound[$1][2][0] = msg[0];
        bound[$1][2][1] = msg[1];
        bound[$1][2][2] = msg[2];
        done[$1] := done[$1]+1;
        used[$1] := used[$1]+[bound[$1][2][1]];
        if used[$1].length > 1 then {
        for v1 in 0..used[$1].length - 1 {
        for v2 in v1+1..used[$1].length-1 {
        if used[$1][v1] > used[$1][v2] then {
        temp:int := used[$1][v1];used[$1][v1]:=used[$1][v2];used[$1][v2]:=temp;}}
        };
```

```
        umctemp : int := 0;
        umctemplength : int := used[$1].length;
        umctempvec : obj;
        for v1 in 0..umctemplength - 2 {
        if used[$1][v1] /= used[$1][v1+1] then {
        umctempvec[umctemp] = used[$1][v1];
        umctemp = umctemp + 1;};
        umctempvec[umctemp] = used[$1][umctemplength-1];};
        used[$1]:= umctempvec;
        };
        OUT.received($2,msg);
        pc[$1][2] = 1;
        bound[$1][2] := 0;
}

 ----- Receive -----
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[tgt[$1] = 1 and (msg[0] = donec and
        round[$1] < msg[2]) and pc[$1][2] = 1]/
        bound[$1][2][0] = msg[0];
        bound[$1][2][1] = msg[1];
        bound[$1][2][2] = msg[2];
        round[$1] := bound[$1][2][2];
        done[$1] := done[$1]+1;
        send[$1] := true;
        counter[$1] := 0;
        used[$1] := used[$1]+[bound[$1][2][1]];
        if used[$1].length > 1 then {
        for v1 in 0..used[$1].length - 1 {
        for v2 in v1+1..used[$1].length-1 {
        if used[$1][v1] > used[$1][v2] then {
        temp:int := used[$1][v1];used[$1][v1]:=used[$1][v2];used[$1][v2]:=temp;}}
        };
        umctemp : int := 0;
        umctemplength : int := used[$1].length;
        umctempvec : obj;
        for v1 in 0..umctemplength - 2 {
        if used[$1][v1] /= used[$1][v1+1] then {
        umctempvec[umctemp] = used[$1][v1];
        umctemp = umctemp + 1;};
        umctempvec[umctemp] = used[$1][umctemplength-1];};
        used[$1]:= umctempvec;
        };
        constraints[$1] := [];
        OUT.received($2,msg);
        pc[$1][2] = 1;
        bound[$1][2] := 0;
}
```

```
 ----- Send -----
SYS.$2.s0 -> $2.s0 {
        allowsend(i)[receiving = false and i = $1 and
        (nbr[$1].length = counter[$1]+done[$1] and color[$1] != undefined and
        (not (false or color[$1]=constraints[$1][0] or color[$1]=constraints[$1][1] or
         color[$1]=constraints[$1][2] or color[$1]=constraints[$1][3] or
         color[$1]=constraints[$1][4] or color[$1]=constraints[$1][5] or
         color[$1]=constraints[$1][6] or color[$1]=constraints[$1][7] or
         color[$1]=used[$1][0] or color[$1]=used[$1][1] or
         color[$1]=used[$1][2] or color[$1]=used[$1][3] or
         color[$1]=used[$1][4] or color[$1]=used[$1][5] or
         color[$1]=used[$1][6] or color[$1]=used[$1][7])))
         and pc[$1][3] = 1]/
        target:int[];
        for j in 0..pc.length-1 {
        tumccounter : bool = false;
        for z in 0..nbr[$1].length-1 {
            if not tumccounter1 and (id[j]=nbr[$1][z]) then {tumccounter := true;}
        };
        if (tumccounter) then
                { target[j] := 1} else {target[j]:=0;}
        };
        receiving=true;
        temp:int = round[$1]+1;
        self.broadcast(target,[donec,color[$1],temp],$1 );
        OUT.sending($2,[donec,color[$1],temp]);
        --- attr update ---
        assigned[$1] := true;
        pc[$1][3] = 2;
}
SYS.$2.s0 -> $2.s0 {
        broadcast(tgt,msg,j)[pc[$1][3] = 2]/
        receiving=false;
        self.allowsend($1);
        pc[$1][3] = 3;
        bound[$1][3] := 0;
        pc[$1][3]:= 0;
})
```

Bibliography

[1] AbC2UMC: translating abc calculus into uml-like state machine. https://github.com/ArBITRAL/AbC2UMC.

[2] ABEL: an implementation of abc in erlang. https://github.com/ArBITRAL/ABEL.

[3] AErlang: programming attribute-based communication in Erlang. https://github.com/ArBITRAL/AErlang.

[4] Akka Toolkit. https://akka.io/.

[5] jRESP: Java Runtime Environment for SCEL Programs. http://jresp.sourceforge.net/.

[6] Orleans - Distributed Virtual Actor Model. https://dotnet.github.io/orleans.

[7] Scala Language. http://www.scala-lang.org.

[8] SMTI instance generator. https://github.com/dannymrock/SMTI-AS-X10.

[9] The Go Programming Language. https://golang.org.

[10] The Pony programming language. https://www.ponylang.io.

[11] The UMC verification framework. http://fmt.isti.cnr.it/umc.

[12] UMC Documentation. http://fmt.isti.cnr.it/umc/DOCS.

[13] A. Abdulkadiroğlu, P. A. Pathak, A. E. Roth, and T. Sönmez. The boston public school match. *American Economic Review*, 95(2):368–371, 2005.

[14] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[15] G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 23–32, 1993.

[16] G. Agha, J. Meseguer, and K. Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2006.

[17] G. Agha and K. Palmskog. A survey of statistical model checking. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(1):6:1–6:39, 2018.

[18] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In *Formal Methods for Open Object-based Distributed Systems*, pages 135–153. Springer, 1997.

[19] Y. A. Alrahman, R. De Nicola, and G. Garbi. Goat: Attribute-based interaction in google go. In *International Symposium on Leveraging Applications of Formal Methods*, pages 288–303. Springer, 2018.

[20] Y. A. Alrahman, R. De Nicola, G. Garbi, and M. Loreti. A distributed coordination infrastructure for attribute-based interaction. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 1–20. Springer, 2018.

[21] Y. A. Alrahman, R. De Nicola, and M. Loreti. On the power of attribute-based communication. In *Proc. of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 1–18, 2016.

[22] Y. A. Alrahman, R. De Nicola, and M. Loreti. Programming of CAS systems by relying on attribute-based communication. In *Proc. of the International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, pages 539–553, 2016.

[23] Y. A. Alrahman, R. De Nicola, and M. Loreti. A behavioural theory for interactions in collective-adaptive systems. *CoRR*, abs/1711.09762, 2017.

[24] Y. A. Alrahman, R. De Nicola, and M. Loreti. Programming the interactions of collective adaptive systems by relying on attribute-based communication. *CoRR*, abs/1711.06092, 2017.

[25] Y. A. Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, and R. Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1840–1845. ACM, 2015.

[26] Y. A. Alrahman and M. Loreti. AbaCuS: A Run-time environment of the AbC calculus. https://github.com/lazkany/AbC, 2016.

[27] S. Anderson, N. Bredeche, A. Eiben, G. Kampis, and M. van Steen. Adaptive collective systems: herding black sheep. http://booksprints-for-ict-research.eu/wp-content/uploads/2013/12/bs4ictrsrch-acs.pdf, 2013.

[28] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(3):329–366, 2004.

[29] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[30] J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

[31] J. Armstrong. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.

[32] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[33] L. Belzner, R. De Nicola, A. Vandin, and M. Wirsing. Reasoning (on) service component ensembles in rewriting logic. In *Specification, Algebra, and Software*, pages 188–211. Springer, 2014.

[34] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the Symposium on Operating System Principles (SOSP)*, pages 123–138, 1987.

[35] P. Biró and G. Norman. Analysis of stochastic matching markets. *International Journal of Game Theory*, 42(4):1021–1040, 2013.

[36] S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. AXD 301: A new generation ATM switching system. *Computer Networks*, 31(6):559–582, 1999.

[37] I. Brito and P. Meseguer. Distributed stable matching problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 152–166. Springer, 2005.

[38] V. Buravlev, R. De Nicola, and C. A. Mezzina. Tuple spaces implementations and their efficiency. In *International Conference on Coordination Languages and Models*, pages 51–66. Springer, 2016.

[39] C. J. Callsen. *Open Distributed Heterogeneous Computing.* PhD thesis, University of Aalborg, Institute for Electronic Systems, 1994.

[40] F. Cesarini and S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development.* O'Reilly Media, Inc., 2009.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM Sigplan Notices*, volume 40, pages 519–538, 2005.

[42] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[43] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[44] R. De Nicola, L. Di Stefano, and O. Inverso. Multi-agent systems with virtual stigmergy. In *Federation of International Conferences on Software Technologies: Applications and Foundations*, pages 351–366. Springer, 2018.

[45] R. De Nicola, T. Duong, O. Inverso, and F. Mazzanti. Verifying properties of systems relying on attribute-based communication. In *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma*, pages 169–190, 2017.

[46] R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang at Work. In *Proc. of the International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 485–497, 2017.

[47] R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: Empowering Erlang with attribute-based communication. In *International Conference of Coordination Models and Languages*, pages 21–39, 2017.

[48] R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: empowering Erlang with attribute-based communication. *Science of Computer Programming*, 2018.

[49] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A language-based approach to autonomic computing. In *Proc. of the International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 25–48, 2013.

[50] R. De Nicola, A. Lluch-Lafuente, M. Loreti, A. Morichetta, R. Pugliese, V. Senni, and F. Tiezzi. Programming and verifying component ensembles. In *Joint European Conferences on Theory and Practice of Software*, pages 69–83. Springer, 2014.

[51] 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 (TAAS)*, 9(2):7:1–7:29, 2014.

[52] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, Proceedings*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[53] R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[54] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[55] Á. F. Díaz, C. B. Earle, and L.-Å. Fredlund. eJason: an implementation of Jason in Erlang. In *Proc. of the International Workshop on Programming Multi-Agent Systems (PROMAS)*, pages 1–16, 2012.

[56] E. D'Osualdo, J. Kochems, and C. L. Ong. Automatic verification of erlang-style concurrency. In *Proc. of International Symposium on Static Analysis (SAS)*, pages 454–476, 2013.

[57] V. G. Engine. Naos server. http://www.naos-engine.com.

[58] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A logical verification methodology for service-oriented computing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2012.

[59] A. Ferscha. Collective adaptive systems. In *Proc. of the International Symposium on Wearable Computers (ISWC)*, pages 893–895, 2015.

[60] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 3–14, 2010.

[61] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[62] L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *ACM SIGPLAN Notices*, volume 42, pages 125–136, 2007.

[63] S. Frølund and G. Agha. A language framework for multi-object coordination. In *European Conference on Object-Oriented Programming*, pages 346–360. Springer, 1993.

[64] S. Frølund and G. Agha. Abstracting interactions based on message sets. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, pages 107–124, 1994.

[65] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[66] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[67] M. Gelain, M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh. Local search approaches in stable matching problems. *Algorithms*, 6(4):591–617, 2013.

[68] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[69] H. Geng. *interActors: A Model for Supporting Complex Communication in Concurrent Systems.* PhD thesis, University of Saskatchewan, 2017.

[70] H. Geng and N. Jamali. Supporting many-to-many communication. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 81–86. ACM, 2013.

[71] H. Geng and N. Jamali. interactors: A model for separating communication concerns of concurrent systems. In *Programming with Actors*, pages 186–215. Springer, 2018.

[72] I. P. Gent and P. Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, pages 141–145, 2002.

[73] S. Gnesi and F. Mazzanti. An abstract, on the fly framework for the verification of service-oriented systems. In *Rigorous software engineering for service-oriented systems*, pages 390–407. Springer, 2011.

[74] R. Goonatilake and R. A. Bachnak. Modeling latency in a network distribution. *Network and Communication Technologies*, 1(2):1, 2012.

[75] J. F. Groote and M. R. Mousavi. *Modeling and analysis of communicating systems*. The MIT Press, 2014.

[76] J. F. Groote and M. A. Reniers. *Algebraic process verification*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 2000.

[77] Q. Guo and J. Derrick. Formally based tool support for model checking erlang applications. *Journal on Software Tools for Technology Transfer*, 13(4):355–376, 2011.

[78] Q. Guo, J. Derrick, C. B. Earle, and L.-Å. Fredlund. Model-checking erlang–a comparison between etomcrl2 and mcerlang. In *Testing–Practice and Research Techniques*, pages 23–38. Springer, 2010.

[79] P. Harrenstein, D. Manlove, and M. Wooldridge. The joy of matching. *IEEE Intelligent Systems*, 28(2):81–85, 2013.

[80] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[81] M. H.ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, 2011.

[82] Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.

[83] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[84] Y. Isobe, Y. Sato, and K. Ohmaki. A calculus of countable broadcasting systems. In *International Conference on Algebraic Methodology and Software Technology*, pages 489–503. Springer, 1995.

[85] K. Iwama and S. Miyazaki. A survey of the stable marriage problem and its variants. In *Proc. of the International Conference on Informatics Education and Research for Knowledge-Circulating Society (ICKS)*, pages 131–136, 2008.

[86] K. Iwama, S. Miyazaki, Y. Morita, and D. Manlove. Stable marriage with incomplete lists and ties. In *Proc. of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 443–452. Springer, 1999.

[87] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[88] Z. Király. Linear time local approximation algorithm for maximum stable marriage. *Algorithms*, 6(3):471–484, 2013.

[89] A. Klarl. From helena ensemble specifications to promela verification models. In *Model Checking Software*, pages 39–45. Springer, 2015.

[90] D. Krzywicki, W. Turek, A. Byrski, and M. Kisiel-Dorohinicki. Massively concurrent agent-based evolutionary computing. *Journal of Computational Science*, 11:153–162, 2015.

[91] E. Letuchy. Facebook Chat. https://www.facebook.com/notes/facebook-engineering/facebook-chat/14218138919.

[92] J. Liebowitz and J. Simien. Computational efficiencies for multi-agents: a look at a multi-agent system for sailor assignment. *International Journal of Electronic Government*, 2(4):384–402, 2005.

[93] M. Loreti and J. Hillston. Modelling and analysis of collective adaptive systems with carma and its tools. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 83–119. Springer, 2016.

[94] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.

[95] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[96] D. F. Manlove, R. W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276(1-2):261–279, 2002.

[97] R. Mateescu and G. Salaün. Translating pi-calculus into lotos nt. In *International Conference on Integrated Formal Methods*, pages 229–244. Springer, 2010.

[98] F. Mazzanti. Designing UML models with UMC. Technical Report 2009-TR-043, Istituto di Scienza e Tecnologie dell'Informazione, CNR, 2009.

[99] F. Mazzanti, A. Ferrari, and G. O. Spagnolo. Towards formal methods diversity in railways: an experience report with seven frameworks. *International Journal on Software Tools for Technology Transfer*, 20(3):263–288, 2018.

[100] B. D. McKay and A. Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[101] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[102] D. Munera, D. Diaz, S. Abreu, F. Rossi, V. Saraswat, and P. Codognet. Solving hard stable matching problems via local search and cooperative parallelization. In *Proc. of the AAAI Conference on Artificial Intelligence*, 2015.

[103] R. H. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[104] A. O'Connell. Inside Erlang, The Rare Programming Language Behind WhatsApp's Success. http://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success.

[105] A. Pironti and R. Sisto. Provably correct java implementations of spi calculus security protocols specifications. *Computers & Security*, 29(3):302–314, 2010.

[106] H. Plociniczak and S. Eisenbach. JErlang: Erlang with joins. In *Proc. of the International Conference on Coordination Languages and Models (COORDINATION)*, pages 61–75, 2010.

[107] K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995.

[108] A. E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92(6):991–1016, 1984.

[109] A. E. Roth. *Who Gets What - and Why: The New Economics of Matchmaking and Market Design.* Houghton Mifflin Harcourt, 2015.

[110] A. E. Roth, T. Sönmez, and M. U. Ünver. Kidney exchange. *The Quarterly Journal of Economics*, 119(2):457–488, 2004.

[111] G. Salvaneschi, C. Ghezzi, and M. Pradella. ContextErlang: A language for distributed context-aware self-adaptive applications. *Science of Computer Programming*, 102:20–43, 2015.

[112] M. P. Singh. Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language. In *Proc. of International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 491–498, 2011.

[113] H. Song and K. J. Compton. Verifying π-calculus processes by promela translation. Technical Report CSE-TR-472-03, University of Michigan, 2003.

[114] S. Stalio, G. D. Carlo, S. Parlati, and P. Spinnato. Resource management on a VM based computer cluster for scientific computing. *CoRR*, abs/1212.4658, 2012.

[115] M. Sulzmann, E. S. Lam, and P. Van Weert. Actors with multi-headed message receive patterns. *Lecture Notes in Computer Science*, 5052:315–330, 2008.

[116] C.-P. Teo, J. Sethuraman, and W.-P. Tan. Gale-shapley stable marriage problem revisited: Strategic issues and applications. *Management Science*, 47(9):1252–1267, 2001.

[117] M. H. ter Beek, S. Gnesi, and F. Mazzanti. From EU projects to a family of model checkers. In *Software, Services, and Systems, volume 8950 of LNCS*, pages 312–328. Springer, 2015.

[118] M. Varshosaz and R. Khosravi. Modeling and verification of probabilistic actor systems using prebeca. In *International Conference on Formal Engineering Methods*, pages 135–150. Springer, 2012.