IMT School for Advanced Studies, Lucca

Lucca, Italy

A Foundational Theory for Attribute-based Communication

PhD Program in Computer Science (CDSS\CS)

XXIX Cycle

By

Yehia Abd Alrahman

2017

The dissertation of Yehia Abd Alrahman is approved.

Program Coordinator: Prof. Mirco Tribastone, IMT School for Advanced Studies, Lucca

Supervisor: Prof. Rocco De Nicola, IMT School for Advanced Studies, Lucca, Italy

Supervisor: Prof. Michele Loreti, Università degli Studi di Firenze, Florence, Italy

The dissertation of Yehia Abd Alrahman has been reviewed by:

Prof. Gul Agha, Department of Computer Science, University of Illinois at Urbana-Champaign, United States

Prof. Luca Aceto, School of Computer Science, Reykjavik University, Iceland

Prof. Francisco Martins, Department of Informatics, University of Lisbon, Portugal

IMT Institute for Advanced Studies, Lucca

2017

Only until you have climbed a mountain can you look behind and see the vast distance you have covered, and remember those you have met along the way who made your trek a little easier. Now that this thesis is finally finished, after many miles of weary travel, I look back to those who helped me turn it into reality and offer my heartfelt thanks.

First of all, I would like to express my sincere gratitude to my supervisor Prof. Rocco De Nicola for his continuous support of my Ph.D study and related research activities, for his patience and motivation. His guidance helped me in all the time of research and the writing of this thesis.

I would like to express my indebtedness and profound veneration to my supervisor Prof. Michele Loreti for all his efforts, his ultimate patience, availability, and immense knowledge. Above all he is a great friend whose continuous support has been critical to my growth.

I offer my heartfelt thanks to the reviewers: Prof. Gul Agha, Prof. Luca Aceto, and Prof. Francisco Martins for their time and precious efforts. Their comments and questions were very useful in preparing the final draft of this thesis.

Special thanks go to the members of the SysMA group at IMT School for fruitful discussions and suggestions, especially to Prof. Mirco Tribastone, Dr. Hugo Torres Vieira, and Dr. Claudio Antares Mezzina. I would like also to express my appreciation to the members of the LFCS laboratory, Edinburgh University, especially to Prof. Jane Hillston for hosting me as a visiting researcher in the final year of my Ph.D.

Words would never be able to fathom the depth of feelings for my reverend mother. She was always an indelible inspiration and affection during my research.

Yehia Abd Alrahman
Lucca, Italy
January, 2017

To my people in my homeland, Syria, who are struggling in their quest for freedom and might not have had the same opportunities that I had.

Contents

List of Figures

List of Tables

Declaration

Most of the material in this thesis has been published. In particular: Chapter 3, 4, and part of Chapter 5 are based on (ADL16a), coauthored with Rocco De Nicola, IMT School for Advanced Studies, Lucca, Michele Loreti, University of Florence. One adapted case study from Chapter 4 is based on (ADL$^+$15), coauthored with Rocco De Nicola, Michele Loreti, Francesco Tiezzi, University of Camerino, and Roberto Vigo, Technical university of Denmark. The scenario from Chapter 2 and the initial part of Chapter 6 is based on (ADL16b), coauthored with Rocco De Nicola and Michele Loreti. Finally, the $Ab^{a}C$uS Java run-time environment, presented in Chapter 6, has been developed in collaboration with Michele Loreti.

Vita

October 1, 1986	Born, Irbid, Jordan
2004-2009	Bachelor degree in Computer Engineering Philadelphia University, Jordan
2010-2012	Master degree in Computer Science Philadelphia University, Jordan
2013-2016	Ph.D In Computer Science IMT School for Advanced Studies Lucca, Italy
Jan-June 2016	Visiting Research Student LFCS Laboratory, School of Informatics The University of Edinburgh, UK
Sep-Dec 2016	Intern/Automated verification and approximation Max Planck Institute for Software Systems, Germany
Feb 2017-present	Research Collaborator IMT School for Advanced Studies Lucca, Italy

Publications

1. Y. Abd Alrahman et al, "Distributed Coordination of Multiparty Communication," International Conference, Coordination 2017, Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, in *Springer LNCS*, (Submitted).
2. Y. Abd Alrahman et al, "On the Power of Attribute-Based Communication," International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, in *Springer LNCS*, vol. 9688, pp.1–18, 2016.
3. Y. Abd Alrahman et al, "Programming of CAS systems by relying on attribute -based communication," 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, in *Springer LNCS*, vol. 9952, pp.539–553, 2016.
4. Y. Abd Alrahman et al, "A calculus for attribute-based communication," In Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC'15, Salamanca, Spain, ACM, pp. 1840 – 1845, 2015.
5. Y. Abd Alrahman et al, "Can We Efficiently Check Concurrent Programs Under Relaxed Memory Models in Maude?," Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, *Springer LNCS*, pp. 21 – 41, 2014.

Presentations

Conference talks:

1. Y. Abd Alrahman, "On the Power of Attribute-based Communication", at *DISCOTEC, FORTE*, Crete, Greece, 2016.
2. Y. Abd Alrahman, "A Calculus for Attribute-based Communication", at *SAC'15*, Salamanca, Spain, 2015.

Invited talks:

3. Y. Abd Alrahman, "An infrastructure for Attribute based Communication", at *QUANTICOL meeting*, Pisa, Italy, 2017.
4. Y. Abd Alrahman, "On the Expressiveness of Attribute-based Communication", at *PEPA CLUB*, Edinburgh, UK, 2016.
5. Y. Abd Alrahman, "On Expressiveness and Behavioural Theory of Attribute based Communication", at *QUANTICOL meeting*, Lucca, Italy, 2015.
6. Y. Abd Alrahman, "A Calculus for Attribute-based Communication", at *CINA meeting*, Turin, Italy, 2015.

Abstract

In open systems, i.e. systems operating in unpredictable environments and with components that may join or leave at any time, behaviors can arise as side effects of intensive components interaction. Finding ways to understand and design these systems and, most of all, to model the interactions of their components, is a difficult but important endeavor. To tackle these issues, we present AbC, a calculus for attribute-based communication. An AbC system consists of a set of parallel agents each of which is equipped with a set of attributes. Communication takes place in an implicit multicast fashion, and interactions among agents are dynamically established by taking into account "connections" as determined by predicates over the attributes of agents. First, the syntax and the semantics of the calculus are presented, then expressiveness and effectiveness of AbC are demonstrated both in terms of modeling scenarios featuring collaboration, reconfiguration, and adaptation and of the possibility of encoding channel-based interactions and other interaction patterns. Behavioral equivalences for AbC are introduced for establishing formal relationships between different descriptions of the same system. A Java run-time environment has been developed to support programming of the above mentioned class of systems by relying on the communication primitives of the AbC calculus. The impact of centralized and decentralized implementations for the underlying communication infrastructure, that mediates the interaction between components, has been studied.

Chapter 1

Introduction

> *But still, the emphasis ought to be on modeling what happens in real systems, whether they are human-made systems like operating systems, or whether they exist already. There is a subtle change from the Turing-like question of what are the fundamental, smallest sets of primitives that you can find to understand computation. I think some of that applies in concurrency, like naming: what is the smallest set of primitives for naming? So some of that applies. But as we move towards mobility, understanding systems that move about globally, you need to commit yourself to a richer set of semantic primitives. I think we are in a terrific tension between (a) finding a small set of primitives and (b) modeling the real world accurately.*

Robin Milner, *from an interview*, Cambridge, 2003

1.1 Motivations

The ever increasing complexity of modern software systems has changed the perspective of software designers who now have to consider a broad

range of new classes of systems, consisting of a large number of interacting components and featuring complex interaction mechanisms, e.g. *Software-Intensive Systems* (Bro06), *IoT Systems* (Kop11), and *Collective Adaptive Systems* (CAS) (Fer15). These systems are usually distributed, heterogeneous, decentralized and interdependent, and are operating more and more in dynamic and often unpredictable environments. There exist different kinds of complexity in the development of software systems. Historically, as software systems grew larger, the focus shifted from the complexity of developing algorithms to the complexity of structuring large systems, and to the inevitable complexities of building distributed and concurrent systems. We are facing another level of complexity arising from the fact that systems have to operate in large, open and non-deterministic environments where the complexity of interaction is increased.

Most of the current programming frameworks still handle the interaction between distributed components by relying on channel-based communication primitives. These include point-to-point (SW03), multicast with explicit addressing (i.e. IP multicast (HC99)), and broadcast communication (Pra91). To derive the interaction between components, these primitives rely on channel names or addresses that are totally independent of the run-time properties, status, and capabilities of the communicating components. This makes programming complex behaviors and interaction mechanisms, that are tightly coupled to components status such as reconfiguration, adaptation, and opportunistic interactions, quite difficult. In our view, it is important to consider more abstract interaction primitives and in this thesis we study the impact of a new paradigm that permits selecting groups of partners by considering their knowledge, status, and capabilities at run-time. The findings we report in this thesis have been triggered by our interest in CAS, and the recent attempts to define appropriate linguistic primitives to deal with such systems, see e.g. TOTA (MZ04), SCEL (DLPT14) and the calculi presented in (ADL$^+$15; VDB13).

CAS consist of large numbers of interacting components which combine their behaviors by forming collectives to achieve specific goals depending on their attributes, objectives, and functionalities. Decision-

making is complex and interaction between components may lead to unexpected behaviors. CAS are inherently scalable and the boundaries between CASs are very fluid in the sense that large number of components may enter or leave the collective at any time and may have different (potentially conflicting) objectives; so they need to dynamically adapt to their environmental conditions and contextual data. New engineering techniques to address the challenges of developing, integrating, and deploying such systems are needed (SCC$^+$12).

To move towards this goal, in our view, it is important to develop a theoretical foundation for this class of systems that would help in understanding their distinctive features. From the point of view of the language designers, the challenges are:

- to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems;
- to guarantee adaptation with respect to the working environment;
- to take into account evolving requirements and to control the emergent behaviors resulting from intensive and complicated interactions.

Obviously, handling the communication mechanisms of CAS can no longer depend on the current existing communication models that are totally independent from the characteristics of the communicating systems. These models do not allow scalability with the high level of dynamicity of such systems and it is a necessary to devise new techniques that change the perspective of how communication can be realized, possibly by taking into account run-time properties, status, and capabilities of systems. The cornerstone concepts of CAS have to guide the development of these communication techniques and their effectiveness has to be assessed by showing to what extent these concepts were guaranteed by such techniques.

The main concepts of CAS can be summarized according to their relevance as follows:

- Adaptivity: components should adapt their behaviors in response to the change in their contextual conditions and collected awareness

data. The interaction policies of components are tightly coupled to their run-time properties, status, and capabilities.

- Awareness: components should be aware of their run-time status, characteristics, capabilities, and should also have partial views of their surroundings.
- Collaboration: components can collaborate and combine their behaviors to achieve system-level goals in response to either changes in the environment or because of their predefined roles.
- Distribution, decentralization, and interdependence: components are distributed on a shared environment and execute independently without any centralized control. However, since these components are partially aware of their surroundings and share the same environment, any change of a component surroundings, possibly induced by other components, might influence its behavior, forming a sort of interdependence between components.
- Anonymity: components should be able to communicate and exchange data without knowing the existence of each other. The identity of a service provider is not relevant, only its characteristics and its ability to provide the service are important.
- Scalability: components can join or leave the system at anytime without disturbing the overall system behavior which means that components should not be tightly coupled. The synchronization dependencies between senders and receivers should be broken in the sense that a sender can always send without being aware of existing receivers. Also receivers should always receive based on the fraction of system that is able to provide the message or the service, not based on a specific sender.

1.2 Approach

In this thesis we focus on modeling the interaction of complex software systems, e.g., CAS, with the appropriate level of abstraction that permits

natural modeling and reasonable verification. These issues can be obviously tackled by resorting to the current existing interaction paradigms, introduced to handle communication in distributed systems. Some of the well-known approaches include: channel-based models (Mil80; Hoa78; MPW92), group-based models (AC93; CKV01; HC99), and publish/subscribe models (BN02; EFGK03). Below we briefly report the main features and limitations of such approaches.

Channel-based models rely on explicit naming for establishing communication links in the sense that communicating partners have to agree on channels or names to exchange messages. This implies that communicating partners are not anonymous to each other. Actually, communicating partners have static interfaces that determine the target of communication e.g., CCS (Mil80), CSP (Hoa78). This issue inspired Robin Milner to develop the π-calculus (MPW92) as a way to mitigate this problem by allowing names to be communicable, thus providing a greater flexibility by enabling dynamic interfaces. However even in π-calculus, the dynamicity of interfaces is limited in the sense that even if we allow generic (bound) input or output actions, these actions are disabled until they are instantiated with specific channel names. This means that a process is only willing to engage in communication when its actions are enabled. Furthermore, π-calculus and most process calculi rely on synchronous communication where a sender blocks until a receiver is available. Though this results in an elegant algebra, it creates synchronization dependencies between senders and receivers and affects the overall scalability of the system.

In *group-based* models, like the one used for IP multicast (HC99), the group is specified in advance in the sense that the reference address of the group is explicitly included in the message. On the other hand, groups or spaces in the ActorSpace model (AC93) are regarded as passive containers of processes (actors) and may be created and deleted with explicit constructs. Spaces may be nested or even overlap and can be created dynamically at run-time. Actually, the notion of space is a first class entity in the ActorSpace model. However, in real CAS systems the notion of a group or collective is quite abstract and is dynamically formed at the time of interaction by the available interested partners. Actually, com-

municating partners are unaware of the existence of each other and they receive messages based on mutual interests. The ActorSpace model relies on asynchronous primitives which are more suitable for distributed settings. The arrival order of messages is nondeterministic but each message is guaranteed to be eventually delivered which makes this model suitable for specific classes of applications. It is worth mentioning that coordination in the ActorSpace model is difficult because of the reliance only on asynchronous primitives.

In the *publish/subscribe* model, like the one used in NASA Goddard Mission Services Evolution Center (GMSEC)[1], each component can take the role of a publisher or a subscriber. Publishers produce messages and subscribers consume them. The subscribers are indirectly addressed by the contents of sent messages. That is, a subscriber expresses its interest independently from the publisher that produces the messages, and then it is asynchronously notified when a message that matches its interest arrives. The propagation of messages from publishers to subscribers is realized by introducing an exchange server that mediates the interaction. The exchange server stores the subscriptions of subscribers, receives messages from publishers, and forwards the messages to the correct subscribers. The result is that publishers and subscribers are unaware of the existence of each other. Though the anonymity of interaction is an effective solution to achieve scalability by allowing participants to enter or leave the system at anytime, the scalability problems move to the realization of the exchange server. In fact, since the exchange server is responsible for subscriptions and message filtering, it should be able to face a large number of participants with evolving subscriptions while maintaining an acceptable level of performance.

Clearly, a good candidate for modeling the interaction in CAS systems should aim at combining the advantages of the already existing communication paradigms while avoiding their inherent shortcomings. In this thesis we argue that *Attribute-based communication* is a promising paradigm for modeling the interaction in highly adaptive systems like CAS.

Attribute-based communication is a novel communication paradigm

[1] `http://opensource.gsfc.nasa.gov/projects/GMSEC_API_30/index.php`

that permits selecting groups of partners by considering the predicates over the attributes they expose. Thus communication takes place anonymously in an implicit multicast fashion without a prior agreement between the communicating partners. Because of the anonymity of the attribute-based interaction, scalability, dynamicity, and openness can be achieved at a high degree in distributed settings. The semantics of sending operations is non-blocking while input operations are blocking. This breaks synchronization dependencies between interacting partners, and communicating partners can enter or leave a group at any time without disturbing the overall system behavior.

Groups or collectives are dynamically formed at the time of interaction by means of available/interested receiving components that satisfy sender predicates. In this way run-time changes of attributes introduce opportunistic interactions between components. Indeed, interaction predicates can be parametrized with respect to local attribute values and when these values change, the interaction groups or collectives do implicitly change. This makes modeling adaptation quite natural.

Modeling opportunistic behavior in classical communication paradigms like channel-based communication, e.g., π-calculus (MPW92), is definitely more challenging. Components should agree on specific names or channels to interact. Channels have no connection with the component attributes, characteristics or knowledge. They are specified as addresses where the exchange should happen. These names/channels are static and changing them locally at run-time requires explicit communication and intensive use of scoping which affect program readability and compositionality.

The attribute-based system is however more than just the parallel composition of interacting partners; it is also parametrized with respect to the environment or space where system components are executed. The environment has a great impact on how components behave. It introduces a new way of indirect communication which we refer to as *Interdependence*, where components mutually influence each other unintentionally. For instance, in the ant foraging system (JR06), when an ant disposes pheromone in the shared space to keep track of her way back home, she influences other ants behavior as they are programmed to follow traces

of pheromone with higher concentration. In this way, the ant unintentionally influences the behavior of the other ants by only modifying the shared space. This type of indirect communication cannot be easily modeled even when relying on asynchronous communication (HT91) where messages are placed in specific temporary places with the intention that other addressed components will pick/receive them at some point of time.

To further support our approach, we would like to stress that attributes make it easy to encode interesting features of CAS. For instance, awareness can be easily modeled by locally reading the values of the attributes that represent either the component status (e.g., the battery level of a robot) or the external environment (e.g., the external humidity). Also localities of CAS components can be naturally modeled as attributes. In fact, the general concept of attribute-based communication can be exploited to provide a unifying framework to encompass different communication models and interaction patterns such as those outlined above and many others.

1.3 Contributions

The main contribution of this thesis is the introduction of *AbC* (ADL16a), a foundational calculus for modeling CAS systems by relying on *Attribute-based Communication*. The *AbC* calculus consists of a minimal set of primitives that permits attribute-based communication. *AbC* systems are represented as sets of parallel components, each equipped with a set of attributes whose values can be modified by internal actions. Communication actions (both send and receive) are decorated with predicates over attributes that partners have to satisfy to make the interaction possible. Thus, communication takes place in an implicit multicast fashion, and communication partners are selected by relying on predicates over the attributes in their interfaces. Unlike IP multicast (HC99) where the reference address of the group is explicitly included in the message, *AbC* components are unaware of the existence of each other and they receive messages only if they satisfy the sender's requirements. It should be noted that the messages should also satisfy the receiver's requirements, other-

wise messages are discarded.

The *AbC* calculus, presented in this thesis, is a refined and extended version of our early work presented in (ADL+15) which from now on we shall call "the old *AbC*". The latter is a very basic calculus with a number of limitations, see the discussion in Chapter 7. Here, we fully redesign the calculus, enrich it with behavioral equivalences and assess its expressiveness and effectiveness. The contributions of the thesis are:

- A process calculus, named *AbC*, with a minimal set of primitives tailored for modeling the interaction in highly dynamic and adaptive systems, like CAS, by relying on attribute-based communication. The set of primitives have been designed to serve this purpose. Actually, the brand new primitives model CAS concepts as first class citizens. For instance, adaptation, awareness, anonymity, and collective formation can be expressed in *AbC* in a natural and intuitive way. We present the formal syntax and semantics of the *AbC* calculus through a running example from the realm of collective-adaptive systems.

- The study of the expressive power of *AbC* both in terms of the ability of modeling scenarios featuring collaboration, reconfiguration, and adaptation and of the possibility of encoding channel-based communication and other communication paradigms. Indeed, we show that attribute-based communication can be used as a general unifying framework to accommodate different models and interaction patterns.

- The definition of behavioral equivalences for *AbC* by first introducing a context based reduction barbed congruence relation and then the corresponding extensional labelled bisimilarity. We also show how to use the equivalence relations to prove system properties.

- The proof of the existence of a uniform encoding from channel-based communication into *AbC* with the conjecture that the converse is not possible. To illustrate the correctness of the encoding, a process calculus, named $b\pi$-calculus (EM01), has been chosen as a repre-

sentative for channel-based process calculi and it has been encoded into AbC.

- $Ab\overset{a}{C}$uS, a Java run-time environment that allows programmers to use the linguistic primitives of the AbC calculus in Java programs, has been introduced. The implementation of $Ab\overset{a}{C}$uS fully relies on the formal semantics of the AbC calculus which enhances the confidence on the behavior of $Ab\overset{a}{C}$uS programs. We provide both centralized and decentralized implementations for the underlying communication infrastructure that mediates the interaction between $Ab\overset{a}{C}$uS components and we study their performance. We provide a scalable and relaxed[2] abstract machine for the AbC calculus and study its implications.

1.4 Structure of the Thesis

The reminder of this thesis is organized as follows:

In Chapter 2 the main features of the AbC calculus are presented informally in a step-by-step fashion using a case study from the realm of collective-adaptive systems. The idea is to motivate the need for the communication primitives of the AbC calculus and to show the impact of attribute-based communication paradigm.

Chapter 3 presents the formal syntax of the AbC calculus and provides evidence of the expressive power of AbC by showing how other communication models and interaction patterns can be easily rendered in AbC. In essence, a static translation from the $b\pi$-calculus syntax into AbC syntax, to illustrate the encoding of channel-based interaction, is presented. Also possible renderings of group-based and publish/subscribe interaction patterns are shown in terms of examples.

Chapter 4 presents the operational semantics for the AbC calculus and employs the semantics in different case studies from different application domains.

[2]We do not preserve the atomicity of message sending.

Chapter 5 presents a behavioral theory for the AbC calculus. Two behavioral relations are introduced: a reduction barbed congruence and its equivalent labeled bisimilarity. At the end of the chapter, bisimilarity is used to prove system properties and the correctness of the encoding, presented in Chapter 3, has also been proved.

Chapter 6 presents a Java run-time environment for supporting the communication primitives of the AbC calculus. Both centralized and decentralized implementations for the underlying communication infrastructure are studied. At the end of the chapter a scalable and relaxed abstract machine for the AbC calculus is discussed.

Chapter 7 and Chapter 8 discuss the related works and sketch some concluding remarks and future works. we also discuss the main shortcomings of the old AbC, published in (ADL$^+$15), and how they are handled in the new version of AbC that we discuss in this thesis.

Finally, in the Appendix, we provide a detailed proof of the completeness of the encoding, presented in Chapter 3, a full implementation of the running example, presented Chapter 2, in $Ab\overset{a}{C}$uS, and the formal definition of the old AbC calculus, published in (ADL$^+$15).

Chapter 2

AbC in a Nutshell

2.1 Introduction

This chapter presents the *AbC* calculus, its design choices, and its communication mechanisms. To help the reader appreciate *AbC* features we proceed with a running example from the realm of collective adaptive systems. The presentation in this chapter is intended to be intuitive and informal. Full details concerning the formal definition of the *AbC* calculus will be presented in the next chapters. Along the way we demonstrate how, through this running example, the *AbC* calculus covers a larger part of well-known collective-adaptive features than is obvious from the syntax. Notably, it covers the following:

- Anonymous interaction: *AbC* components are anonymous to each other and interact based on the satisfaction of predicates over their run-time attribute values.
- Awareness: *AbC* components have local views to their status and their environment. These views are used to collect awareness data at run-time to be used for decision making.
- Adaptation: The interaction predicates of *AbC* components are parametrized with respect to their local views and any run-time change in their views changes the possible set of interaction partners.

Another level of adaptation is achieved by modifying the shared environment. Since this environment is shared, any change to it might influence the behavior of other components. This ensures decentralized control while achieving some sort of interdependence between components. The interesting part about such kind of behavior is that it is not usually intentional. For example, as mentioned previously, when an ant disposes pheromone in the shared space, it does not directly instruct the other ants to follow her traces of pheromone, but actually increases the probability that other ants would adjust their behaviors.

- Scalability: This is achieved in *AbC* by means of different factors. First, the communication primitives in *AbC* support group-based communication rather than binary communication which scale well in large systems. Second, the non-blocking nature of sending operations in *AbC* breaks the synchronization dependencies between senders and receivers. The sender does not wait for a receiver and actually is not aware of the existence of receivers. Finally, the anonymous nature of the interaction requires no agreement between interacting partners and this can offer a high degree of scalability in distributed settings.

- Collaboration: This is achieved as a form of adaptation at the system level where components decide to combine their behavior based on changes in the surrounding environment of one or more components.

We proceed with a high-level specification language syntax as reported in Figure 1, rather than with the formal syntax of the *AbC* calculus, to explain the main features of attribute-based communication in an intuitive way, thus avoiding excessive verbosity in the running example. This syntax has a one-to-one correspondence with the formal syntax of the *AbC* calculus. Actually it represents a generic template for defining an *AbC* component which is the basic building block of *AbC* specifications. As shown in Figure 1 each component has a name and a set of optional arguments used to initialize its state. The body of a component consists

```
1   Component name ( type_1 arg_1,..., type_n arg_n) =
2       Env { ···
3           attr_kind aname_i := expression_i; ···
4       }
5       behavior { ···
6          Process P_i =
7              inist_stmt
8          EndProcess
9           ···
10      }
11      init { P_1|···|P_i }
12  EndComponent
```

Figure 1: A high-level specification of an *AbC* component.

of three main blocks: its environment block, `Env`, which consists of a set of attributes identifiers, possibly with assigned initial values; its behavior block, `behavior`, which consists of a set of process definitions; and its initialization block, `init`, which specifies its initial behavior, possibly a parallel composition of a set of co-located processes $P_1|\cdots|P_i$. Each process is defined with a unique process identifier P_i and a set of initial statements/commands *inist_stmt*.

In this section we focus on the role of the communication primitives and the external environment in deriving the communication between interacting components. We motivate the need for the communication primitives of the *AbC* calculus and show why attribute-based communication is a good candidate for handling the interactions in collective-adaptive systems.

2.2 A Smart Conference System

We proceed with a smart conference system as a running example which will be used to illustrate the main concepts of attribute-based communication. The idea is to exploit the mobile devices of the conference participants to guide them to their locations of interest. Each participant expresses his topic of interest and the conference venue is responsible for guiding each participant into the location that matches his interest. The

conference venue is composed of a set of rooms where the conference sessions are to be held. We assume that the name of each room identifies its location, e.g., "1^{st} Floor, Room.101" and each participant has a unique *id*. The conference program and session relocation can be dynamically adjusted at anytime to handle specific situations, i.e., a crowded session can be moved into a larger room and this should be done seamlessly without any disruption to the whole conference program. When relocation happens, the new updates should be communicated to the interested participants.

Each participant is represented as an *AbC* component, Figure 2, and the conference venue is represented as a set of parallel *AbC* components, each of which representing a room, Figure 3. The overall system is represented as the parallel composition of the conference venue and the set of available participants. As mentioned previously, each *AbC* component consists of a set of attributes, which we call *attribute environment*, that represents its status, capabilities and also a partial view of its surrounding environment and a process which takes these attribute values as parameters and represents the component behavior. The attribute environments for participants and rooms contain the following attributes according to the specifications in Figure 2 and Figure 3 respectively:

- `id`: Identifies the identity of a participant.
- `interest`: Identifies the current participant topic of interest.
- `destination`: Identifies the location of the room where the topic of interest for a participant is to be held.
- `name`: Identifies the location of a room, e.g., "1^{st} Floor, Room.101".
- `role`: Identifies the role of a component.
- `session`: Identifies the current scheduled session for a room.
- `previousSession`: Identifies the previous session that was supposed to be held in a room if any.

```
1   Component ParticipantComponent ( String id, String topic) =
2       Env {
3           attrib id := id;
4           attrib interest := nil;
5           attrib destination := nil;
6       }
7       behavior {
8           Process ParticipantAgent = ···
9       }
10      init { ParticipantAgent }
11  EndComponent
```

Figure 2: The *AbC* specifications for a participant component

- `newSession`: Identifies an environmental attribute (i.e., its value is provided by the environment), that associates a room with a new session.

- `relocate`: Identifies a boolean environmental attribute, when enabled it instructs the room to start the relocation process.

2.2.1 The participant component behavior

The behavior of the participant component is represented by the process `ParticipantAgent` as reported in Figure 2, Line 10. When a participant arrives to the conference venue, a participant component is created and associated with a `ParticipantAgent` process where the topic of interest is selected and is passed to the process `ParticipantAgent` through the component argument `topic` (Line 1).

The process `ParticipantAgent` is reported in Specification 2.1 below. The participant starts executing by updating the value of his attribute *interest* with the selected topic as shown at line 2. The command `setValue (interest, topic)` is used to update the value of the attribute interest with the initial topic of interest. This will allow the participant to communicate his topic of interest to the conference venue by sending a session request to nearby providers; in our case, this is a room. The send operation, at line 3, consists of mainly two parts: the predicate

```
1   Component RoomComponent ( String name, String role, String session)
      =
2     Env {
3       attrib name := name;
4       attrib role := role;
5       attrib session := session;
6       attrib previousSession := nil;
7       attrib newSession := nil;
8       attrib relocate := False;
9     }
10    behavior {
11      Process Service = ···
12      Process Relocation = ···
13      Process Updating = ···
14    }
15    init { Service | Relocation | Updating}
16  EndComponent
```

Figure 3: The *AbC* specifications for a room component

"`role==PROVIDER`" which specifies the targeted group of the message to be the set of nearby components with a provider role; and the tuple of the communicated values "⟨`this.interest, REQUEST, this.id`⟩". The communicated values include the current topic of interest of the participant, a label `REQUEST`, and the identity of the participant respectively. Once the message is emitted, the process blocks its execution until a reply notification that matches his interest arrives. The notification contains the session name, a `REPLY` label, and the name of the room where the session is to be held. The participant is eligible to receive the message only if its attribute environment satisfies the attached message predicate. If the participant is eligible to receive the message, the receive method, Line 4, passes the tuple of communicated values "o" to the following predicate:

$$\texttt{(o.get(0)==this.interest} \wedge \texttt{o.get(1)==REPLY)}$$

The predicate checks if the received values satisfy the receiving constraints. In principle it returns true if the first element of the tuple matches the participant topic of interest and the second element is a

REPLY label, otherwise it returns false. By receiving this notification, the process updates its destination and waits for new possible updates about his topic of interest. It should be noted that the send operation is non-blocking while the receive one blocks until the desired message arrives. In anyway, synchronization is still required (i.e., messages are not buffered) if receivers are available and that is why sometimes we need to spawn a new AbC process to make sure that messages are not lost.

Specification 2.1: The process `ParticipantAgent`

```
1   Process ParticipantAgent =
2     setValue(interest, topic);
3     send(role==Provider , ⟨this.interest , REQUEST ,this.id⟩);
4     var value = receive( o -> ( o.get(0)==this.interest ∧
          o.get(1)==REPLY ) );
5     setValue(destination, value.get(2));
6     repeat {
7       var value = receive( o -> ( o.get(1)==this.interest ∧
            o.get(2)==UPDATE ));
8       setValue(destination, value.get(3));
9     }
10  EndProcess
```

The code in lines 6-9 ensures that the participant is always ready to receive new updates about the topic of his interest. Precisely, it blocks until it receives an update notification about a session that matches the participant interest. The notification message contains the previous session that was supposed to be held in this room, the current session, an `UPDATE` label, and the name of the room where the session of interest has been moved. The predicate "`( o.get(1)==this.interest ∧ o.get(2)==UPDATE )`", at line 7, returns true if the attached session matches the participant topic of interest and the message is labeled with "`UPDATE`", otherwise it returns false. Once a notification message is received, the process updates the destination to the new location and waits for future updates. The `repeat{}` structure is equivalent to an infinite loop that keeps executing the code in its body. As we will see in the next chapter this structure is equivalent to a recursive call in process calculi.

Discussion

We consider the collective adaptive features that were evident in the previous specifications. These features include anonymity, scalability, and some sort of adaptivity. The role of the attribute environment was not clear in this part of the running example and we will discuss it alongside with its features in the next two sections.

Anonymity was guaranteed by allowing the interaction primitives (both send and receive) to rely on predicates over the run-time attribute values of the interacting partners rather than on specific names or addresses to derive the interaction. These attributes might represent the run-time status, capabilities and knowledge of the interacting partners. This means that the question of being qualified to receive a message no longer depends on what channel or address you are listening to, but rather if your run-time attribute values satisfy the sender requirements. For instance, the `ParticipantAgent` process sends a session request to the group of components that currently serve as providers and expects to receive an interest reply from a component which holds a session that matches his topic of interest. The sending operation is non-blocking and the participant is unaware if his message has been received or not. There is no prior agreement between the participant and the conference venue. Actually, the set of possible receivers is specified at the time of interaction in the sense that any change in the session schedule will change the possible set of targeted components. Any number of new participants can arrive to the conference venue without any disruption to the overall system behavior, under the assumption that the total number of participants cannot exceed the capacity of the overall conference venue.

The interaction primitives adopt multiparty rather than binary communication which scale well in large systems. Actually AbC supports an implicit multicast communication in the sense that the multicast group is not specified in advance, but rather is specified at the time of interaction by means of the available set of receivers with attributes satisfying the sender predicate. The non-blocking nature of the AbC multicast alongside with anonymity of the interaction break the synchronization dependencies

between senders and receivers and make it more suitable for supporting applications that require a higher degree of scalability.

Some sort of adaptation was evident in the `ParticipantAgent` process. For instance, the code in lines 6-9, the participant changes his destination based on interest updates coming from the conference venue. This is an obvious kind of adaptation which relies solely on explicit communication messages from other partners. More interesting kinds of adaptation will be presented in the next two sections.

2.2.2 The room component behavior

The behavior of a room component in the conference venue is presented by the parallel composition of three different processes, namely `Service`, `Relocation`, and `Updating` as reported in Figure 3, Line 15.

The `Service` process is responsible for providing a normal service for a room. When a room receives a session request, the process `Service` replies back to the requester with the room location in case that the current session of the room matches the requester topic of interest. The `Relocation` and the `Updating` processes are responsible for handling sessions and participants relocation from one room to another. These co-located processes cannot communicate directly, but can influence the behavior of each other through the shared attribute environment by modifying the values of some attributes at run-time. Since the interaction predicates of *AbC* processes are parametrized with respect to the shared attribute values, any run-time change to these values changes the set of possible interaction partners and introduces opportunistic interaction between components. This makes modeling adaptivity in *AbC* specifications quite natural.

The process `Service`, reported in Specification 2.2, shows how the room normally handles participant requests. The `Service` process is busy-waiting for session requests from the available participants. Once a message with a possible session request is received, it is passed to the predicate `( o.get(0)==this.session` $\wedge$ `o.get(1)==REQUEST )` (Line 3) to check if its first value matches the current room session and its second value is a "REQUEST" label. If so, a new *AbC* process is created to send an interest reply message to the requester and again the process `Service` is made

available to handle session requests. The operation `exec`, Line 4, is used to create a new process and execute it in parallel with the main process. In principle process `Service` replicates itself every time a session request is received to ensure that it is always ready to handle concurrent requests from different participants. This kind of replication is safe in the sense that we create a new process only when a session request is received and this process terminates soon after. The send operation at line 5 replies to the requester with an interest reply addressing the requester with its identity `id==value.get(2)`. The interest reply message contains the current session of the room, a `REPLY` label, and the name of the room. Note that, since the predicate of the interest reply message addresses the requester with its identity, there is only one possible receiver for this message which is the requester himself.

Specification 2.2: The service process

```
1     Process Service =
2      repeat {
3        var value = receive( o -> ( o.get(0)==this.session ∧
              o.get(1)==REQUEST ) );
4        exec( new Process =
5                  send(id==value.get(2), ⟨ this.session , REPLY
                       ,this.name⟩);
6                EndProcess
7        );
8      }
9     EndProcess
```

The `ParticipantAgent` process in the previous section appends the participant identity to the session request message and the `Service` process sends an interest reply only to the requesters addressing them with their identities in the interaction predicate as mentioned before. In this way a one-to-one communication between a room and a participant is guaranteed. The size of the collective is reduced to include only two partners and if for some reason the message is lost, the participant ends up in a deadlock state waiting for an interest reply message that will never arrive. The implementation of the processes `ParticipantAgent` and `Service` can be enhanced by relying only on the attributes `session` and `interest` for deriving the communication. This means that the `ParticipantAgent`

process no longer needs to append its identity in the session requests and the process `Service` will send interest replies to participants with interests that match the room's current session. The superiority of attribute-based communication arises in the sense that as the collective size increases, the probability that a participant ends up in a deadlock state decreases. The idea is that if more than one participant is interested in the same topic, only one successful reception of a session request from one of them is sufficient so that all of them receive an interest reply from a component with a session that matches their topic of interest.

The process `Relocation` reported in Specification 2.3 is responsible for handling unexpected changes of the schedule for a room. The idea is to handle these run-time changes in a way such that interested participants in the new session and also other rooms where a swap of schedule should happen are notified. The behavior of this process is triggered by environmental changes. The environmental attributes `NewSession` and `Relocate` play a crucial role in controlling the behavior of the `Relocation` process. The values of these attributes are provided by the environment or by other components working in the same environment. These components might be humans or sensors that intervene to adapt the system behavior in a way that keeps it functioning properly.

A possible scenario is that the session is becoming too crowded and needs to be relocated to another larger room with possibly few attendees. Another component that plays a portal role and keeps information about the capacity and the run-time utilization of all rooms can propose a new session that best fits with the capacity of this room. Note that the portal component only proposes suitable sessions for rooms based on their capacities and their run-time utilization, but has no control on their behaviors. The decision of relocation is made by the room itself depending on the readings from sensors which set the value of the attribute `relocate` to true if the level of overcrowding exceeded a certain threshold. To achieve relocation we have to steer the crowd from one room to another and vice versa. Actually, three different parties have to be notified about the changes of the schedule. The participants who are interested in the new proposed session, the room that is currently assigned the new proposed session and

should swap its session with the crowded room, and the participants who are interested in the crowded session.

The `Relocation` process is only responsible for notifying the room where a swap of schedule should happen and also the participants who are interested in the new proposed session. The `Updating` process in Specification 2.4 will take part in notifying the participants who are interested in the crowded session about the new location of their topic of interest. An awareness construct is needed to enable the `Relocation` process to keep track of environmental changes. The awareness construct is to be used to collect run-time awareness data from the attribute environment of the component where the process `Relocation` is executed and based on these data decisions can be made. The construct appears at line 3, Specification 2.3 and is called `waitUntil()`. It takes a predicate as input argument and blocks the execution of the process `Relocation` until the predicate evaluates to true.

Specification 2.3: The relocation process

```
1   Process Relocation =
2    repeat {
3      waitUntil(this.relocate==true);
4      setValue(previousSession, this.session);
5      setValue(session, this.newSession);
6      setValue(relocate , false);
7      send((interest==this.session) ∨ (session==this.session)
8       , ⟨this.previousSession, this.session, UPDATE, this.name⟩);
9    }
10  EndProcess
```

The process `Relocation`, in Specification 2.3, blocks its execution until the value of the attribute `relocate` becomes true, indicating that the overcrowding of the current session exceeded a certain threshold and a relocation is needed. The `Relocation` process first updates the room's previous session to the current one and the current session to the new session provided by the portal component through the value of the attribute `newSession`. The relocation flag *relocate* is turned off by setting its value to false (Lines 4-6). The process continues by sending a session/interest update to the participants who are interested in the new assigned session and also to the room where a swap of schedule should happen as shown

in the "∨" predicate at line 7. The interest update message contains the name of the previous session, the name of the new session, an `UPDATE` label, and the name of the room (Line 8).

To relocate the sessions and steer the crowds correctly both of the involved rooms should collaborate and propagate the changes in their schedules to the interested participants. The global goal is to make both groups of participants, interested in either one of the two rooms sessions, aware of the new location of their topics of interest. As we have seen before the process `Relocation` was partly responsible for propagating the changes, through a session update message, to the group of participants who are interested in the new assigned session for the room which was crowded. The same message was used to ask the other room, where a swap of schedule should happen, to collaborate. Now, it is the responsibility of the other room to update its session and propagate these changes to the other group of participants who are interested in the crowded session. Actually, this is the role of the process `Updating`, reported in Specification 2.4.

Specification 2.4: The Updating process

```
1   Process Updating =
2    repeat {
3      var value = receive( o -> ( o.get(1)== this.session ∧
           o.get(2)==UPDATE ) );
4      setValue( previousSession, this.session);
5      setValue( session, value.get(0));
6      exec( new Process =
7                send((interest==this.session)∨
8                     (session==this.session), ⟨
                          this.previousSession , this.session,
                          UPDATE ,this.name⟩));
9             EndProcess
10     );
11   }
12  EndProcess
```

When a room receives a session update message, it passes the message to the predicate `( o.get(1)==this.session ∧ o.get(2)==UPDATE )`which checks if the message is relevant for the current room as shown at line 3, Specification 2.4. If this is the case, the attribute `previousSession` takes the value of the attribute `session` and the attribute `session` takes the

value of the communicated previous session name from the other room (Lines 4-5). Once the changes have been applied, a new *AbC* process is created to send a session/interest update message and again the process `Updating` is made available to handle session update messages. This is important to ensure that a room is always ready to handle concurrent session updates. The new created process sends a session/update message to the other group of participants so that they relocate to the new destination. It should be noted that the structure of this message is exactly the same of the one sent by process `Relocation`. The only difference is that the sent values depend on the current attribute values of the room where the process `Updating` is executed.

Discussion

We consider other collective adaptive features that were evident in the previous specification excerpts. These features include awareness, adaptation and interdependence, and also collaboration to achieve specific goals. These features were possible because of their dependencies on the attribute environment.

The attribute environments play a crucial role in orchestrating the behavior of *AbC* components. They make the components aware of their own status and also provide partial views of the surrounding environment. Components behave differently under different environmental contexts. This is possible because the behavior of *AbC* processes is parametric with respect to the run-time attribute values of the component in which they are executed. For instance, the normal behavior of a room is represented by the process `Service`. If the room is made aware, through its sensors, that the present number of participants exceeds its capacity, the room triggers the execution of the process `Relocation` to cope with the new situation. The room has also a partial view of its own environment, in our case this is the portal component. As mentioned previously, at run-time the portal component proposes to each room an alternative session that best fits with its capacity through the value of the attribute "`newSession`" and the room considers this proposal when relocation is needed. The constructs "`waitUntil()`" and "`this.a`" are used as environmental pa-

rameters that influence the behavior of AbC processes at run-time. For instance, the behavior of the process `Relocation`, in Specification 2.3, is controlled by the guard "`waitUnil(this.relocate== true)`" at line 3, which blocks its execution until the value of the attribute `relocate` becomes true. In the update operation `setValue()` at line 4, the dependency of the attribute "`previousSession`" on the value of attribute "`session`" is made possible through the reference `this.` which returns the current value of the attribute after the dot.

Components might adapt their behavior by either considering collected awareness data or receiving adaptation requests from other components using explicit communication as we have seen in the process `ParticipantAgent` before. Adaptation can come in different ways either by triggering the execution of a process in response to changes in the environment as the case with the process `Relocation`, or by explicitly changing the values of local attributes which change the possible set of interacting partners as shown in the code of the process `Relocation` after line 6. When the value of the attribute `session` is changed using the method "`setValue()`", each sending or receiving predicate that depends on the value of this attribute will consider another possible set of interacting partners. This way interdependence between co-located processes arises. For instance the process `Service` will now consider session requests from participants with interests that match the new session name.

Interdependence between components arises either directly or indirectly. For instance, when the sensors detect that the present number of participants in the current session exceeds the room capacity, the value of attribute `relocate` is set to true and this will trigger the relocation process. The participants do not communicate their presence directly, but interdependence arises from the fact that these components share the same working space. Interdependence can also arise directly when one or more components are responsible for providing data to other components. In our example the portal component proposes new sessions for rooms based on their capacity and their run-time utilization.

One interesting feature that is evident in the behavior of the smart conference system is collaboration which is crucial when a certain behavior

should be achieved at the system level. One component alone cannot achieve such kind of behavior. The idea is to combine local component behaviors through message exchange to achieve a global goal. In our example the goal was to relocate sessions and steer the crowd from one session to another in a way that each participant is notified about the new location of his session. That was achieved by allowing two different components one executing process `Relocation` and the other executing process `Updating`. As we have seen before each of the two rooms was responsible for updating its behavior and steering part of the crowd. The overall combined behavior of these rooms and the targeted participants allowed this goal to be achieved.

It is worth mentioning that the initial behavior of all components with the same type (i.e., room components or participant components) is exactly the same. However, since this behavior is parametrized with respect to the run-time attribute values of each component, components might behave differently as we have seen when collaboration happened between components to steer the crowd. This means that the context where a component is executing has a great influence on its behavior and it either disables or enables specific behaviors based on the run-time requirements. In some sense, the behavior of components evolves based on contextual conditions. Components do not need to have complex behavior to achieve adaptation at the system level. Complex and emergent behavior can be achieved by combining the local behavior of individual components to achieve system level goals.

Chapter 3

The AbC Calculus and its Expressive Power

In this chapter we formally present the syntax of the AbC calculus and show its expressive power by means of encoding other communication paradigms into it. For the sake of simplicity we explain the syntax in a step by step fashion using the smart conference system from the previous chapter.

3.1 Syntax of the AbC Calculus

The syntax of the AbC calculus is reported in Table 1. The top-level entities of the calculus are *components* (C); a component is either a process P associated with an *attribute environment* Γ, denoted $\Gamma : P$, or the parallel composition $C_1 \| C_2$ of two components, or the replicating component $!C$ which can always create a new copy of C. An *attribute environment* $\Gamma : \mathcal{A} \twoheadrightarrow \mathcal{V}$ is a partial map from attribute identifiers $a \in \mathcal{A}$ to values $v \in \mathcal{V}$ where $\mathcal{A} \cap \mathcal{V} = \emptyset$. A value could be a number, a name (string), a tuple, etc. The scope of names say $\tilde{n}$, can be restricted by using the restriction operator $\nu\tilde{n}$. For instance, in a component of the form $C = C_1 \parallel \nu\tilde{n}C_2$, the occurrences of the names $\tilde{n}$ in C_2 are only visible within C_2. The visibility of attribute values can be restricted while the visibility of attribute iden-

(Components)	$C ::= \Gamma : P \quad \mid \quad C_1 \| C_2 \quad \mid \quad !C \quad \mid \quad \nu \tilde{x} C$	
(Processes)	$P ::=$	
(Inaction)		$\mathsf{0}$
(Input)	$\mid$	$\Pi(\tilde{x}).P$
(Output)	$\mid$	$(\tilde{E})@\Pi.P$
(Update)	$\mid$	$[\tilde{a} := \tilde{E}]P$
(Awareness)	$\mid$	$\langle \Pi \rangle P$
(Non-determinism)	$\mid$	$P_1 + P_2$
(Interleaving)	$\mid$	$P_1 \mid P_2$
(Call)	$\mid$	K
(Predicates)	$\Pi ::= \mathsf{tt} \quad \mid \quad \mathsf{ff} \quad \mid \quad E_1 \bowtie E_2 \quad \mid \quad \Pi_1 \wedge \Pi_2$	
	$\mid \quad \Pi_1 \vee \Pi_2 \quad \mid \quad \neg \Pi$	
(Expressions)	$E ::= v \quad \mid \quad x \quad \mid \quad a \quad \mid \quad \texttt{this}.a$	

Table 1: The syntax of the AbC calculus

tifiers is instead never limited. The attribute identifiers represent domain concepts and it is assumed that each component in a system is always aware of them[1]

Example 3.1 (step 1/6)**.** *The smart conference system in Chapter 2, Section 2.2 is represented as the parallel composition of the conference venue and the set of available participants. The conference venue is represented as a set of parallel* AbC *components, each of them representing a room* $(Room_1 \| \ldots \| Room_n)$ *and each room has the following form* $\Gamma_i : R$ *where* Γ_i *represents the attribute environment of the room and* R *represent its behavior. The Participant instead has the following form* $\Gamma_p : P$ *where* Γ_p *represents its attribute environment and* P *represents its behavior. The overall system is defined below:*

$$Room_1 \| \ldots \| Room_n \parallel Participant_1 \parallel \ldots \parallel Participant_m$$

□

[1] In the rest of this thesis, we shall however occasionally use the term "attribute" instead of "attribute identifier".

$\Gamma \models \mathsf{tt}$	for all Γ
$\Gamma \models \mathsf{ff}$	for no Γ
$\Gamma \models E_1 \bowtie E_2$	iff $\Gamma(E_1) \bowtie \Gamma(E_2)$
s.t. $\Gamma(v) = v \ \wedge \ \Gamma(\texttt{this}.a) = \texttt{this}.a$	
$\Gamma \models \Pi_1 \wedge \Pi_2$	iff $\Gamma \models \Pi_1$ and $\Gamma \models \Pi_2$
$\Gamma \models \Pi_1 \vee \Pi_2$	iff $\Gamma \models \Pi_1$ or $\Gamma \models \Pi_2$
$\Gamma \models \neg\Pi$	iff not $\Gamma \models \Pi$

Table 2: The predicate satisfaction

A *process* is either the *inactive* process $\mathsf{0}$, or a process modeling *action-prefixing* $\bullet.P$ (where "$\bullet$" is replaced with an action), *attribute update* $[\tilde{a} := \tilde{E}]P$, *context awareness* $\langle\Pi\rangle P$, *nodeterministic choice* between two processes $P_1 + P_2$, *parallel composition* of two processes $P_1|P_2$, or *recursive behaviour* K (it is assumed that each process has a unique process definition $K \triangleq P$).

The attribute update construct in $[\tilde{a} := \tilde{E}]P$ sets the value of each attribute in the sequence $\tilde{a}$ to the evaluation of the corresponding expression in the sequence $\tilde{E}$ with respect to the attribute environment where the update process is executing. The term $\tilde{a}$ is the sequence of pairwise different attribute identifiers. The awareness construct in $\langle\Pi\rangle P$ is used to test awareness data about a component status or its environment. This construct blocks the execution of process P until the predicate Π becomes true. The parallel operator "|" models the interleaving between co-located (i.e., residing within the same component). In what follows, we shall use the notation $[\![\Pi]\!]_\Gamma$ (resp. $[\![E]\!]_\Gamma$) to indicate the evaluation of a predicate Π (resp. an expression E) under the attribute environment Γ. Notice that the expression $[\![\texttt{this}.a]\!]_\Gamma$ denotes the value of the attribute a under Γ (i.e.,$[\![\texttt{this}.a]\!]_\Gamma = \Gamma(a)$). In essence, the evaluation operator $[\![\bullet]\!]_\Gamma$ is used to concretize the predicates or the expressions by replacing the occurrences of the expression $\texttt{this}.a$ with their concrete values under Γ at the time of evaluation. This means that the evaluation of a predicate returns a concrete predicate while the evaluation of an expression returns a concrete value.

Example 3.2 (step 2/6)**.** *The structures of process P, specifying the behavior of a participant, and the process R, specifying the behavior of a room, are defined as follows:*

$$P \triangleq [\mathtt{this}.interest := initialTopic]\ P'$$

$$R \triangleq Service \quad | \quad Relocation \quad | \quad Updating$$

When a participant arrives to the conference venue, he selects his topic of interest and behaves as P'. On the other hand, the room behavior is defined as the the parallel composition of three different processes Service, Relocation, and Updating. It should be noted that these process cannot communicate directly and they can influence the behavior of each other by only updating the attribute environment. □

In AbC there are two kinds of *communication actions*:

- the attribute-based input $\Pi(\tilde{x})$ which binds to sequence $\tilde{x}$ the corresponding values received from components whose communicated messages and/or attributes satisfy the predicate Π;
- the attribute-based output $(\tilde{E})@\Pi$ which evaluates the sequence of expressions $\tilde{E}$ under the attribute environment and then sends the result to the components whose attributes satisfy the predicate Π.

A *predicate* Π is either a binary operator $\bowtie$[2] between two values or a propositional combination of predicates. Predicate tt is satisfied by all components and is used when modeling broadcast while ff is not satisfied by any component and is used when modeling silent moves. The satisfaction relation $\models$ of predicates is presented in Table 2. In the rest of this thesis, we shall use the relation $\doteq$ to denote a semantic equivalence for predicates as defined below.

Definition 1 (Predicate Equivalence)**.** *Two predicates are semantically equivalent, written $\Pi_1 \doteq \Pi_2$, iff for every environment Γ, it holds that:*

$$\Gamma \models \Pi_1 \text{ iff } \Gamma \models \Pi_2$$

[2]Note that $\bowtie$ ranges over basic binary operations like $>$, $<$, $\leq$, $\geq$, $=$, etc. In other words, every predicate of the form $E_1 \bowtie E_2$ should be decidable.

Clearly, the predicate equivalence, defined above, is decidable because we limit the expressive power of predicates by considering only standard boolean expressions and simple constraints on attribute values as shown in Table 2.

An *expression* E is either a constant value $v \in \mathcal{V}$, or a variable x, or an attribute identifier a, or a reference to a local attribute value `this`.a. The properties of *self-awareness* and *context-awareness* are guaranteed in AbC by referring to the values of local attributes via a special name `this`. (i.e., `this`.a). These values represent either the current status of a component (i.e., *self-awareness*) or the external environment as perceived by the component (i.e., *context-awareness*). Expressions within predicates contain also variable names, so predicates can check whether the values that are sent to a specific component do satisfy specific conditions. This permits a sort of pattern-matching. For instance, component $\Gamma : (x > 2)(x, y).P$ receives a sequence of values "x, y" from another component only if the value x is greater than 2.

We assume that our processes are *closed* (i.e., without free process variables), and that free names can be used whenever needed. The constructs νx and $\Pi(\tilde{x})$ act as binders for names (i.e., in $\nu x C$ and $\Pi(\tilde{x}).P$, x and $\tilde{x}$ are bound in C and P, respectively). We use the notation $bn(P)$ to denote the set of bound names of P. The free names of P are those that do not occur in the scope of any binder and are denoted by $fn(P)$. The set of names of P is denoted by $n(P)$. The notions of bound and free names are applied in the same way to components, but free names also include all attribute values that do not occur in the scope of any binder.

Example 3.3 (step 3/6)**.** *In the previous step, if we further specify the process P' in P, the processes Service, Relocation, and Updating in R, the behavior of a participant and a room becomes:*

$$P \triangleq [\mathit{this}.interest := initialTopic]$$
$$(\mathit{this}.interest,\ REQUEST,\ \mathit{this}.id)@(role = Provider).\ P''$$

$$Service \triangleq (x = \mathit{this}.session \wedge y = REQUEST)(x,\ y,\ z).\ S'$$

$$\begin{aligned}
&Relocation \triangleq \\
&\langle \mathit{this}.relocate = \mathsf{tt} \rangle [\mathit{this}.prevSession := \mathit{this}.session, \\
&\quad \mathit{this}.session := \mathit{this}.newSession,\ \mathit{this}.relocate := \mathsf{ff}] \\
&\quad (\mathit{this}.prevSession,\ \mathit{this}.session,\ UPDATE,\ \mathit{this}.name) \\
&\qquad @(interest = \mathit{this}.session \vee session = \mathit{this}.session).Relocation
\end{aligned}$$

$$Updating \triangleq (y = \mathit{this}.session \wedge z = UPDATE)(x, y, z, l).\ U'$$

The correspondence with respect to the specifications in Chapter 2, Section 2.2 is evident. The participant sends a session request to the nearby providers and continues as P''. On the other hand, the room accepts session requests by the Service process which waits for a session request that matches the room current session and continues as S'. Processes Relocation and Updating are defined accordingly. Notice that a single attribute update construct can be used for specifying a set of possible updates. The behavior of the continuation processes P'', S', and U' are shown below:

$$\begin{aligned}
&P'' \triangleq\ (x = \mathit{this}.interest \wedge y = interestRply)(x,\ y,\ z). \\
&\quad [\mathit{this}.dest := z]()@\mathsf{ff}.Upd
\end{aligned}$$

$$\begin{aligned}
&Upd \triangleq (y = \mathit{this}.interest \wedge z = interestUpd)(x,\ y,\ z,\ l). \\
&\quad [\mathit{this}.dest := l]()@\mathsf{ff}.Upd
\end{aligned}$$

$$\begin{aligned}
&S' \triangleq (\\
&\quad (\mathit{this}.session,\ interestRply,\ \mathit{this}.name)@(id = z).\mathsf{0} \\
&\quad | \\
&\quad Service \\
&)
\end{aligned}$$

$$\begin{aligned}
&U' \triangleq [\mathit{this}.prevSession := \mathit{this}.session,\ \mathit{this}.session := x] \\
&\quad ((\mathit{this}.prevSession,\ \mathit{this}.session,\ interestUpd,\ \mathit{this}.name) \\
&\qquad @(interest = \mathit{this}.session \vee session = \mathit{this}.session).\mathsf{0} \\
&\qquad | \\
&\qquad Updating \\
&\quad)
\end{aligned}$$

Notice that ()@ff *denotes an output action on a false predicate. The execution of this action cannot be perceived by any component and is interpreted as a silent move in AbC as we will see in the operational semantics chapter. The attribute update is not an action and it takes place atomically with the first move of the following process, in our case this is a silent move.* □

3.2 Expressiveness of the AbC Calculus

In this section, we provide an evidence of the expressive power of the AbC calculus by showing how different communication models and interaction patterns can be easily rendered in AbC and advocate the use of attribute-based communication as a unifying framework to encompass different communication models.

3.2.1 Encoding channel-based interaction

The interaction primitives in AbC are purely based on attributes. In contrast to other process calculi where senders and receivers have to agree on an explicit channel or name, AbC relies on the satisfaction of predicates over attributes for deriving the interaction.

Attribute values in AbC can be modified by means of internal actions. Changing attribute values permits opportunistic interaction between components in the sense that an attribute update might provide new opportunities of interaction. This is because the selection of interaction partners depends on predicates over the attributes they expose. Changing the values of these attributes implies changing the set of possible partners and this is why modelling adaptivity in AbC is quite natural. Offering this possibility is difficult in channel-based process calculi. Indeed, we would like to argue that finding a compositional encoding in channel-based process calculi for the following simple AbC system is very difficult, if not impossible :

$$\begin{aligned}\Gamma_1 : (msg,\ \texttt{this}.b)@(\mathsf{tt}) \| \\ \Gamma_2 : ([\texttt{this}.a := 5]()@\mathsf{ff}.P \mid (y \leq \texttt{this}.a)(x, y).Q)\end{aligned}$$

If we assume that initially $\Gamma_1(b) = 3$ and $\Gamma_2(a) = 2$, we have that changing the value of the local attribute a to "5" by the left-hand side process in the second component gives it an opportunity of receiving the message "msg" from the process residing in the first component. One would argue that using restriction to hide local communication and bound input/output actions would be sufficient to encode such kind of behaviors in channel-based process calculi. However, this is not the case in the sense that bound

input/output actions are only willing to engage in communication when they are instantiated with concrete channel names. In the example above, the input action in the process at the right hand side of the interleaving operator of the second component is always enabled. This means that before the update, an input is available on the predicate $y \leq 2$ and after the update it is available on the predicate $y \leq 5$.

Looking from the opposite perspective one might ask whether it is possible to model channel based message passing in AbC. Indeed, a feature that is not present in AbC is the possibility of specifying a single name/channel where the exchange happens instantaneously, i.e., the possibility of relying on a channel that appears at the time of interaction and disappears afterwards. Attributes are always available in the attribute environment and cannot disappear when one would like them to do so. However, this is not a problem, since it is possible to exploit the fact that AbC predicates can check the message values. Thus, we can add the name of the channel where the exchange happens as a value in the message. The receiver is left with the responsibility to check the compatibility of that value with its receiving channel.

To show the correctness of this encoding, we choose the $b\pi$-calculus (EM01) as a representative for channel-based process calculi. The $b\pi$-calculus is a good choice because it uses broadcast instead of binary communication as a basic primitive for interaction which makes it a sort of variant of value-passing CBS (Pra91). Furthermore, channels in $b\pi$-calculus can be communicated like in π-calculus (MPW92) which is considered as one of the richest paradigms introduced for concurrency so far.

Based on a separation results presented in (EM99), it has been proved that $b\pi$-calculus and π-calculus are incomparable in the sense that there does not exist any uniform, parallel-preserving translation from $b\pi$-calculus into π-calculus up to any "reasonable" equivalence. On the other hand, in π-calculus a process can non-deterministically choose the communication partner while in $b\pi$-calculus cannot.

Proving the existence of a uniform and parallel-preserving encoding of $b\pi$-calculus into AbC up to some reasonable equivalence ensures at least the same separation results between AbC and π-calculus.

(Component Level)

$(\!|G|\!)_c \triangleq \emptyset : (\!|G|\!)_p \qquad (\!|P_1 \| P_2|\!)_c \triangleq (\!|P_1|\!)_c \parallel (\!|P_2|\!)_c$

$(\!|\nu \tilde{x} P|\!)_c \triangleq \nu \tilde{x} (\!|P|\!)_c$

(Process Level)

$(\!|\texttt{nil}|\!)_p \triangleq 0 \qquad (\!|\tau.G|\!)_p \triangleq ()@\mathsf{ff}.(\!|G|\!)_p$

$(\!|a(\tilde{x}).G|\!)_p \triangleq \Pi(y, \tilde{x}).(\!|G|\!)_p$
$\quad \texttt{with} \quad \Pi = (y = a) \quad \texttt{and} \quad y \notin n((\!|G|\!)_p)$

$(\!|\bar{a}\tilde{x}.G|\!)_p \triangleq (a, \tilde{x})@(a = a).(\!|G|\!)_p$

$(\!|(rec\ A\langle\tilde{x}\rangle).G)\langle\tilde{y}\rangle|\!)_p \triangleq (A(\tilde{x}) \triangleq (\!|G|\!)_p)$
$\qquad\qquad\qquad\qquad\qquad\qquad \texttt{where} \quad fn((\!|G|\!)_p) \subseteq \{\tilde{x}\}$
$(\!|G_1 + G_2|\!)_p \triangleq (\!|G_1|\!)_p + (\!|G_2|\!)_p$

Table 3: Encoding bπ-calculus into AbC

We consider a two-level syntax of $b\pi$-calculus (i.e., only static contexts (Mil89) are considered) as shown below.

$P ::= G \ \mid P_1 \| P_2 \ \mid \nu x P$

$G ::= \texttt{nil} \ \mid a(\tilde{x}).G \ \mid \bar{a}\tilde{x}.G \ \mid G_1 + G_2 \mid (rec\ A\langle\tilde{x}\rangle.G)\langle\tilde{y}\rangle$

Dealing with the one level $b\pi$-syntax would not add any difficulty concerning channel encoding; only the encoding of parallel composition and name restriction occurring under a prefix or a choice would be slightly more intricate. As reported in Table 3, the encoding of a $b\pi$-calculus process P is rendered as an AbC component $(\!|P|\!)_c$ with $\Gamma = \emptyset$. Notice that $(\!|G|\!)_c$ encodes a $b\pi$-sequential process while $(\!|P|\!)_c$ encodes the parallel composition of $b\pi$-sequential processes. The channel is rendered as the first element in the sequence of values. For instance, in the output action $(a, \tilde{x})@(a = a)$, a represents a channel name, so the input action $(y = a)(y, \tilde{x})$ will always check the first element of the received values to

decide whether to accept or discard the message. Notice that the predicate $(a = a)$ is satisfied by any Γ, however including the channel name in the predicate is crucial to encode name restriction correctly.

The formal definition which specifies what properties are preserved by this encoding and a proof sketch for the correctness of the encoding up to a specific behavioral equivalence will be presented in Chapter 5, Section 5.4.

3.2.2 Encoding interaction patterns

In this section, we provide insights on how the concept of *attribute-based communication* can be exploited to provide a general unifying framework encompassing different interaction patterns tailored for multiway interactions. We show how the notion of *group* in group-based (AC93; CKV01; HC99) and publish/subscribe-based (BN02; EFGK03) interaction patterns can be naturally rendered in AbC. Since these interaction patterns do not have formal descriptions, we proceed by relying on examples.

We start with group-based interaction patterns and show that when modelling a group name as an attribute in AbC, the constructs for joining or leaving a given group can be modelled as attribute updates, like in the following example:

$$\begin{aligned}\Gamma_1 : (msg,\ \texttt{this}.group)@(group = a) \ \| \\ \Gamma_2 : ((y = b)(x,\ y) \mid [\texttt{this}.group := c]()@\texttt{ff}) \ \| \ldots \\ \| \ \ \Gamma_7 : ((y = b)(x,\ y) \mid [\texttt{this}.group := a]()@\texttt{ff})\end{aligned}$$

We assume that initially we have $\Gamma_1(group) = b$, $\Gamma_2(group) = a$, and $\Gamma_7(group) = c$. Component 1 wants to send the message "msg" to group "a". Only Component 2 is allowed to receive it as it is the only member of group "a". Component 2 can leave group "a" and join "c" by performing an attribute update with a silent move. On the other hand, if Component 7 joined group "a" before "msg" is emitted then both of Component 2 and Component 7 will receive the message.

It is worth mentioning that a possible encoding of group communication into $b\pi$-calculus has been introduced in (EM01). The encoding is relatively complicated and does not guarantee the causal order of message

reception. "Locality" is neither a first class construct in $b\pi$-calculus nor in AbC. However, "locality" (in this case, the group name) can be naturally modeled as an attribute in AbC while in $b\pi$-calculus, more efforts are needed.

Publish/subscribe interaction patterns can be considered as special cases of the attribute-based ones. For instance, a natural modeling of the topic-based publish/subscribe model (EFGK03) into AbC can be accomplished by allowing publishers to broadcast messages with "tt" predicates (i.e., satisfied by all subscribers) and only subscribers can check the compatibility of the exposed publishers attributes with their subscriptions, see the following example:

$$\Gamma_1 : (msg,\ \texttt{this}.topic)@(\mathsf{tt}) \parallel \Gamma_2 : (y = \texttt{this}.subscription)(x,\ y) \parallel \\ \ldots \parallel \Gamma_n : (y = \texttt{this}.subscription)(x,\ y)$$

The publisher broadcasts the message "msg" tagged with a specific topic for all possible subscribers (the predicate "tt" is satisfied by all); subscribers receive the message if the topic matches their subscription.

The operational semantics of the AbC calculus abstracts from a specific underlying communication infrastructure that mediates the interaction between components. Thus, we do not model the exchange server or the broker that mediates the interaction between publishers and subscribers. We consider this an implementation related concern which we are going to discuss in details in Chapter 6.

Chapter 4

AbC Operational Semantics

In this chapter we first introduce the operational semantics of the *AbC* calculus and then we explore the modeling power of the *AbC* primitives through different case studies form different application domains. We use interaction fragments to show how the semantics rules apply.

The operational semantics of *AbC* is based on two relations. The transition relation $\longmapsto$ that describes the behavior of single components and the transition relation $\longrightarrow$ that relies on the former relation and describes system behaviors.

4.1 Operational semantics of component

We use the transition relation $\longmapsto \subseteq \text{Comp} \times \text{CLAB} \times \text{Comp}$ to define the local behavior of a component where Comp denotes the set of components and CLAB is the set of transition labels α generated by the following grammar:

$$\alpha ::= \lambda \quad | \quad \widetilde{\Pi(\tilde{v})} \qquad\qquad \lambda ::= \nu\tilde{x}\overline{\Pi}\tilde{v} \quad | \quad \Pi(\tilde{v})$$

The λ-labels are used to denote *AbC* output ($\nu\tilde{x}\overline{\Pi}\tilde{v}$) and input ($\Pi(\tilde{v})$) actions. The output and input labels contain the sender's predicate that

specifies the communication partners Π, and the transmitted values $\tilde{v}$. An output is called "bound" if its label contains a bound name (i.e., if $\tilde{x} \neq \emptyset$). The α-labels include an additional label $\widetilde{\Pi(\tilde{v})}$ to denote the case where a component is not able to receive a message. As it will be shown later in this section, this kind of labels is crucial to appropriately handle dynamic constructs like choice and awareness. Free names in α are specified as follows:

- $fn(\nu\tilde{x}\overline{\Pi}(\tilde{v})) = fn(\Pi(\tilde{v}))\backslash\tilde{x}$ and $fn(\Pi(\tilde{v})) = fn(\Pi) \cup \tilde{v}$
- $fn(\widetilde{\Pi(\tilde{v})}) = fn(\Pi) \cup \tilde{v}$

The free names of a predicate is the set of names occurring in that predicate except for attribute identifiers. Notice that `this`.a is only a reference to the value of the attribute identifier a. Only the output label has bound names (i.e., $bn(\nu\tilde{x}\overline{\Pi}\tilde{v}) = \tilde{x}$).

The transition relation $\longmapsto$ is formally defined in Table 4 and Table 5. We start by defining the set of rules used by a component to discard unwanted input messages and then we continue with the actual behavior of a single component. These rules will be later used to correctly define the behavior of systems and also individual components. We consider AbC terms up to α-conversion ($\equiv_\alpha$).

Discarding input. The rules for enabling components to discard input messages are presented in Table 4. The label $\widetilde{\Pi(\tilde{v})}$ is used to indicate discarding an input message.

Rule (**FBrd**) states that any sending component discards messages from other components and stay unchanged. Rule (**FRcv**) states that if one of the receiving requirements is not satisfied then the component will discard the message and stay unchanged.

Rule (**FUpd**) state that process $[\tilde{a} := \tilde{E}]P$ discards a message if process P is able to discard the same message after applying attribute updates i.e., $\Gamma[\tilde{a} \mapsto \tilde{v}]$ where $\forall a \in \tilde{a}$ and $\forall v \in \tilde{v}$, we have that: $\Gamma[a \mapsto v](a') = \Gamma(a')$ if $a \neq a'$ and v otherwise. The following structural congruence rule is used

FBrd

$$\Gamma : (\tilde{E})@\Pi.P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma : (\tilde{E})@\Pi.P$$

FRcv

$$\frac{[\![\Pi[\tilde{v}/\tilde{x}]]\!]_\Gamma \neq \mathsf{tt} \ \vee \ (\Gamma \not\models \Pi')}{\Gamma : \Pi(\tilde{x}).P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma : \Pi(\tilde{x}).P}$$

FUpd

$$\frac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad \Gamma[\tilde{a} \mapsto \tilde{v}] : P \xmapsto{\widetilde{\Pi(\tilde{w})}} \Gamma[\tilde{a} \mapsto \tilde{v}] : P}{\Gamma : [\tilde{a} := \tilde{E}]P \xmapsto{\widetilde{\Pi(\tilde{w})}} \Gamma : [\tilde{a} := \tilde{E}]P}$$

FZero

$$\Gamma : \mathsf{0} \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : \mathsf{0}$$

FAware1

$$\frac{[\![\Pi]\!]_\Gamma \doteq \mathsf{tt} \qquad \Gamma : P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma : P}{\Gamma : \langle \Pi \rangle P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma : \langle \Pi \rangle P}$$

FAware2

$$\frac{[\![\Pi]\!]_\Gamma \doteq \mathsf{ff}}{\Gamma : \langle \Pi \rangle P \xmapsto{\widetilde{\Pi'(\tilde{v})}} \Gamma : \langle \Pi \rangle P}$$

FSum

$$\frac{\Gamma : P_1 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_1 \qquad \Gamma : P_2 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_2}{\Gamma : P_1 + P_2 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_1 + P_2}$$

FInt

$$\frac{\Gamma : P_1 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_1 \qquad \Gamma : P_2 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_2}{\Gamma : P_1 | P_2 \xmapsto{\widetilde{\Pi(\tilde{v})}} \Gamma : P_1 | P_2}$$

Table 4: Discarding input

to collapse multiple consecutive attribute updates into one.

$$[a_1 := E_1][a_2 := E_2]P \equiv \begin{cases} [a_1 := E_1, a_2 := E_2]P & \text{if } a_1 \neq a_2 \\ [a_1 := E_2]P & \text{otherwise} \end{cases}$$

Rule (**FAware1**) states that process $\langle \Pi \rangle P$ discards a message even if Π evaluates to (tt) if process P is able to discard the same message. Rule (**FAware2**) states that if Π in process $\langle \Pi \rangle P$ evaluates to ff, process $\langle \Pi \rangle P$ will discard any message from other processes.

Rule (**FZero**) states that process $\mathsf{0}$ always discards messages from other processes. Rule (**FSum**) states that process $P_1 + P_2$ discards a message if both its subprocesses P_1 and P_2 can do so. The role of the discarding label is to keep dynamic constructs like awareness and choice from dis-

solving after a message refusal. Rule (**FInt**) has a similar meaning of Rule (**FSum**).

Example 4.1 (step 5/6)**.** *Assume that the current scheduled session for room r_1 is "Theory". A participant, say p_i with $id = 1$ and he is interested in "$Verification$", sent a session request for nearby providers. If room r_1 receives this request, it can discard it by taking the following transition.*

$$\Gamma_i : \overbrace{Service|Relocation|Updating}^{R} \xrightarrow{(role=\widetilde{Provider})(Verification,\ REQUEST,\ 1)} \Gamma_i : Service|Relocation|Updating$$

*Clearly, process Service discards the request because it does not satisfy its receiving predicate (i.e., $(x = Theory \wedge y = REQUEST)[Verification/x, REQUEST/y] \neq \mathtt{tt}$), so it applies rule (**FRcv**) and stays unchanged. Process Relocation also discards the request because it is not ready to accept messages, so it applies rule (**FAware2**) and stays unchanged. Finally process Updating discards the message for a similar reason to process Service and stays unchanged. The process R, which is the parallel composition of those processes, applies the rule (**FInt**) and discards the request.* □

Component behavior. The set of rules in Table 5 describes the behavior of a single AbC component. The symmetrical rules for **(Sum)** and **(Int)** are omitted.

Rule (**Brd**) evaluates the sequence of expressions $\tilde{E}$, say to $\tilde{v}$, and the predicate Π_1 to Π after replacing any reference (i.e., `this`.a) with its value according to the attribute environment Γ, and sends this information in the message, afterwards the process evolves to P.

Rule (**Rcv**) replaces the free occurrences of the input sequence variables $\tilde{x}$ in the receiving predicate Π with the corresponding message values $\tilde{v}$ and evaluates Π under the environment Γ. If the evaluation semantically equals to $\mathtt{tt}$ and the receiver environment Γ satisfies the sender predicate Π', the input action is performed and the substitution $[\tilde{v}/\tilde{x}]$ is applied to the continuation process P.

$$\textbf{Brd}\ \frac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad [\![\Pi_1]\!]_\Gamma = \Pi}{\Gamma : (\tilde{E})@\Pi_1.P \overset{\overline{\Pi}\tilde{v}}{\longmapsto} \Gamma : P} \qquad \textbf{Rcv}\ \frac{[\![\Pi[\tilde{v}/\tilde{x}]]\!]_\Gamma \doteq \mathsf{tt} \quad \Gamma \models \Pi'}{\Gamma : \Pi(\tilde{x}).P \overset{\Pi'(\tilde{v})}{\longmapsto} \Gamma : P[\tilde{v}/\tilde{x}]}$$

$$\textbf{Upd}\ \frac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad \Gamma[\tilde{a} \mapsto \tilde{v}] : P \overset{\lambda}{\longmapsto} \Gamma[\tilde{a} \mapsto \tilde{v}] : P'}{\Gamma : [\tilde{a} := \tilde{E}]P \overset{\lambda}{\longmapsto} \Gamma[\tilde{a} \mapsto \tilde{v}] : P'}$$

$$\textbf{Aware}\ \frac{[\![\Pi]\!]_\Gamma \doteq \mathsf{tt} \quad \Gamma : P \overset{\lambda}{\longmapsto} \Gamma' : P'}{\Gamma : \langle\Pi\rangle P \overset{\lambda}{\longmapsto} \Gamma' : P'} \qquad \textbf{Sum}\ \frac{\Gamma : P_1 \overset{\lambda}{\longmapsto} \Gamma' : P_1'}{\Gamma : P_1 + P_2 \overset{\lambda}{\longmapsto} \Gamma' : P_1'}$$

$$\textbf{Rec}\ \frac{\Gamma : P \overset{\alpha}{\longmapsto} \Gamma' : P' \quad K \triangleq P}{\Gamma : K \overset{\alpha}{\longmapsto} \Gamma' : P'} \qquad \textbf{Int}\ \frac{\Gamma : P_1 \overset{\lambda}{\longmapsto} \Gamma' : P_1'}{\Gamma : P_1 | P_2 \overset{\lambda}{\longmapsto} \Gamma' : P_1' | P_2}$$

Table 5: Component semantics

Rule (**Upd**) evaluates the sequence of expressions $\tilde{E}$ under the environment Γ, apply attribute updates, and then performs an action with a λ label if process P under the updated environment can do so.

Rule (**Aware**) evaluates the predicate Π under the environment Γ. If the evaluation semantically equals to tt, process $\langle\Pi\rangle P$ proceeds by performing an action with a λ-label and continues as P' if process P can perform the same action.

Rule (**Sum**) and its symmetric version represent the non-deterministic choice between the subprocesses P_1 and P_2 in the sense that if any of them say P_1 performs an action with a λ-label and becomes P_1' then the overall process continues as P_1'.

Rule (**Rec**) and rule (**Int**) are the standard rules for handling process definition and interleaving of the actions of two processes, respectively.

Example 4.2 (step 4/6)**.** *Assume that we have a participant p_1 with $id = 1$ and initial topic equals to "Verification". The participant agent P, running on the mobile device of participant p_1, can take the following transition.*

$$\Gamma_p : P \overset{\overline{(role=Provider)}(Verification,\ REQUEST,\ 1)}{\longmapsto} \Gamma_p[interest \mapsto Verification] : P'$$

$$\textbf{Comp}\ \frac{\Gamma : P \overset{\lambda}{\mapsto} \Gamma' : P'}{\Gamma : P \overset{\lambda}{\rightarrow} \Gamma' : P'} \qquad \textbf{C-Fail}\ \frac{\Gamma : P \overset{\widetilde{\Pi(\tilde{v})}}{\longmapsto} \Gamma : P}{\Gamma : P \overset{\Pi(\tilde{v})}{\longrightarrow}\ \Gamma : P} \qquad \textbf{Rep}\ \frac{C \overset{\gamma}{\rightarrow} C'}{!C \overset{\gamma}{\rightarrow} C' \| !C}$$

$$\tau\textbf{-Int}\ \frac{C_1 \overset{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\longrightarrow} C_1' \quad \Pi \simeq \mathsf{ff}}{C_1 \| C_2 \overset{\tau}{\rightarrow} C_1' \| C_2} \qquad \textbf{Res}\ \frac{C[y/x] \overset{\gamma}{\rightarrow} C' \quad y \notin n(\gamma) \wedge y \notin fn(C)\backslash\{x\}}{\nu x C \overset{\gamma}{\rightarrow} \nu y C'}$$

$$\textbf{Sync}\ \frac{C_1 \overset{\Pi(\tilde{v})}{\longrightarrow} C_1' \quad C_2 \overset{\Pi(\tilde{v})}{\longrightarrow} C_2'}{C_1 \parallel C_2 \overset{\Pi(\tilde{v})}{\longrightarrow} C_1' \parallel C_2'}$$

$$\textbf{Com}\ \frac{C_1 \overset{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\longrightarrow} C_1' \quad C_2 \overset{\Pi(\tilde{v})}{\longrightarrow} C_2' \qquad \begin{array}{c}\Pi \neq \mathsf{ff} \\ \tilde{x} \cap fn(C_2) = \emptyset\end{array}}{C_1 \parallel C_2 \overset{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\longrightarrow} C_1' \parallel C_2'}$$

$$\textbf{Hide1}\ \frac{C \overset{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\longrightarrow} C' \qquad \begin{array}{c}(\Pi \blacktriangleright y) \simeq \mathsf{ff} \\ y \in n(\Pi)\end{array}}{\nu y C \overset{\nu\tilde{x}\overline{\mathsf{ff}}\tilde{v}}{\longrightarrow} \nu y \nu\tilde{x} C'} \qquad \textbf{Hide2}\ \frac{C \overset{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\longrightarrow} C' \qquad \begin{array}{c}(\Pi \blacktriangleright y) \not\simeq \mathsf{ff} \\ y \in n(\Pi)\end{array}}{\nu y C \overset{\nu\tilde{x}\overline{\Pi \blacktriangleright y}\tilde{v}}{\longrightarrow} \nu y C'}$$

$$\textbf{Open}\ \frac{C[y/x] \overset{\overline{\Pi}\tilde{v}}{\longrightarrow} C' \quad \Pi \neq \mathsf{ff} \quad y \in \tilde{v}\backslash n(\Pi) \wedge y \notin fn(C)\backslash\{x\}}{\nu x C \overset{\nu y\overline{\Pi}\tilde{v}}{\longrightarrow} C'}$$

Table 6: System semantics

*The process P first applies the rule (**Upd**) to update its interest to "Verification" and then applies rule (**Brd**) and sends a session request to nearby providers.* □

4.2 Operational semantics of systems

An *AbC* system describes the global behavior of a component and the underlying communication between different components. We use the transition relation $\longrightarrow \subseteq \text{Comp} \times \text{SLAB} \times \text{Comp}$ to define the behavior of a system where Comp denotes the set of components and SLAB is the set of transition labels γ which are generated by the following

grammar:

$$\gamma \ ::= \nu\tilde{x}\overline{\Pi}\tilde{v} \quad | \quad \Pi(\tilde{v}) \quad | \quad \tau$$

The γ-labels extend λ with τ to denote silent moves (i.e., send on a false predicate $()@\mathsf{ff}$). The τ-label has no free or bound names. The definition of the transition relation $\longrightarrow$ depends on the definition of the relation $\longmapsto$ in the previous section in the sense that the effect of local behavior is lifted to the global one. The transition relation $\longrightarrow$ is formally defined in Table 6; there the symmetric rules for **τ-Int** and **Com** are omitted.

Rule (**Comp**) states that the relations $\longmapsto$ and $\longrightarrow$ coincide when performing either an input or output actions. Rule (**C-Fail**) states that any component $\Gamma : P$ can discard a message and stay unchanged if its local process is willing to do so. Rule (**Rep**) is standard for replication. Rule (**τ-Int**) and its symmetric rule model the interleaving between components C_1 and C_2 when performing a silent move (i.e., a send action $(\tilde{v})@\Pi$ with $\Pi \doteq \mathsf{ff}$). In this thesis, we will use $()@\mathsf{ff}$ to denote a silent action/move.

Rule (**Res**) states that component $\nu x C$ with a restricted name x can still perform an action with a γ-label as long as x does not occur in the names of the label and component C can perform the same action. If necessary, we allow renaming with conditions that ensure avoiding name clashing.

Rule (**Sync**) states that two parallel components C_1 and C_2 can synchronize while performing an input action. This means that the same message is received by both C_1 and C_2. Rule (**Com**) states that two parallel components C_1 and C_2 can communicate if C_1 can send a message with a predicate that is different from ff and C_2 can possibly receive that message.

Rules (**Hide1**) and (**Hide2**) are unique to AbC and introduce a new concept that we call predicate restriction "$\bullet \blacktriangleright x$" as reported in Table 7. In process calculi where broadcasting is the basic primitive for communication like CSP (Hoa78) and $b\pi$-calculus (MPW92), broadcasting on a private channel is equal to performing an internal action and no other process can observe the broadcast except the one that performed it.

For example in $b\pi$-calculus, if we let

$$
\begin{array}{lcl}
\mathsf{tt}\blacktriangleright x & = & \mathsf{tt} \\
\mathsf{ff}\blacktriangleright x & = & \mathsf{ff} \\
(a=m)\blacktriangleright x & = & \begin{cases} \mathsf{ff} & \texttt{if } x=m \\ a=m & \texttt{otherwise} \end{cases} \\
(\Pi_1 \wedge \Pi_2)\blacktriangleright x & = & \Pi_1\blacktriangleright x \ \wedge \ \Pi_2\blacktriangleright x \\
(\Pi_1 \vee \Pi_2)\blacktriangleright x & = & \Pi_1\blacktriangleright x \ \vee \ \Pi_2\blacktriangleright x \\
(\neg\Pi)\blacktriangleright x & = & \neg(\Pi\blacktriangleright x)
\end{array}
$$

Table 7: Predicate restriction $\bullet\blacktriangleright x$

$P = \nu a(P_1 \| \ P_2) \| \ P_3$ where $P_1 = \bar{a}v.Q$, $P_2 = a(x).R$, and $P_3 = b(x)$ then if P_1 broadcasts on a we would have that only P_2 can observe it since P_2 is within the scope of the restriction. P_3 and other processes only observe an internal action, so $P \xrightarrow{\tau} \nu a(Q \| R[v/x]) \| \ b(x)$.

This idea is generalized in AbC to what we call predicate restriction "$\bullet \blacktriangleright x$" in the sense that we either hide a part or the whole predicate using the predicate restriction operator "$\bullet \blacktriangleright x$" where x is a restricted name and the "$\bullet$" is replaced with a predicate. If the predicate restriction operator returns ff then we get the usual hiding operator like in CSP and $b\pi$-calculus because the resulting label is not exposed according to (τ-**Int**) rule (i.e., sending with a false predicate).

If the predicate restriction operator returns something different from ff then the message is exposed with possibly a smaller predicate and the restricted name remains private. Note that any private name in the message values (i.e., $\tilde{x}$) remains private if $(\Pi \blacktriangleright y) \doteq \mathsf{ff}$ as in rule (**Hide1**) otherwise it is not private anymore as in rule (**Hide2**). In other words, messages are sent on a channel that is partially exposed.

We would like to stress that the predicate restriction operator, that filters the exposure of the communication predicate either partially or completely, is very useful when modelling user-network interaction. The user observes the network as a single node and interacts with it through

a public channel and is not aware of how the messages are propagated through the network. Networks propagate messages between their nodes through private channels while exposing messages to users through public channels. For instance, if a network sends a message with the predicate $(keyword = \texttt{this}.topic \ \vee \ capability = fwd)$ where the name "fwd" is restricted then the message is exposed to the user at every node with forwarding capability in the network with this predicate $(keyword = \texttt{this}.topic)$. Network nodes observe the whole predicate but they receive the message only because they satisfy the other part of the predicate (i.e., $(capability = fwd)$). In the following Lemma, we prove that the satisfaction of a restricted predicate $\Pi \blacktriangleright x$ by an attribute environment Γ does not depend on the name x that is occurring in Γ.

Lemma 4.1. $\Gamma \models \Pi \blacktriangleright x$ *iff* $\forall v.\ \Gamma[v/x] \models \Pi \blacktriangleright x$ *for any environment* Γ*, predicate* Π*, and name* x*.*

Proof. The "if" implication is straightforward. For the "only if" implication, the proof is carried out by induction on the structure of Π.

- if $(\Pi = \mathsf{tt})$: according to Table 7, $(\mathsf{tt} \blacktriangleright x = \mathsf{tt})$ which means that the satisfaction of tt does not depend on x (i.e., $\Gamma \models \mathsf{tt} \blacktriangleright x$ iff $\Gamma \models \mathsf{tt}$). From Table 1, we have that tt is satisfied by all Γ, so it is easy to that if $\Gamma \models \mathsf{tt} \blacktriangleright x$ then $\forall v.\ \Gamma[v/x] \models \mathsf{tt} \blacktriangleright x$ as required.

- if $(\Pi = \mathsf{ff})$: according to Table 7, $(\mathsf{ff} \blacktriangleright x = \mathsf{ff})$ which again means that the satisfaction of ff does not depend on x. From Table 1, we have that ff is not satisfied by any Γ, so this case holds vacuously.

- if $(\Pi = (a = m) \blacktriangleright x)$: according to Table 7, we have two cases:

  - if $(x = m)$ then $\Pi = \mathsf{ff}$ and by induction hypotheses, the case holds vacuously.
  - if $(x \neq m)$ then $\Pi = (a = m)$, according to Table 1, we have that $\Gamma \models (a = m)$ iff $\Gamma(a) = m$. Since $x \neq m$, then $\Gamma(a) = m$ holds for any value of x in Γ and we have that if $\Gamma \models (a = m) \blacktriangleright x$ then $\forall v.\ \Gamma[v/x] \models (a = m) \blacktriangleright x$ as required.

- if $(\Pi = \Pi_1 \wedge \Pi_2)$: according to Table 7, $(\Pi_1 \wedge \Pi_2) \blacktriangleright x = (\Pi_1 \blacktriangleright x \wedge \Pi_2 \blacktriangleright x)$. From Table 1, we have that $\Gamma \models (\Pi_1 \blacktriangleright x \wedge \Pi_2 \blacktriangleright x)$ iff $\Gamma \models \Pi_1 \blacktriangleright x$ and $\Gamma \models \Pi_2 \blacktriangleright x$. By induction hypotheses, we have that

if ($\Gamma \models \Pi_1 \blacktriangleright x$ then $\forall v.\ \Gamma[v/x] \models \Pi_1 \blacktriangleright x$) and if ($\Gamma \models \Pi_2 \blacktriangleright x$ then $\forall v.\ \Gamma[v/x] \models \Pi_2 \blacktriangleright x$).
$\Gamma \models (\Pi_1 \blacktriangleright x \wedge \Pi_2 \blacktriangleright x)$ iff $\forall v.(\Gamma[v/x] \models \Pi_1 \blacktriangleright x \wedge \Gamma[v/x] \models \Pi_2 \blacktriangleright x)$ and now we have that if $\Gamma \models (\Pi_1 \wedge \Pi_2) \blacktriangleright x$ then $\forall v.\ \Gamma[v/x] \models (\Pi_1 \wedge \Pi_2) \blacktriangleright x$ as required.

- if $(\Pi = \Pi_1 \vee \Pi_2)$: This case if analogous to the previous one.
- if $(\Pi = \neg\Pi)$: According to Table 7, $(\neg\Pi) \blacktriangleright x = \neg(\Pi \blacktriangleright x)$. From Table 1, we have that $\Gamma \models \neg(\Pi \blacktriangleright x)$ iff not $\Gamma \models (\Pi \blacktriangleright x)$. By induction hypotheses, we have that if (not $\Gamma \models \Pi \blacktriangleright x$ then $\forall v.$ not $\Gamma[v/x] \models \Pi \blacktriangleright x$) and now we have that if $\Gamma \models \neg(\Pi) \blacktriangleright x$ then $\forall v.$ $\Gamma[v/x] \models \neg(\Pi) \blacktriangleright x$ as required.

□

Rule (**Open**) states that a component has the ability to communicate a private name to other components. This rule is different from the one in π-calculus in the sense that AbC represents multiparty settings. This implies that the scope of the private name x is not expanded to include a group of other components but rather the scope is dissolved. In other words, when a private name is communicated in AbC then the name is not private anymore. Note that, a component that is sending on a false predicate (i.e., $\Pi \doteq \mathsf{ff}$) cannot open the scope.

Example 4.3 (step 6/6)**.** *Let us assume that a participant, say p_1 with $id = 1$ and with "Theory" as his initial topic of interest, sent a session request for nearby providers. The participant p_1 applies rule (**Comp**) and take the following transition:*

$$\Gamma_p : P \xrightarrow{\overline{(role=Provider)}(Theory,\ REQUEST,\ 1)} \Gamma_p[interest \mapsto Theory] : P'$$

*If a room, say r_1 scheduled for the session "Theory", receives this message, it applies rule (**Comp**) and take the following transition:*

$$\Gamma_1 : Service|Relocation|Updating \xrightarrow{(role=Provider)(Theory,\ REQUEST,\ 1)} \Gamma_1 : S'[Theory/x,\ REQUEST/y,\ 1/z] \\ |Relocation|Updating$$

*All other rooms will just discard the request and apply rule (**C-Fail**). Now the overall system evolves by applying rule (**Com**) as follows:*

$$S \xrightarrow{\overline{(role=Provider)}(Theory,\ REQUEST,\ 1)} \Gamma_1 : (\ S'[Theory/x,\ REQUEST/y,\ 1/z]\ |\ Relocation|Updating)\ \|\ \Gamma_p : P'\ \|\ \Gamma_2 : R_2 \|\ \ldots \|\ \Gamma_n : R_n$$

The components $\Gamma_2 : R_2,\ \ldots,\ \Gamma_n : R_n$ represent the other rooms in the conference venue. □

4.3 Case Studies: The AbC calculus at work

We start with a very simple case study to show how predicates can be used to derive the attribute-based interaction in an intuitive way. Afterward we proceed with more involved case studies to show the modeling power of the *AbC* primitives.

4.3.1 TV Streaming channels

We show how the *AbC* calculus can be used to model two application scenarios, both of them from the realm of TV streaming channels like Sky Online, Sky Go, Netflix, etc. The first scenario is simple and demonstrates how natural and intuitive it is to use *AbC* to model run-time collective formation in these systems. The second scenario is more involved and shows how predicates over attributes can be used to tell apart messages coming from different parties (as it is usually done via channels in calculi relying on more standard communication models).

Basic scenario

In this scenario, we consider a TV broadcaster (e.g., CNN) represented by the process `CNN`, and two receivers represented by the processes `RcvA` and `RcvB`, respectively, as shown in Table 8. The overall system is expressed as the parallel composition $\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_A : \texttt{RcvA} \parallel \ldots \parallel \Gamma_B : \texttt{RcvB}$, where the dots refer to other possible broadcasters or receivers.

$$\texttt{CNN} \triangleq (v_s, \texttt{this.Qbrd})@\Pi_{sport}.\texttt{CNN} + (v_n, \texttt{this.Qbrd})@\Pi_{news}.\texttt{CNN} + [\texttt{Qbrd} := \texttt{LD}]()@\texttt{ff}.\texttt{CNN} + [\texttt{Qbrd} := \texttt{HD}]()@\texttt{ff}.\texttt{CNN}$$

$$\texttt{RcvA} \triangleq (\texttt{y = HD})(x,\ y).\texttt{RcvA} + [\texttt{Genre} := \texttt{Sport}]()@\texttt{ff}.\texttt{RcvA} + [\texttt{Genre} := \texttt{News}]()@\texttt{ff}.\texttt{RcvA}$$

$$\texttt{RcvB} \triangleq (\texttt{tt})(x,\ y).\texttt{RcvB} + \ldots$$

Table 8: First scenario: process definitions

CNN periodically broadcasts Sport or News and targets different groups of receivers based on the predicates Π_{sport} and Π_{news} given in Table 9. Π_{sport} targets the group of receivers who want to watch Sport (Genre = Sport) provided that those receivers have subscribed to CNN (CNN-Sub = tt). On the other hand, Π_{news} targets the group of receivers who want to watch News (Genre = News). The quality of the broadcasted multimedia varies according to different factors (i.e., low bandwidth, etc). CNN channel non-deterministically chooses to broadcast low-definition (Qbrd := LD) or high-definition (Qbrd := LD) multimedia. The receiving processes RcvA and RcvB almost have the same behaviour except that RcvA is only interested in high quality broadcasts while RcvB is willing to accept broadcasts of any quality. So they either accept the broadcast that their attributes in Γ_A and Γ_B satisfy, or change the genre.

Now we show a fragment of the possible interactions in this scenario. For the sake of readability, we shall use grey-shaded box to indicate those components that are involved in the evolution. Assume that initially the attribute environments Γ_{cnn}, Γ_A and Γ_B are defined as follows:

$$\Gamma_{cnn} = \{(\texttt{Qbrd}, \texttt{HD}), \ldots\}, \qquad \Gamma_A = \{(\texttt{Genre}, \texttt{News}), \ldots\}$$
$$\Gamma_B = \{(\texttt{Genre}, \texttt{News}), \ldots\}$$

Assume also that the process CNN initiates the interaction by broadcasting high quality News. As shown in Table 10, both RcvA and RcvB can join the collective and receive the broadcast because their attributes satisfy

$$\Pi_{sport} = (\texttt{Genre} = \texttt{Sport}) \wedge (\texttt{CNN-Sub} = \texttt{tt})$$

$$\Pi_{news} = (\texttt{Genre} = \texttt{News})$$

Table 9: First scenario: predicates

$$\Gamma_{cnn}:\texttt{CNN} \parallel \Gamma_A:\texttt{RcvA} \parallel \ldots \parallel \Gamma_B:\texttt{RcvB}$$

$$\xrightarrow{\overline{\Pi_{news}}(v_s,\texttt{HD})} \quad \Gamma_{cnn}:\texttt{CNN} \parallel \Gamma_A:\texttt{RcvA} \parallel \ldots \parallel \Gamma_B:\texttt{RcvB}$$

$$\vdots$$

$$\xrightarrow{\overline{\texttt{ff}}()} \quad \Gamma_{cnn}[\texttt{Qbrd} \mapsto \texttt{LD}]:\texttt{CNN} \parallel \Gamma_A:\texttt{RcvA} \parallel \ldots \parallel \Gamma_B:\texttt{RcvB}$$

$$\xrightarrow{\overline{\Pi_{news}}(v_s,\texttt{LD})} \quad \Gamma_{cnn}:\texttt{CNN} \parallel \Gamma_A:\texttt{RcvA} \parallel \ldots \parallel \Gamma_B:\texttt{RcvB}$$

Table 10: First scenario: interaction fragment

the condition of the broadcast (i.e., $\Gamma_A \models \Pi_{news}$ and $\Gamma_B \models \Pi_{news}$). The system evolves according to rule (**Com**) in Table 6. However, after a while CNN chooses to lower the quality of multimedia ($\texttt{Qbrd} \mapsto \texttt{LD}$) to cope with some situations (i.e., low bandwidth) and the system evolves according to rule (τ**-Int**). Finally, CNN continues broadcasting News and in this case RcvA chooses to leave the collective because the quality of the broadcast does not satisfy its receiving predicate (i.e., $(y = \texttt{HD})[\texttt{LD}/y] \neq \texttt{tt}$), while RcvB stays because it has no requirement for the input quality (i.e., $(\texttt{tt})(x,\ y)$).

Scenario with subscription

In the previous scenario we assume that receivers expect to interact with only content providers (i.e., CNN, BBC, *etc.*) that broadcast a unique kind of data (i.e., Multimedia). For this reason we do not consider compatibility

$$
\begin{array}{ll}
\texttt{CNN} & \triangleq (v_s,\ \texttt{this.Content})@\Pi_{sport}.\texttt{CNN} \\
& + (v_n,\ \texttt{this.Content})@\Pi_{news}.\texttt{CNN} \\
\\
\texttt{CNN-Ctr} & \triangleq [\texttt{Trans} := \texttt{Renew}] \\
& \qquad (\texttt{tt},\ \texttt{this.Content},\ \texttt{this.Trans})@\Pi_{conf}.\texttt{CNN-Ctr} \\
& + [\texttt{Trans} := \texttt{check}] \\
& \qquad (\texttt{ff},\ \texttt{this.Content},\ \texttt{this.Trans})@\Pi_{canc}.\texttt{CNN-Ctr} \\
\\
\texttt{RcvA} & \triangleq \Pi_{brd}(x,\ y).\texttt{RcvA} \\
& \quad + [\texttt{CNN-Paid} := \texttt{tt}]()@\texttt{ff}.\texttt{RcvA} \\
& \quad + [\texttt{Genre} := \texttt{News}]()@\texttt{ff}.\texttt{RcvA} \\
& \quad + \Pi_{sub}(x,\ y,\ z).[\texttt{CNN-valid} := \texttt{x}, \texttt{CNN-Sub} := \texttt{x}]()@\texttt{ff}.\texttt{RcvA} \\
& \quad + \Pi_{unsub}(x,\ y,\ z).[\texttt{CNN-Sub} := \texttt{x}]()@\texttt{ff}.\texttt{RcvA} \\
\\
\texttt{RcvB} & \triangleq \Pi_{brd}(x,\ y).\texttt{RcvB} \\
& \quad + \ldots
\end{array}
$$

Table 11: Second scenario: process definitions

of data on the receiving inputs. In channel-based process calculi, this is equivalent to the idea that all processes communicate with each other by agreeing on only one channel/name. However, in reality processes interact with different kinds of processes and receive different kinds of data. Usually processes rely on different channels/names to tell apart messages coming from different parties. In our calculus we do not have channels and rely only on predicates to handle this situation.

In the following, we extend the previous scenario to show how predicates can replace channels. We assume that the receiving processes are able to subscribe or cancel their subscriptions for some contents. In our scenario, if the processes `RcvA` and `RcvB` want to subscribe or cancel their subscription for `CNN`, they have to communicate with another party which is called CNN center (`CNN-Ctr`) as shown in Table 11. The overall system is expressed as the parallel composition $\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr} \parallel \Gamma_A : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$. The process `CNN-Ctr` identifies its contents as subscription contents (`Content = Subscription`) and periodically looks for users who are willing to subscribe or cancel their subscription for `CNN`. The process `CNN-Ctr` either sets the type of transaction (`Trans`) to `Renew` and broadcasts a subscription confirmation message to all processes that satisfy the predicate Π_{conf} or sets the type of transaction to `check` and broadcasts a subscription cancel message to all processes that satisfy the predicate Π_{canc}. The behaviour of `CNN` is reduced to only broadcasting Sports or News and now `CNN` identifies its contents as Multimedia in its attributes environment Γ_{cnn}. The behaviour for the processes `RcvA` and `RcvB` is extended to enable them to communicate with different parties and receive different types of data in an appropriate way. This is done by imposing predicates on the input actions. Hence, the process receives data only from those whose messages satisfy its input predicates. `RcvA` and `RcvB` either receive broadcast from those satisfying the predicates Π_{brd}, Π_{sub} or Π_{unsub}, pay CNN subscription (`CNN-Paid` $\mapsto$ `tt`), or change the genre. It is obvious that a subscriber can cheat and pretend that he already paid by changing the attribute `CNN-Paid` to `tt` without communicating with the subscription center `CNN-Ctr`. However, this is not the point of the example in which we intend to show how the communication

$$\Pi_{conf} = (\texttt{CNN-Paid} = \texttt{tt}) \ \wedge \ (\texttt{CNN-valid} = \texttt{ff})$$

$$\Pi_{canc} = (\texttt{CNN-Paid} = \texttt{ff}) \ \wedge \ (\texttt{CNN-valid} = \texttt{ff})$$

$$\Pi_{brd} = (y = \texttt{Multimedia})$$

$$\Pi_{sub} = (y = \texttt{subscription}) \ \wedge \ (z = \texttt{Renew})$$

$$\Pi_{unsub} = (y = \texttt{subscription}) \ \wedge \ (z = \texttt{check})$$

Table 12: Second scenario: predicates

links between interacting partners are established in a simple and intuitive scenario. Once `RcvA` and `RcvB` subscribe, their subscriptions become valid (`CNN-Valid` $\mapsto$ `tt`) and they can receive paid broadcasts.

The predicates $\Pi_{conf}, \ldots, \Pi_{unsub}$ are specified in Table 12. The predicates Π_{sport} and Π_{news} are the same as in the previous scenario and the rest can be intuitively described as follows:

- Π_{conf} targets users who paid and their subscription expired.
- Π_{canc} targets users who did not pay and their subscription expired.
- Π_{brd} targets the processes who broadcast Multimedia where y is a place holder for the second element of the received values.
- Π_{sub} targets processes for subscription where y and z are place holders for received values.
- Π_{unsub} targets processes for canceling subscription where y and z are place holders for received values.

Now we show a fragment of the possible interactions when process `CNN -Ctr` is involved. Assume that the initial states of the environments Γ_{cnn}, Γ_{ctr}, Γ_A, and Γ_B associated to the processes `CNN`, `CNN-Ctr`, `RcvA` and `RcvB`

respectively are as follows:

$$\Gamma_{cnn} = \{(\texttt{Content}, \texttt{Multimedia}), \dots\}$$
$$\Gamma_{ctr} = \{(\texttt{Trans}, \phi), (\texttt{Content}, \texttt{Subscription}), \dots\}$$
$$\Gamma_A = \{(\texttt{CNN-Sub}, \texttt{ff}), (\texttt{Genre}, \texttt{Sport}), (\texttt{CNN-Paid}, \texttt{ff}), (\texttt{CNN-valid}, \texttt{ff}), \dots\}$$
$$\Gamma_B = \{(\texttt{CNN-Sub}, \texttt{tt}), (\texttt{CNN-valid}, \texttt{tt}), (\texttt{Genre}, \texttt{Sport}), \dots\}$$

Assume that process CNN initiates the interaction by broadcasting Sports which targets subscribed receivers as shown in Table 13. Only RcvB is allowed to receive the broadcast because its attributes satisfy the condition of the broadcast (i.e., $\Gamma_B \models \Pi_{sport}$). The system evolves according to rule (**Com**) in Table 6 and RcvA, CNN-Ctr stay unchanged. Then RcvA pays the subscription by changing the value of its attribute CNN-Paid to tt and the system evolves according to rule (τ**-Int**). At this moment CNN-Ctr starts looking for processes who have expired subscription and already paid for renewal. It takes a step by setting its attribute Trans to Renew and broadcasts a subscription confirmation message. Only RcvA is allowed to receive this message because it is willing to subscribe and its attributes satisfy Π_{conf}. The system evolves according to rule (**Com**). RcvA takes the next step to update its subscription validity. Finally, CNN continues broadcasting Sports and in this case both RcvA and RcvB are allowed to receive the broadcast because now also the attributes of RcvA satisfy the condition of the broadcast (i.e., $\Gamma_A \models \Pi_{sport}$). Hence, the system evolves according to rule (**Com**).

4.3.2 Stable Marriage Problem

We consider the classical stable marriage problem (SMP) (GS62), a problem of finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element.

In our scenario, we consider n men and n women, where each person has ranked all members of the opposite sex in order of preferences; we have to engage the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr} \parallel \Gamma_A : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$

$\xrightarrow{\overline{\Pi_{sport}}(v_s,\ \texttt{Multimedia})}$

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr} \parallel \Gamma_A : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$

$\xrightarrow{\overline{\texttt{ff}}()}$

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr}$
$\parallel \Gamma_A[\texttt{CNN-Paid} \mapsto \texttt{tt}] : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$

$\xrightarrow{\overline{\Pi_{conf}}(\texttt{tt},\ \texttt{Subscription},\ \texttt{Renew})}$

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr}[\texttt{Trans} \mapsto \texttt{Renew}] : \texttt{CNN-Ctr}$
$\parallel \Gamma_A : [\texttt{CNN-valid} := \texttt{tt}, \texttt{CNN-Sub} := \texttt{tt}]()@\texttt{ff}.\texttt{RcvA}$
$\parallel \Gamma_B : \texttt{RcvB}$

$\xrightarrow{\overline{\texttt{ff}}()}$

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr}$
$\parallel \Gamma_A[\texttt{CNN-valid} \mapsto \texttt{tt},\ \texttt{CNN-Sub} \mapsto \texttt{tt}] : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$

$\xrightarrow{\overline{\Pi_{sport}}(v_s,\ \texttt{Multimedia})}$

$\Gamma_{cnn} : \texttt{CNN} \parallel \Gamma_{ctr} : \texttt{CNN-Ctr} \parallel \Gamma_A : \texttt{RcvA} \parallel \Gamma_B : \texttt{RcvB}$

Table 13: Second scenario: interaction fragment

is deemed stable. For convenience we assume there are no ties; thus, if a person is indifferent between two or more possible partners he/she is nevertheless required to rank them in some order. The marriage scenario can be modeled in AbC as follows:

$$Man_1 \| \ldots \| Man_n \| Woman_1 \| \ldots \| Woman_n$$

Men and women interact in parallel and each is modeled as an AbC component, Man_i of the form $\Gamma_{m,i} : M$ and $Woman_i$ of the form $\Gamma_{w,i} : W$. The attribute environments of men and women, $\Gamma_{m,i}$ and $\Gamma_{w,i}$, contain the following attributes:

- *partner*: identifies the current partner identity; in case a person is not engaged yet, the value of its partner is -1;
- $preferences$: a ranking list of the person preferences, the head of this list is the person's preferred partner;
- M_{id} for man and W_{id} for woman, identify their identities;
- $exPartner$ for a woman, identifies her ex-fiancé.

The structures of process M, specifying the behavior of a man, and the process W, specifying the behavior of a woman, are defined as follows:

$$M \triangleq [\texttt{this}.partner := \texttt{Top}(\texttt{this}.preferences), \\ \texttt{this}.preferences := \ominus(\texttt{this}.preferences)]\ \text{a.M'}$$

$$W \triangleq \quad \text{b.}\ (\ \langle \texttt{BOF}(\texttt{this}.partner,\ y)\rangle\ W_1\ + \\ \langle \neg\texttt{BOF}(\texttt{this}.partner,\ y)\rangle\ W_2\ )\ |\ W$$

A man, M, picks his first best from the ranking list "$\texttt{this}.preferences$" and assumes it to be his partner. This element is removed from his preferences. In the same transition he proposes to this possible partner by executing action a (to be specified later) and then continues as M' . The prefix `this` is a reference to the value assigned to the attribute identifier "$preferences$". Functions `Top(arg)` and $\ominus$`(arg)` both take a list as an

argument. The former returns the first element of the list if the list is not empty and the empty string otherwise, while the latter returns the list resulting from the removal of its first element.

On the other hand, the behavior of a woman, W, is activated by receiving a proposal, i.e., executing action b (to be specified later). A woman either accepts this proposal from a "y" man if she will be better off with him and continues as W_1 or refuses it if she prefers her current fiancè and continues as W_2. The parallel composition with W ensures that the woman is always willing to consider new proposals. $\texttt{BOF}(arg_1,\ arg_2)$ is a boolean function that takes as arguments the current partner and the new man, respectively, and determines whether the woman will be better off with the new man or not, given her current fiancè and her preferences. If she is not engaged, this function will always return true.

This example shows other uses of the awareness construct $\langle\Pi\rangle$ where it is not only used to wait for some attribute values to change in response to environmental changes, but also can be used to implement a form of guarded process as in the π-calculus. In this way, a process behavior can be enabled in response to the received messages. In our example, the function $\texttt{BOF}(arg_1,\ arg_2)$ enables the process W_1 or disables the process W_2 in response of executing the action b which indicates a proposal reception. In the main running example, we already showed how to use the awareness operator to wait for attribute values to change. More specifically, in the process *Relocation*, Example 3.3, the relocation process waits until the relocate attribute becomes true (i.e., $\langle \texttt{this}.relocate = \mathsf{tt}\rangle$).

If we further specify the action "a" and the process M' in M, the action "b" and the processes W_1 and W_2 in W, the behavior of a man and a woman becomes:

$$
\begin{aligned}
M \triangleq\ & [\texttt{this}.partner := \texttt{Top}(\texttt{this}.preferences),\\
& \texttt{this}.preferences := \ominus(\texttt{this}.preferences)]\\
& (propose,\ \texttt{this}.M_{id})@(W_{id} = \texttt{this}.partner).\\
& (x = invalid)(x).M
\end{aligned}
$$

$$W \triangleq (x = propose)(x,\ y).\ (\ \langle \texttt{BOF}(\texttt{this}.partner,\ y)\rangle\ [\texttt{this}.exPartner := \texttt{this}.partner,\ \texttt{this}.partner := y]\ (invalid)@(M_{id} = \texttt{this}.exPartner).0 + \langle \neg\texttt{BOF}(\texttt{this}.partner,\ y)\rangle\ (invalid)@(M_{id} = y).0\ \ )\ |\ W$$

Obviously, action "a" is a proposal message to be sent to the selected partner. This message contains a label "*propose*" to indicate the type of the message and the sender identity M_{id}. The man stays engaged as long as he does not receive an invalidation message from the woman he proposed to. The invalidation message contains a label "*invalid*" to indicate the message type. If this message is received, the man starts all over again and picks his second best and so on.

On the other hand, action "b" is used to receive a proposal message from a "y" man. If the woman prefers "y", she will consider her current partner as her ex-partner, get engaged to "y", and send an invalidation message to her ex-fiancé so that he looks for another partner. This is also true for the case when she is not engaged, but in this case she will send an invalidation message with a predicate $(M_{id} = -1)$ which will not be received by anyone. If she prefers her partner, she will send the invalidation message to "y". □

Although the interaction in this specific scenario is based on partners identities, the interaction in AbC is usually more general and assumes anonymity between the interacting partners. Interaction relies on predicates over attributes that can be changed at anytime. This means that components interact without a prior agreement between each other.

Example 4.4 (Interaction fragment)**.** *Let us assume that a man, say m_4 with $id = 4$ and initial list of preferences $l_m = \langle 4,\ 1,\ 3,\ 2\rangle$, wants to propose to the woman he ranked as his first best, in our case this is woman w_4. On the other hand, woman w_4 is currently engaged to man m_3 and her list of preferences is $l_w = \langle 2,\ 1,\ 4,\ 3\rangle$. Man m_4 removes his first best from his ranking list and assumes it to be his partner, sends a proposal to woman w_4 by applying rule (**Comp**), and takes the following*

transition.

$$\Gamma_m : M \xrightarrow{\overline{(W_{id}=4)}(propose,\ 4)} \Gamma_p[partner \mapsto 4,\ preferences \mapsto \langle 1,\ 3,\ 2\rangle] : (x = invalid)(x).M$$

*Man m_4 considers himself as engaged until he receives an invalidation message from his partner. On the other hand, woman w_4 receives this message by applying rule (**Comp**) and takes the following transition:*

$$\Gamma_w : W \xrightarrow{(W_{id}=4)(propose,\ 4)} \Gamma_w : (\ \langle BOF(this.partner,\ 4)\rangle [this.exPartner := this.partner,\ this.partner := 4] (invalid)@(M_{id} = this.exPartner).0 + \langle \neg BOF(this.partner,\ 4)\rangle (invalid)@(M_{id} = 4).0\ \ ) \mid W$$

*The overall system evolves by applying rule (**Com**). Woman w_4 can take another step and get engaged to man m_4, since she will be better off him (i.e., $BOF(3,\ 4)|_{l_w=\langle 2,\ 1,\ 4,\ 3\rangle} = \mathsf{tt}$). So she updates her current partner and sends an invalidation message to her ex-partner by applying rule (**Comp**) as shown in the following transition:*

$$\xrightarrow{\overline{(M_{id}=3)}(invalid)} \Gamma_w[exPartner \mapsto 3,\ partner \mapsto 4] : 0 \mid W$$

□

4.3.3 A swarm robotics scenario in *AbC*

We consider a scenario where a swarm of robots spreads throughout a given disaster area with the goal of locating and rescuing possible victims. All robots playing the same role execute the same code, defining the functional behavior, and a set of adaptation mechanisms, regulating the interactions among robots and their environments. All robots initially

play the explorer role to search for victims in the environment. Once a robot finds a victim, it changes its role to "rescuer" and sends victim's information to nearby explorers. The collective (the swarm) starts forming in preparation for the rescuing procedure. As soon as another robot receives victim's information, it changes its role to "helper" and moves to join the rescuers-collective. The rescuing procedure starts only when the collective formation is complete. During exploration, in case of critical battery level, a robot enters a power saving mode until it is recharged.

The swarm robotics model exploits the fact that a process running on a robot can either read the values of some attributes that are provided by its sensors or read and update the other attributes in its attribute environment. Reading the values of the attributes controlled by sensors either provides information about the robot environment or information about the current status of the robot. We could say that in the former case the model formalises *context-awareness* while in the latter case it formalizes *self-awareness*. For instance, when reading the value of the *collision* attribute in the attribute environment $\Gamma(collision) = \mathtt{tt}$ the robot becomes aware that a collision with a wall in the arena is imminent and this triggers an adaptation mechanism to change its direction. On the other hand, reading the value of the *batteryLevel* attribute $\Gamma(batteryLevel) = 15\%$ makes the robot aware that its battery level is critical (i.e., $< 20\%$) and this triggers an adaptation mechanism to halt the movement and to take the robot into the power saving mode.

We assume that each robot has a unique identity (*id*) and since the robot acquires information about its environment or its own status by reading the values provided by sensors, no additional assumption about its initial state is needed. It is worth mentioning that sensors and actuators are not modelled by AbC because they represent the robot internal infrastructure while AbC model represents the programmable behaviour of the robot (i.e., its running code).

The robotics scenario is modelled as a set of parallel AbC components, each of which represents a robot ($Robot_1 \| \ldots \| Robot_n$) and each robot has the following form ($\Gamma_i : P_R$). The behaviour of a single robot is modelled

in the following AbC process P_R:

$$P_R \triangleq (Rescuer + Explorer) | RandWalk | IsMoving$$

The robot follows a random walk in exploring the disaster arena. The robot can become a "rescuer" when he becomes aware of the presence of a victim by locally reading the value of an attribute controlled by its sensors or remain an "explorer" and keep sending queries for information about the victim from nearby robots whose role is either "*rescuer*" or "*helper*".

If sensors recognise the presence of a victim and the value of "*victimPerceived*" becomes "tt", the robot updates its "*state*" to "*stop*" (which triggers an actuation signal to halt the actuators and stop movement), computes the victim position and the number of the required robots to rescue the victim and stores them in the attributes "*vPosition*" and "*count*" respectively, changes its role to "*rescuer*", and waits for queries from nearby explorers. Once a message from an explorer is received, the robot sends back the victim information to the requesting robot addressing it by its identity "*id*" and the collective (i.e., the swarm) starts forming in preparation for the rescuing procedure.

$$\begin{aligned}
&Rescuer \triangleq \\
&\quad \langle \texttt{this}.victimPerceived = \texttt{tt} \rangle [\texttt{this}.state := stop,\ \texttt{this}.count := 3, \\
&\quad \texttt{this}.vPosition := <3,4>,\ \texttt{this}.role := rescuer]()@\texttt{ff}. \\
&\quad (y = qry \wedge z = explorer)(x,\ y,\ z). \\
&\quad (\texttt{this}.vPosition,\ \texttt{this}.count,\ ack,\ \texttt{this}.role)@(id = x)
\end{aligned}$$

On the other hand, if the victim is still not perceived, the robot continuously sends queries for information about the victim to the nearby robots whose role is either "rescuer" or "*helper*". The query message contains the robot identity "*this.id*" , a special name "*qry*" to indicate the request type, and the current role of the robot "this.*role*". If an acknowledgement arrives containing victim's information, the robot changes its role

to "*helper*" and starts the helping procedure.

$$\begin{aligned} &\mathit{Explorer} \triangleq \\ &\qquad (\texttt{this}.\mathit{id},\ \mathit{qry},\ \texttt{this}.\mathit{role})@(\mathit{role} = \mathit{rescuer} \vee \mathit{role} = \mathit{helper}). \\ &\qquad (((z = \mathit{rescuer} \vee z = \mathit{helper}) \wedge x = \mathit{ack})(\mathit{vpos},\ c,\ x,\ z). \\ &\qquad\ [\texttt{this}.\mathit{role} := \mathit{helper}]()@\mathsf{ff}.\mathit{Helper} + \mathit{Rescuer} + \mathit{Explorer}) \end{aligned}$$

Remark 4.1. *The interaction between an explorer robot currently running and a rescuer robot that is waiting for a request from nearby explorers suggests a possible way of modelling binary communication like in π-calculus (MPW92). Rendezvous can be modelled in a similar way by defining an attribute to count the number of needed acknowledments to signal synchronisation.*

The "*Helper*" process defined below is triggered by receipt of the victim information from the rescuer-collective as mentioned above.

$$\begin{aligned} &\mathit{Helper} \triangleq [\texttt{this}.\mathit{vPosition} := \mathit{vpos},\ \texttt{this}.\mathit{target} := \mathit{vpos}]()@\mathsf{ff}. \\ &\qquad (\\ &\qquad\quad \langle \texttt{this}.\mathit{position} = \texttt{this}.\mathit{target} \rangle [\texttt{this}.\mathit{role} := \mathit{rescuer}]()@\mathsf{ff} \\ &\qquad\quad | \\ &\qquad\quad \langle c > 1 \rangle (y = \mathit{qry} \wedge z = \mathit{explorer})(x,\ y,\ z). \\ &\qquad\quad (\texttt{this}.\mathit{vPosition},\ c - 1,\ \mathit{ack},\ \texttt{this}.\mathit{role})@(\mathit{id} = x) \\ &\qquad) \end{aligned}$$

The helping robot stores the victim position in the attribute "*vPosition*" and updates its target to be the victim position. This triggers the actuators to move to the specified location. The robot moves towards the victim but at the same time is willing to respond to other robots queries, in case more than one robot is needed for the rescuing procedure. Once the robot reaches the victim (i.e., its position coincides with the victim position), the robot changes its role to "*rescuer*" and joins the rescuer-collective.

The "*RandWalk*" process is defined below. This process computes a random direction to be followed by the robot. Once a collision is detected by the proximity sensor, a new random direction is calculated.

$$\begin{aligned} &\mathit{RandWalk} \triangleq [\texttt{this}.\mathit{direction} := 2\pi \mathit{rand}()]()@\mathsf{ff}. \\ &\qquad\qquad\qquad\qquad \langle \texttt{this}.\mathit{collision} = \mathsf{tt} \rangle \mathit{RandWalk} \end{aligned}$$

Finally, process "*IsMoving*" captures the status of the battery level in a robot at any time. Once the battery level drops into a critical level

(i.e., less than 20%), the robot changes its status to "*stop*" which results in halting the actuators and the robot enters the power saving mode. The robot stays in this mode until it is recharged to at least 90% and then it starts moving again.

$$\begin{aligned} IsMoving \triangleq\ & \langle \texttt{this}.state = move \wedge \neg(\texttt{this}.batteryLevel > 20\%)\rangle \\ & [\texttt{this}.state := stop]()@\texttt{ff}.\langle \texttt{this}.batteryLevel \geq 90\%\rangle \\ & [\texttt{this}.state := move]()@\texttt{ff}.IsMoving \end{aligned}$$

For simplifying the presentation, in this scenario we are not modelling the charging task and assume that this task is accomplished according to some predefined procedure. It is worth mentioning that if more victims are found in the arena, different rescuer-collectives will be spontaneously formed to rescue them. To avoid forming multiple collectives for the same victim, we assume that sensors only detect isolated victims. Light-based message communication (OGCD10) between robots can be used. Thus once a robot has reached a victim, it signals with a specific color light to other robots not to discover the victim next to it (PBMD15). Since we do not model the failure recovery in this scenario, we assume that all robots are fault-tolerant and they cannot fail. For more details, a runtime environment for supporting the linguistic primitives of AbC can be found at the following website `http://lazkany.github.io/AbC/`. There we provide also a short tutorial to provide some intuition about how to use these primitives for programming.

Example 4.5 (Interaction fragment)**.** *Let us assume that the role of* $Robot_1$ *is "rescuer" and* $Robot_2$ *is "explorer". Robot*$_2$ *can send a query to nearby rescuing or helping robots (i.e.,* $Robot_1$*) by using rule (**Comp**) and generate this transition:*

$$Robot_2 \xrightarrow{\overline{(role=rescuer \vee role=helping)}(2,\ qry,\ explorer)} \Gamma_2 : (P_2 | P_3)$$

On the other hand, $Robot_1$ *can receive this query by using rule (**Comp**) and generate this transition:*

$Robot_1$

$$\xrightarrow{(role{=}rescuer \vee role{=}helping)(2,\ qry,\ explorer)}$$

$$\Gamma_1 : (P_1'[2/x,\ qry/y,\ explorer/z] | P_3)$$

*Other robots which are not addressed by communication discard the message by applying rule (**C-Fail**). Now the overall system evolves by applying rule (**Com**) as follows:*

S

$$\xrightarrow{\overline{(role{=}rescuer \vee role{=}helping)}(2,\ qry,\ explorer)}$$

$$\Gamma_1 : (P_1'[2/x,\ qry/y,\ explorer/z] | P_3)$$
$$\| \ \Gamma_2 : (P_2 | P_3) \| \ \Gamma_3 : P_{R_3} \| \ \ldots \| \ \Gamma_n : P_{R_n}$$

□

Chapter 5

Behavioral Theory for AbC

In this chapter, we define a behavioral theory for AbC. We start by introducing a reduction barbed congruence, then we present an equivalent definition of a labeled bisimulation. At the end of the chapter, we extract some equational laws and we sketch the proof of the correctness of the encoding of Section 3.2.1 up to strong reduction barbed congruence.

5.1 Reduction barbed congruence

In the behavioral theory, two terms are considered as equivalent if they cannot be distinguished by any external observer. The choice of observables is important to assess models of concurrent systems and their equivalences. For instance, in the π-calculus both message transmission and reception are considered to be observable. However, this is not the case in AbC because sending is non-blocking and only message transmission can be observed. It is important to notice that the transition $C \xrightarrow{\Pi(\tilde{v})} C'$ does not necessarily mean that C has performed an input action but rather it means that C *might* have performed an input action. Indeed, this transition might happen due to the application of one of two different rules in Table 6, namely (**Comp**) which guarantees reception and (**C-Fail**) which

models non-reception. Hence, input actions cannot be observed by an external observer and only output actions are observable in AbC. In this thesis we use the term "barb" as synonymous with observable, following the works in (BDP99a; MS92). In what follows, we shall use the following notations:

- $C \xrightarrow{\tau} C'$ iff $\exists \tilde{x}, \tilde{v}$, and Π such that $C \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C'$ and $\Pi \doteq \mathsf{ff}$.
- $\Rightarrow$ denotes $(\xrightarrow{\tau})^*$.
- $\stackrel{\gamma}{\Rightarrow}$ denotes $\Rightarrow\xrightarrow{\gamma}\Rightarrow$ if $(\gamma \neq \tau)$.
- $\stackrel{\hat{\gamma}}{\Rightarrow}$ denotes $\Rightarrow$ if $(\gamma = \tau)$ and $\stackrel{\gamma}{\Rightarrow}$ otherwise.
- $\multimap$ denotes $\xrightarrow{\gamma}$ where γ is an output or $\gamma = \tau$.

Definition 2 (External context)**.** *An external context $\mathcal{C}[\bullet]$ is a component term with a hole, denoted by $[\bullet]$. The external contexts of the* AbC *calculus are generated by the following grammar:*

$$\mathcal{C}[\bullet] ::= \quad [\bullet] \quad | \quad [\bullet] \| C \quad | \quad C \| [\bullet] \quad | \quad \nu x[\bullet] \quad | \quad ![\bullet]$$

Definition 3 (Barb)**.** *Let $C{\downarrow_\Pi}$ mean that component C can send a message with a predicate Π' (i.e., $C \xrightarrow{\nu\tilde{x}\overline{\Pi'}\tilde{v}}$ where $\Pi' \doteq \Pi$ and $\Pi' \neq \mathsf{ff}$). We write $C \Downarrow_\Pi$ if $C \multimap^* C' \downarrow_\Pi$ for some C'. From now on, we consider the predicate Π to denote only its meaning, not its syntax. In other words, we consider predicates up to semantic equivalence $\doteq$.*

Definition 4 (Barb Preservation)**.** *$\mathcal{R}$ is barb-preserving iff for every $(C_1, C_2) \in \mathcal{R}$, $C_1{\downarrow_\Pi}$ implies $C_2 \Downarrow_\Pi$*

Definition 5 (Reduction Closure)**.** *$\mathcal{R}$ is reduction-closed iff for every $(C_1, C_2) \in \mathcal{R}$, $C_1 \multimap C_1'$ implies $C_2 \multimap^* C_2'$ for some C_2' such that $(C_1', C_2') \in \mathcal{R}$*

Definition 6 (Context Closure)**.** *$\mathcal{R}$ is context-closed iff for every (C_1, C_2) $\in \mathcal{R}$ and for all external contexts $\mathcal{C}[\bullet]$, $(\mathcal{C}[C_1], \mathcal{C}[C_2]) \in \mathcal{R}$*

Now, everything is in place to define reduction barbed congruence. We define notions of strong and weak barbed congruence to reason about

AbC components following the definition of maximal sound theory by Honda and Yoshida (HY95). This definition is a slight variant of Milner and Sangiorgi's barbed congruence (MS92) and it is also known as open barbed bisimilarity (SW03).

Definition 7 (Weak Reduction Barbed Congruence)**.** *A weak reduction barbed congruence is a symmetric relation $\mathcal{R}$ over the set of* AbC*-components which is barb-preserving, reduction closed, and context-closed.*

Two components are weak barbed congruent, written $C_1 \cong C_2$, if $(C_1, C_2) \in \mathcal{R}$ for some weak reduction barbed congruence relation $\mathcal{R}$. The strong reduction congruence "$\simeq$" is obtained in a similar way by replacing $\Downarrow$ with $\downarrow$ and $\twoheadrightarrow^*$ with $\rightarrow$.

Lemma 5.1. *If $C_1 \cong C_2$ then*

- *$C_1 \Downarrow_\Pi$ iff $C_2 \Downarrow_\Pi$*
- *$C_1 \twoheadrightarrow^* C_1'$ implies $C_2 \twoheadrightarrow^* \cong C_1'$ where $\cong C_1'$ denotes a component that is weakly bisimilar to C_1'.*

Proof. (We prove each statement separately)

- The proof of first statement proceeds by induction on the length of the derivation $\twoheadrightarrow_n^*$ where n is the number of derivations. We only prove the "if implication" and the "only if implication" follows in a similar way.
  - Base case, $n = 0$: Since $C_1 \cong C_2$, we have that $C_1 \downarrow_\Pi$ implies $C_2 \Downarrow_\Pi$ as indicated in Definition 7 and Definition 4 respectively. From Definition 3, we have that $C_1 \downarrow_\Pi$ if $C_1 \twoheadrightarrow_0^* C_1' \downarrow_\Pi$ and $C_1 \equiv C_1'$ for some C_1' and $n = 0$. In other words, $C_1 \Downarrow_\Pi^0$ implies $C_2 \Downarrow_\Pi$ as required.
  - Suppose that $\forall k \leq n$: $C_1 \Downarrow_\Pi^k$ implies $C_2 \Downarrow_\Pi$ where $C_1 \Downarrow_\Pi^k$ denotes $C_1 \twoheadrightarrow_k^* C_1' \downarrow_\Pi$. It is sufficient to prove the claim for $k+1$.
    Now, we have that $C_1 \twoheadrightarrow_{k+1}^* C_1' \downarrow_\Pi = C_1 \rightarrow C_1'' \twoheadrightarrow_k^* C_1' \downarrow_\Pi$ for some C_1''. Since $C_1 \cong C_2$ and from Definition 7 and Definition 5, we have that $C_1 \rightarrow C_1''$ implies $C_2 \twoheadrightarrow^* C_2''$ for some C_2''

such that $C_1'' \cong C_2''$. Since $C_1'' \rightarrowtriangle_k^* C_1' \downarrow_\Pi$ and from induction hypothesis, we have that $C_1'' \Downarrow_\Pi^k$ implies $C_2'' \Downarrow_\Pi$. As a result, we have that $C_1 \Downarrow_\Pi$ implies $C_2 \Downarrow_\Pi$ as required.

- Again the proof of second statement proceeds by induction on the length of the derivation $\rightarrowtriangle_n^*$ where n is the number of derivations.
  - Base case, $n = 1$: Since $C_1 \cong C_2$ and from Definition 7 and Definition 5, we have that $C_1 \rightarrowtriangle C_1'$ implies $C_2 \rightarrowtriangle^* C_2'$ for some C_2' such that $C_1' \cong C_2'$. In other words, $C_1 \rightarrowtriangle_1^* C_1'$ implies $C_2 \rightarrowtriangle^* \cong C_1'$ as required.
  - Suppose that $\forall k \leq n$: $C_1 \rightarrowtriangle_k^* C_1'$ implies $C_2 \rightarrowtriangle^* \cong C_1'$. It is sufficient to prove the claim for $k+1$.
    Now, we have that $C_1 \rightarrowtriangle_{k+1}^* C_1' = C_1 \rightarrowtriangle C_1'' \rightarrowtriangle_k^* C_1'$ for some C_1''. Since $C_1 \cong C_2$, we have that $C_1 \rightarrowtriangle C_1''$ implies $C_2 \rightarrowtriangle^* C_2''$ for some C_2'' such that $C_1'' \cong C_2''$. Since $C_1'' \rightarrowtriangle_k^* C_1'$ and by induction hypothesis, we have that $C_1'' \rightarrowtriangle_k^* C_1'$ implies $C_2'' \rightarrowtriangle^* \cong C_1'$.
    As a result, we have that $C_1 \rightarrowtriangle^* C_1'$ implies $C_2 \rightarrowtriangle^* \cong C_1'$ as required.

□

5.2 Bisimulation Proof Methods

In this section, we define a notion of bisimulation for AbC components, given in terms of an LTS. We prove that bisimilarity coincides with the reduction barbed congruence, introduced in the previous section, and thus represents a valid tool for proving that two components are reduction barbed congruent.

Definition 8 (Weak Bisimulation)**.** *A symmetric binary relation $\mathcal{R}$ over the set of* AbC*-components is a weak bisimulation if for every action γ, whenever $(C_1, C_2) \in \mathcal{R}$ and γ is of the form τ, $\Pi(\tilde{v})$, or $(\nu \tilde{x}\overline{\Pi}\tilde{v}$ with $\Pi \neq \mathsf{ff})$, it holds that:*

$$C_1 \xrightarrow{\gamma} C_1' \text{ implies } C_2 \stackrel{\hat{\gamma}}{\Rightarrow} C_2' \text{ and } (C_1', C_2') \in \mathcal{R}$$

where every predicate Π occurring in γ is matched by its semantics meaning in $\hat{\gamma}$. Two components C_1 and C_2 are weakly bisimilar, written $C_1 \approx C_2$ if there exists a weak bisimulation $\mathcal{R}$ relating them. Strong bisimilarity, "$\sim$", is defined in a similar way by replacing $\stackrel{\hat{\gamma}}{\Rightarrow}$ with $\xrightarrow{\gamma}$.

It is easy to prove that $\sim$ and $\approx$ are equivalence relations by relying on the classical arguments of (Mil89). However, our bisimilarity enjoys a much more interesting property: closure under any external context.

The following Lemma is useful to prove that a component with a restricted name does not need any renaming when performing a τ action. We will use it in the proof of Lemma 5.4.

Lemma 5.2. *$C[y/x] \Rightarrow C'$ implies $\nu x C \Rightarrow \nu y C'$ if and only if $y \notin fn(C)$.*

Proof. The proof proceeds by induction on the length of the derivation $\Rightarrow_n$ where n is the number of derivations.

- Base Case, $n = 0$:
  $C[y/x] \equiv_\alpha C'$ which implies $\nu x C \equiv_\alpha \nu y C[y/x]$ where $\equiv_\alpha$ is the structural congruence under α-conversion.

- Suppose that $\forall k \leq n$: $C[y/x] \Rightarrow_k C'$ implies $\nu x C \Rightarrow_k \nu y C'$
  if $C[y/x] \Rightarrow_{n+1} C'$, then we have that $C[y/x] \Rightarrow_n C'' \xrightarrow{\tau} C'$ for some C''. This implies that $\nu x C \Rightarrow_n \nu y C''$ and $C'' \xrightarrow{\tau} C'$ which means that $\nu y C'' \xrightarrow{\tau} \nu y C'$.

  In other words, $C'' \xrightarrow{\tau} C'$ implies $C''[y/y] \xrightarrow{\tau} C'$. Now we can apply (Res) rule. Since $y \notin fn(C'') \backslash \{y\}$ and $y \notin n(\tau)$, we have that $\nu y C'' \xrightarrow{\tau} \nu y C'$ and we have that $\nu x C \Rightarrow \nu y C'$ as required.

□

In the next three lemmas, we prove that our bisimilarity is preserved by parallel composition, name restriction, and replication.

Lemma 5.3 ($\sim$ and $\approx$ are preserved by parallel composition)**.** *Let C_1 and C_2 be two components such that:*

- *$C_1 \sim C_2$ implies $C_1 \| C \sim C_2 \| C$ for all components C.*
- *$C_1 \approx C_2$ implies $C_1 \| C \approx C_2 \| C$ for all components C.*

Proof. (We only prove the second statement)
It is sufficient to prove that the relation $\mathcal{R} = \{(C_1 \| C, C_2 \| C) |$ for all C such that $(C_1 \approx C_2)\}$ is a weak bisimulation. Depending on the last rule applied to derive the transition $C_1 \| C \xrightarrow{\gamma} \hat{C}$, we have several cases.

- Assume that $C_1 \| C \xrightarrow{\tau} \hat{C}$: Then the last applied rule is (τ-Int) or its symmetrical counterpart.

- If (τ-Int) is applied then $\hat{C} = C_1' \| C$ and $C_1 \xrightarrow{\tau} C_1'$. Since $C_1 \approx C_2$ then there exists C_2' such that $C_2 \Rightarrow C_2'$ and $(C_1' \approx C_2')$. By applying (τ-Int) several times, we have that $C_2 \| C \Rightarrow C_2' \| C$ and $(C_1' \| C, C_2' \| C) \in \mathcal{R}$
- If the symmetrical counterpart of (τ-Int) is applied then $\hat{C} = C_1 \| C'$ and $C \xrightarrow{\tau} C'$. So it is immediate to have that $C_2 \| C \Rightarrow C_2 \| C'$ and $(C_1 \| C', C_2 \| C') \in \mathcal{R}$

- Assume that $C_1 \| C \xrightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} \hat{C}$ with $\hat{x} \cap fn(C) = \emptyset$ and $\Pi \neq \mathtt{ff}$, then the last applied rule is (Com) or its symmetrical counterpart.
  - If (Com) is applied then $\hat{C} = C_1' \| C'$, $C_1 \xrightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} C_1'$ and $C \xrightarrow{\Pi(\tilde{v})} C'$. Since $C_1 \approx C_2$ then there exists C_2' such that $C_2 \xRightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} C_2'$ and $(C_1' \approx C_2')$. By an application of (Com) and several application of (τ-Int), we have that $C_2 \| C \xRightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} C_2' \| C'$ and $(C_1' \| C', C_2' \| C') \in \mathcal{R}$
  - If the symmetrical counterpart of (Com) is applied then $\hat{C} = C_1' \| C'$, $C_1 \xrightarrow{\Pi(\tilde{v})} C_1'$ and $C \xrightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} C'$. So it is immediate to have that $C_2 \| C \xRightarrow{\nu \tilde{x} \overline{\Pi} \tilde{v}} C_2' \| C'$ and $(C_1' \| C', C_2' \| C') \in \mathcal{R}$

- $C_1 \| C \xrightarrow{\Pi(\tilde{v})} \hat{C}$, then the last applied rule is (Sync) and $\hat{C} = C_1' \| C'$, $C_1 \xrightarrow{\Pi(\tilde{v})} C_1'$, and $C \xrightarrow{\Pi(\tilde{v})} C'$. Since $C_1 \approx C_2$ then there exists C_2' such that $C_2 \xRightarrow{\Pi(\tilde{v})} C_2'$ and $(C_1' \approx C_2')$. By an application of (Sync) and several application of (τ-Int), we have that $C_2 \| C \xRightarrow{\Pi(\tilde{v})} C_2' \| C'$ and $(C_1' \| C', C_2' \| C') \in \mathcal{R}$.

The strong case of bisimulation ($\sim$) follows in a similar way. □

Lemma 5.4 ($\sim$ and $\approx$ are preserved by name restriction)**.** *Let C_1 and C_2 be two components, then the following statements hold:*

- *$C_1 \sim C_2$ implies $\nu x C_1 \ \sim \ \nu x C_2$ for all names x.*
- *$C_1 \approx C_2$ implies $\nu x C_1 \ \approx \ \nu x C_2$ for all names x.*

Proof. (We only prove the second statement)
It is sufficient to prove that the relation $\mathcal{R} = \{(C, B) |\ C = \nu x C_1,\ B = \nu x C_2 \text{ with } (C_1 \approx C_2)\}$ is a weak bisimulation. We have several cases depending on the performed action in deriving the transition $C \xrightarrow{\gamma} \hat{C}$.

- If $(\gamma = \tau)$ then only rule (Res) is applied. if (Res) is applied, then $C_1[y/x] \xrightarrow{\tau} C_1'$ and $\hat{C} = \nu y C_1'$. As $(C_1 \approx C_2)$, We have that $C_2[y/x] \Rightarrow C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2 and several applications of (Res), we have that $B \Rightarrow \nu y C_2'$ and $(\nu y C_1', \nu y C_2') \in \mathcal{R}$.

- If $(\gamma = \nu\tilde{y}\overline{\Pi}\tilde{v})$ then either rule (Open), (Res), (Hide1) or (Hide2) is applied.

  - If (Open) is applied, then $x \in (\tilde{v} \cup \tilde{y}) \backslash n(\Pi)$ and $C_1[z/x] \xrightarrow{\overline{\Pi}\tilde{v}} C_1'$ with $\hat{C} = C_1'$. As $(C_1 \approx C_2)$, we have that $C_2[z/x] \stackrel{\overline{\Pi}\tilde{v}}{\Longrightarrow} C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2, an application of (Open), and several applications of (Res), we have that $B \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} C_2'$ and $(C_1', C_2') \in \mathcal{R}$.

  - If (Res) is applied, then $C_1[z/x] \xrightarrow{\nu\tilde{y}\overline{\Pi}\tilde{v}} C_1'$ and $\hat{C} = \nu z C_1'$. As $(C_1 \approx C_2)$, we have that $C_2[z/x] \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2 and several applications of (Res), we have that $B \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} \nu z C_2'$ and $(\nu z C_1', \nu z C_2') \in \mathcal{R}$

  - If (Hide1) is applied, then $C_1 \xrightarrow{\nu\tilde{y}\overline{\Pi}\tilde{v}} C_1'$ and $\hat{C} = \nu x \nu\tilde{y} C_1'$. As $(C_1 \approx C_2)$, we have that $C_2 \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2, an application of (EHide1), and several applications of (Res), we have that $B \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} \nu x \nu\tilde{y} C_2'$ and $(\nu x \nu\tilde{y} C_1', \nu x \nu\tilde{y} C_2') \in \mathcal{R}$

  - If (Hide2) is applied, then $C_1 \xrightarrow{\nu\tilde{y}\overline{\Pi}\tilde{v}} C_1'$ and $\hat{C} = \nu x C_1'$. As $(C_1 \approx C_2)$, we have that $C_2 \stackrel{\nu\tilde{y}\overline{\Pi}\tilde{v}}{\Longrightarrow} C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2, an application of (EHide2), and several applications of (Res), we have that $B \stackrel{\nu\tilde{y}\overline{\Pi \blacktriangleright x}\tilde{v}}{\Longrightarrow} \nu x C_2'$ and $(\nu x C_1', \nu x C_2') \in \mathcal{R}$

- If $(\gamma = \Pi(\tilde{v}))$ then $x \notin n(\gamma)$ and only rule (Res) is applied. So we have that $C_1[y/x] \xrightarrow{\Pi(\tilde{v})} C_1'$ and $\hat{C} = \nu y C_1'$. As $(C_1 \approx C_2)$, we have that $C_2[y/x] \stackrel{\Pi(\tilde{v})}{\Longrightarrow} C_2'$ with $(C_1' \approx C_2')$. By Lemma 5.2 and several applications of (Res), we have that $B \stackrel{\Pi(\tilde{v})}{\Longrightarrow} \nu y C_2'$ and $(\nu y C_1', \nu y C_2') \in \mathcal{R}$

The strong case of bisimulation $(\sim)$ follows in a similar way. $\square$

Lemma 5.5 ($\sim$ and $\approx$ are preserved by replication)**.** *Let C_1 and C_2 be two components such that:*

- $C_1 \sim C_2$ *implies* $!C_1 \ \sim !C_2$.
- $C_1 \approx C_2$ *implies* $!C_1 \ \approx !C_2$.

Proof. (We only prove the second statement)
Given that $C_1 \approx C_2$, we have that there is a weak bisimulation relation, say $\mathcal{R}$, that relate them where $\mathcal{R} = \{(C_1, C_1) \mid C_1 \approx C_2\}$. The closure of $\mathcal{R}$ under parallel composition is also a weak bisimulation as shown by Lemma 5.3. We will use the notation $\mathcal{R}_\parallel$ to denote the closure of $\mathcal{R}$ under parallel composition. Now it is sufficient to prove that the relation $\mathcal{R}' = \{(!C_1, !C_2)\} \bigcup \{(C' \| !C_1, C'' \| !C_2) |\ (C', C'') \in \mathcal{R}_\parallel)\}$ is a weak bisimulation. The proof follows easily by applying rule (Rep). if $!C_1 \xrightarrow{\gamma} \hat{C}$, so we have that $C_1 \xrightarrow{\gamma} C_1'$ and $\hat{C} = C_1' \| !C_1$. As $(C_1 \approx C_2)$, then there exists C_2' such that $C_2 \stackrel{\gamma}{\Rightarrow} C_2'$ with $(C_1' \approx C_2')$. By an application of rule (Rep) and several applications of rule (Comp), we have that $!C_2 \stackrel{\gamma}{\Rightarrow} C_2' \| !C_2$ and $(C_1' \| !C_1,\ C_2' \| !C_2) \in \mathcal{R}'$ as required.

The strong case of bisimulation ($\sim$) follows in a similar way. □

As an immediate consequence of Lemma 5.3, Lemma 5.4, and Lemma 5.5, we have that $\sim$ and $\approx$ are congruence relations (i.e., closed under any external AbC context). We are now set to show that our bisimilarity represents a proof technique for establishing reduction barbed congruence.

Theorem 5.1 (Soundness)**.** *Let C_1 and C_2 be two components such that:*

- $C_1 \sim C_2$ *implies* $C_1 \ \simeq \ C_2$.
- $C_1 \approx C_2$ *implies* $C_1 \ \cong \ C_2$.

Proof. (We only prove the second statement)
It is sufficient to prove that bisimilarity is barb-preserving, reduction-closed, and context-closed.

- (Barb-preservation): By the definition of the barb $C_1{\downarrow_\Pi}$ if $C_1 \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}}$ for an output label $\nu\tilde{x}\overline{\Pi}\tilde{v}$ with $\Pi \neq \mathsf{ff}$. As $(C_1 \approx C_2)$, we have that also $C_2 \stackrel{\nu\tilde{x}\overline{\Pi}\tilde{v}}{\Longrightarrow}$ and $C_2 \Downarrow_\Pi$.

- (Reduction-closure): $C_1 \twoheadrightarrow C_1'$ means that either $C_1 \xrightarrow{\tau} C_1'$ or $C_1 \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_1'$. As $(C_1 \approx C_2)$, then there exists C_2' such that either $C_2 \Rightarrow C_2'$ or $C_2 \xRightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_2'$ with $(C_1' \approx C_2')$. So $C_2 \twoheadrightarrow^* C_2'$.

- (Context-closure): Let $(C_1 \approx C_2)$ and let $\mathcal{C}[\bullet]$ be an arbitrary AbC-context. By induction on the structure of $\mathcal{C}[\bullet]$ and using Lemma 5.3, Lemma 5.4, and Lemma 5.5, we have that $\mathcal{C}[C_1] \approx \mathcal{C}[C_2]$.

In conclusion, we have that $C_1 \cong C_2$ as required. □

Finally, we prove that our bisimilarity is more than a proof technique, but rather it represents a complete characterization of the reduction barbed congruence.

Lemma 5.6 (Completeness)**.** *Let C_1 and C_2 be two components, then the following statements hold:*

- *$C_1 \simeq C_2$ implies $C_1 \sim C_2$.*
- *$C_1 \cong C_2$ implies $C_1 \approx C_2$.*

Proof. (We only prove the second statement)
It is sufficient to prove that the relation $\mathcal{R} = \{(C_1, C_2) \mid C_1 \cong C_2\}$ is a weak bisimulation.

1. Suppose that $C_1 \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_1'$ for any Π and a sequence of values $\tilde{v}$ where $\Pi \neq \mathsf{ff}$. We build up a context to mimic the effect of this transition. Our context has the following form:

$$\mathcal{C}[\bullet] \triangleq [\bullet] \parallel \prod_{i\in I} \Gamma_i : \Pi_i(\tilde{x}_i).\langle \tilde{x}_i = \tilde{v}\rangle(\tilde{x}_i,\ a)@(in = a)$$

$$\parallel \prod_{j\in J} \Gamma_j : (y = a)(\tilde{x}_j,\ y).(\tilde{x}_j,\ b)@(out = b)$$

where $|\tilde{x_i}| = |\tilde{x_j}|$, $I \cap J = \emptyset$ for all J and $\Gamma_j \models (in = a)$, and the names a and b are fresh. Π_i is an arbitrary predicate. We use the notation $\langle \tilde{x}_i = \tilde{v}\rangle$ to denote $\langle (x_{i,1} = v_1) \wedge (x_{i,2} = v_2) \wedge \cdots \wedge (x_{i,n} = v_n)\rangle$ where $n = |\tilde{x_i}|$ and $\prod_{i\in I} \Gamma_i : P_i$ to denote the parallel composition of all components $\Gamma_i : P_i$, for $i \in I$. To be able to mimic the effects

of the transition $C_1 \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_1'$ by the above context we need to assume that $(\Gamma_i \models \Pi)$ and Π_i is satisfied given the sequence of values $\tilde{v}$. Intuitively, the existence of a barb on $(in = a)$ indicates that the action has not yet happened, whereas the presence of a barb on $(out = b)$ together with the absence of the barb on $(in = a)$ ensures that the action happened.

As $\cong$ is context-closed, $C_1 \cong C_2$ implies $\mathcal{C}[C_1] \cong \mathcal{C}[C_2]$. Since $C_1 \xrightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_1'$, it follows that:

$$\mathcal{C}[C_1] \twoheadrightarrow^* C_1' \parallel \prod_{i\in I} \Gamma_i : 0 \parallel \prod_{j\in J} \Gamma_j : (\tilde{v},\ b)@(out = b) \ = \ \hat{C_1}$$

with $\hat{C_1} \not\Downarrow_{(in=a)}$ and $\hat{C_1} \Downarrow_{(out=b)}$.
The reduction sequence above must be matched by a corresponding reduction sequence $\mathcal{C}[C_2] \twoheadrightarrow^* \hat{C_2} \cong \hat{C_1}$ with $\hat{C_2} \not\Downarrow_{(in=a)}$ and $\hat{C_2} \Downarrow_{(out=b)}$. By Lemma 5.1 and the conditions on the barbs, we get the structure of the above reduction sequence as follows:

$$\mathcal{C}[C_2] \twoheadrightarrow^* C_2' \parallel \prod_{i\in I} \Gamma_i : 0 \parallel \prod_{j\in J} \Gamma_j : (\tilde{v},\ b)@(out = b) \ \cong \ \hat{C_1}$$

This implies that $C_2 \xRightarrow{\nu\tilde{x}\overline{\Pi}\tilde{v}} C_2'$. Reduction barbed congruence is preserved by name restriction, so we have that $\nu a \nu b \hat{C_1} \cong \nu a \nu b \hat{C_2}$ and $C_1' \cong C_2'$ as required.

2. Suppose that $C_1 \xrightarrow{\Pi(\tilde{v})} C_1'$ for some Π and a sequence of values $\tilde{v}$. Assume $C_1 \equiv \Gamma : P_1$, we build up the following context to mimic the effect of this transition.

$$\mathcal{C}[\bullet] \triangleq [\bullet] \parallel \Gamma' : (\tilde{v})@(in = a).(\tilde{v})@(out = b)$$

where $\Gamma \models \Pi$ and $\Pi = (in = a)$, and the names a and b are fresh. As $\cong$ is context-closed, $C_1 \cong C_2$ implies $\mathcal{C}[C_1] \cong \mathcal{C}[C_2]$. Since $C_1 \xrightarrow{\Pi(\tilde{v})} C_1'$, it follows that:

$$\mathcal{C}[C_1] \twoheadrightarrow^* C_1' \parallel (\tilde{v})@(out = b) \ = \ \hat{C_1}$$

with $\hat{C_1} \not\Downarrow_{(in=a)}$ and $\hat{C_1} \Downarrow_{(out=b)}$.
The reduction sequence above must be matched by a corresponding reduction sequence $\mathcal{C}[C_2] \twoheadrightarrow^* \hat{C_2} \cong \hat{C_1}$ with $\hat{C_2} \not\Downarrow_{(in=a)}$ and

$\hat{C}_2 \Downarrow_{(out=b)}$. By Lemma 5.1, we have that:

$$\mathcal{C}[C_2] \;\rightarrowtriangle^* C_2' \parallel (\tilde{v})@(out = b) \cong \hat{C}_1$$

This implies that $C_2 \stackrel{\Pi(\tilde{v})}{\Longrightarrow} C_2'$. Reduction barbed congruence is preserved by name restriction, so we have that $\nu a \nu b \hat{C}_1 \cong \nu a \nu b \hat{C}_2$ and $C_1' \cong C_2'$ as required.

3. Suppose that $C_1 \xrightarrow{\tau} C_1'$. This case is straightforward.

□

Theorem 5.2 (Characterization)**.** *Bisimilarity and reduction barbed congruence coincide.*

Proof. As a direct consequence of Theorem 5.1 and Lemma 5.6, we have that bisimilarity and reduction barbed congruence coincide. □

5.3 Properties of the Bisimilarity Relation

We have already proved in the previous section that bisimilarity is a congruence relation with respect to all external AbC contexts (i.e., component level contexts), presented in Definition 2. In this section we want to show that because of the dependencies of processes on the attribute environment, except for the awareness operator, all process-level operators do not preserve bisimilarity. The rest of this section is concerned with other properties and equational laws, exhibited by bisimilarity. The properties also hold for strong bisimilarity unless stated otherwise.

The following remark shows that weak bisimilarity is not preserved by most process level operators.

Remark 5.1. *Let* $\Gamma : P \approx \Gamma : Q$*, then*

- $\Gamma : P\sigma \not\approx \Gamma : Q\sigma$ *for some substitution* σ
- $\Gamma : \alpha.P \not\approx \Gamma : \alpha.Q$ *for some action* α
- $\Gamma : P|R \not\approx \Gamma : Q|R$ *for some process* R
- $\Gamma : \langle \Pi \rangle P \approx \Gamma : \langle \Pi \rangle Q$ *for every predicate* Π

- $\Gamma : [\tilde{a} := \tilde{E}]P \not\approx \Gamma : [\tilde{a} := \tilde{E}]Q$ *for some update* $[\tilde{a} := \tilde{E}]$

Proof. Let $C_1 = \Gamma : \overbrace{\langle \texttt{this}.a = w \rangle (v')@\Pi.0}^{P}$ where $\Gamma(a) = v$, $C_2 = \Gamma : \overbrace{0}^{Q}$, and $R = [a := v]()@\mathsf{ff}.0$. It is easy to see that $C_1 \approx C_2$, because both components are not able to progress. Notice that $[\![\texttt{this}.a = w]\!]_\Gamma \triangleq \mathsf{ff}$.

- If we apply the substitution $[v/w]$ to both processes P and Q, we have that $\Gamma : P[v/w] \xrightarrow{\overline{\Pi} v'}$ and $\Gamma : Q[v/w] \not\xrightarrow{\overline{\Pi} v'}$ and $\Gamma : P\sigma \not\approx \Gamma : Q\sigma$ as required.
- The statement, $\Gamma : \alpha.P \not\approx \Gamma : \alpha.Q$ for some action α, is a direct consequence of the first statement. For instance, consider an input prefix of the following form $(\mathsf{tt})(w)$.
- The statement, $\Gamma : P|R \not\approx \Gamma : Q|R$ for some process R, holds easily from our example when we put the process R in parallel of the processes P and Q.
- The statement, $\Gamma : \langle \Pi \rangle P \approx \Gamma : \langle \Pi \rangle Q$ for every predicate Π, is a direct sequence of operational rules for the awareness operator.
- The statement, $\Gamma : [\tilde{a} := \tilde{E}]P \not\approx \Gamma : [\tilde{a} := \tilde{E}]Q$ for some update $[\tilde{a} := \tilde{E}]$, holds easily with the following update $[a := w]$.

□

It should be noted that if we close bisimilarity under substitutions by definition, all of the statements in Remark 5.1 follow directly. The definition would be a slight variant of the notion of full bisimilarity proposed by Sangiorgi and Walker in (SW03). In this way, the components C_1 and C_2 in the proof above are no longer bisimilar since they are not closed under the substitution $[v/w]$. However, the notion of full bisimilarity is more finer than the notion of bisimilarity proposed in this thesis.

The following remark shows that, as expected, the non-deterministic choice does not preserve bisimilarity. The reason is related to the fact that input transitions cannot be observed. Below we explain the issue with a concrete example.

Remark 5.2. $\Gamma : P \approx \Gamma : Q$ *does not imply* $\Gamma : P + R \approx \Gamma : Q + R$ *for every process* R

Proof. Let $C_1 = \Gamma : \Pi_1(x).\mathsf{0}$, $C_2 = \Gamma : \Pi_2(x).\mathsf{0}$, and $R = (v)@\Pi.\mathsf{0}$. Though the receiving predicates for both components are different we still have that $C_1 \approx C_2$ and this is because that input actions are not perceived. When a message $\overline{\Pi_3}w$ arrives, where $\Gamma \models \Pi_3$, $[\![\Pi_1[w/x]]\!]_\Gamma \doteq \mathsf{tt}$ and $[\![\Pi_2[w/x]]\!]_\Gamma \doteq \mathsf{ff}$, component C_1 applies rule (**Comp**) and evolves to $\Gamma : \mathsf{0}$ while component C_2 applies rule (**C-Fail**) and stays unchanged. Both transitions carry the same label and again $\Gamma : \mathsf{0}$ and $\Gamma : \Pi_2(x).\mathsf{0}$ are equivalent for the same reason. An external observer cannot distinguish them.

Now if we allow mixed choice within a single component, then one can distinguish between $\Pi_1(x)$ and $\Pi_2(x)$.

$$\Gamma : \Pi_1(x).\mathsf{0} + R \not\approx \Gamma : \Pi_2(x).\mathsf{0} + R$$

Assume that the message $\overline{\Pi_3}w$ is arrived, we have that:

$$\Gamma : \Pi_1(x).\mathsf{0} + R \xrightarrow{\Pi_3(w)} \Gamma : \mathsf{0} \stackrel{\overline{\Pi v}}{\not\longrightarrow}$$

while

$$\Gamma : \Pi_2(x).\mathsf{0} + R \xrightarrow{\Pi_3(w)} \Gamma : \Pi_2(x).\mathsf{0} + R \xrightarrow{\overline{\Pi v}} \Gamma : \mathsf{0}$$

However, this is obvious since our relation is defined at the component-level. So it abstracts from internal behavior and characterizes the behavior of AbC systems from an external observer point of view. In practice this is not a problem since mixed choice is very hard to be implemented. □

The following lemmas prove useful properties about AbC operators (i.e., parallel composition is commutative, associative, ...).

Lemma 5.7 (Parallel composition)**.**

- $C_1 \| C_2 \approx C_2 \| C_1$
- $(C_1 \| C_2) \| C_3 \approx C_1 \| (C_2 \| C_3)$
- $\Gamma : \mathsf{0} \parallel C \approx C$

Proof. Directly from the operational semantics of AbC, Chapter 4 and from the definition of bisimilarity in Definition 8. □

Lemma 5.8 (Name restriction)**.**

- $\nu x C \approx C$ *if* $x \notin fn(C)$
- $\nu x \nu y C \approx \nu y \nu x C$ *if* $x \neq y$
- $\nu x C_1 \parallel C_2 \approx \nu x (C_1 \parallel C_2)$ *if* $x \notin fn(C_2)$

Proof. We only prove the last statement and the other statements are straightforward. To prove that $\nu x C_1 \parallel C_2 \approx \nu x (C_1 \parallel C_2)$ if $x \notin fn(C_2)$, it is sufficient to prove that the relation $\mathcal{R} = \{(\nu x C_1 \parallel C_2, \nu x (C_1 \parallel C_2)) \mid x \notin fn(C_2) \text{ and } \nu x C \approx C\}$ is a weak bisimulation. We do a case analysis on the transition $\nu x C_1 \parallel C_2 \xrightarrow{\gamma} \hat{C}$.

We omit the trivial cases when C_2 takes a step.

- Case ($\gamma = \nu x \overline{\Pi} \tilde{v}$) where $x \in \tilde{v}$: We can only apply rule (**Com**) and we have that $\nu x C_1 \parallel C_2 \xrightarrow{\nu x \overline{\Pi} \tilde{v}} C_1' \parallel C_2' = \hat{C}'$. On the other hand $\nu x (C_1 \parallel C_2)$ evolves to $C_1' \parallel C_2'$ by Lemma 5.2, an application of (Open), and several applications of (Res) and we have that $C_1' \parallel C_2' \approx \hat{C}'$.
- Case ($\gamma = \nu y \overline{\Pi} \tilde{v}$) where $x \neq y \wedge x \notin \tilde{v}$: We can only apply rule (**Com**) and we have that $\nu x C_1 \parallel C_2 \xrightarrow{\nu y \overline{\Pi} \tilde{v}} \nu z C_1' \parallel C_2'$ if $\nu x C_1 \xrightarrow{\nu y \overline{\Pi} \tilde{v}} \nu z C_1'$. On the other hand $\nu x (C_1 \parallel C_2)$ evolves to $\nu z (C_1' \parallel C_2')$ by Lemma 5.2 and several applications of (Res).
- Case ($\gamma = \Pi(\tilde{v})$): Again with rule (**Com**), we have that $\nu x C_1 \| C_2 \xrightarrow{\Pi(\tilde{v})} \nu y C_1' \parallel C_2'$, while $\nu x (C_1 \parallel C_2)$ evolves to $\nu y (C_1' \parallel C_2')$ by Lemma 5.2 and several applications of (Res).
- Case ($\gamma = \tau$): $\nu x C_1 \parallel C_2$ can only apply rule (τ-**Int**) and evolves to $\nu x \nu \tilde{y} C_1' \parallel C_2$ if $\nu x C_1 \xrightarrow{\nu \tilde{y} \overline{\Pi \blacktriangleright x} \tilde{v}} \nu x \nu \tilde{y} C_1'$ where $\Pi \blacktriangleright x \doteq \mathsf{ff}$. On the other hand $\nu x (C_1 \parallel C_2)$ evolves to $\nu x \nu \tilde{y} (C_1' \parallel C_2)$ by Lemma 5.2, an application of (Hide1), and several applications of (Res), since $C_1 \xrightarrow{\nu \tilde{y} \overline{\Pi} \tilde{v}} C_1'$ where $\Pi \blacktriangleright x \doteq \mathsf{ff}$. Notice that $\nu \tilde{y} \overline{\mathsf{ff}} \tilde{v}$ is equivalent to τ (i.e., sending on a false predicate).
- Case ($\gamma = \nu \tilde{y} \overline{\Pi \blacktriangleright x} \tilde{v}$) where where $\Pi \blacktriangleright x \neq \mathsf{ff}$ and $x \in n(\Pi)$: $\nu x C_1 \parallel C_2$ can only apply rule (**Com**) and evolves to $\nu x C_1' \parallel C_2'$ if $\nu x C_1 \xrightarrow{\nu \tilde{y} \overline{\Pi \blacktriangleright x} \tilde{v}} \nu x C_1'$. On the other hand $\nu x (C_1 \parallel C_2)$ evolves to

$\nu x(C_1' \parallel C_2')$ by Lemma 5.2, an application of (Hide2), and several applications of (Res), since $C_1 \xrightarrow{\nu\tilde{y}\overline{\Pi}\tilde{v}} C_1'$.

By induction hypotheses we have that $(\nu x C_1 \parallel C_2, \nu x(C_1 \parallel C_2)) \in \mathcal{R}$ as required.

□

Lemma 5.9 (Non-deterministic choice)**.**

- $\Gamma : P_1 + P_2 \approx \Gamma : P_2 + P_1$
- $\Gamma : (P_1 + P_2) + P_3 \approx \Gamma : P_1 + (P_2 + P_3)$
- $\Gamma : P + \mathsf{0} \approx \Gamma : P$
- $\Gamma : P + P \approx \Gamma : P$
- $\Gamma : \langle \Pi \rangle (P + Q) \approx \Gamma : \langle \Pi \rangle P + \langle \Pi \rangle Q$
- $\Gamma : [\tilde{a} := \tilde{E}](P + Q) \approx \Gamma : [\tilde{a} := \tilde{E}]P + [\tilde{a} := \tilde{E}]Q$

Proof. Directly from the operational semantics of AbC, Chapter 4 and from the definition of bisimilarity in Definition 8. □

Lemma 5.10 (Interleaving)**.**

- $\Gamma : P_1 | P_2 \approx \Gamma : P_2 | P_1$
- $\Gamma : (P_1 | P_2) | P_3 \approx \Gamma : P_1 | (P_2 | P_3)$
- $\Gamma : P | \mathsf{0} \approx \Gamma : P$

Proof. Directly from the operational semantics of AbC, Chapter 4 and from the definition of bisimilarity in Definition 8. □

Lemma 5.11. $\Gamma_1 : P \approx \Gamma_2 : [\tilde{a} := \tilde{E}]P$ *if and only if* $\Gamma_2[\tilde{a} \mapsto [\![\tilde{E}]\!]_{\Gamma_2}] = \Gamma_1$.

Proof. Directly from the operational semantics of AbC, Chapter 4 and from the definition of bisimilarity in Definition 8. □

Lemma 5.12 (Awareness)**.**

- $\Gamma : \langle \mathsf{ff} \rangle P \approx \Gamma : \mathsf{0}$
- $\Gamma : \langle \mathsf{tt} \rangle P \approx \Gamma : P$

- $\Gamma : \langle \Pi_1 \rangle \langle \Pi_2 \rangle P \approx \Gamma : \langle \Pi_1 \wedge \Pi_2 \rangle P$

Proof. Directly from the operational semantics of AbC, Chapter 4 and from the definition of bisimilarity in Definition 8. □

Lemma 5.13 (Silent components cannot be observed)**.** *Let $Act(P)$ denote the set of actions in process P. If $Act(P)$ does not contain any output action, then:*

$$\Gamma : P \approx \Gamma : \mathsf{0}$$

Proof. The proof follows from the fact that components with no external side-effects (i.e., do not exhibit barbs) cannot be observed. When $Act(P)$ does not contain output actions, component $\Gamma : P$ can either make silent moves, which component $\Gamma : \mathsf{0}$ can mimic by simply doing nothing, or input a message, which component $\Gamma : \mathsf{0}$ can mimic by discarding the message. □

Now we proceed with a set of examples to show interesting observations about the AbC calculus.

Example 5.1. *Let $C_1 = \Gamma : \Pi(x).P$ and $C_2 = \Gamma : \Pi_1(x).P + \Pi_2(x).P$ where $\Pi \triangleq \Pi_1 \vee \Pi_2$, it holds that:*

$$C_1 \approx C_2$$

Clearly, components C_1 and C_2 are bisimilar because any message, accepted by C_2, can also be accepted by C_1 and vice versa. After a successful input both components proceed with the same continuation process $P[v/x]$. For instance the message $\overline{\Pi_1} v$ which is satisfied by only predicate Π_2 (i.e., $[\![\Pi_2[v/x]]\!]_\Gamma \triangleq \mathsf{tt}$) is still satisfied by predicate Π. The overlapping between the input and the non-deterministic choice constructs is clear in this scenario. For this special case we can replace the non-deterministic choice with an "or" predicate while preserving the observable behavior. The intuition is illustrated in Figure 5.3.

Corollary 5.1. *Let C_1 and C_2 be two components where $C_1 = \Gamma : \Pi_1(\tilde{x}).P + \cdots + \Pi_n(\tilde{x}).P$ and $C_2 = \Gamma : (\Pi_1 \vee \Pi_2 \vee \cdots \vee \Pi_n)(\tilde{x}).P$ we have that $C_1 \approx C_2$.*

Example 5.2. *$\Gamma_1 : (E_1)@\Pi.P \approx \Gamma_2 : (E_2)@\Pi.P$ if and only if $[\![E_1]\!]_{\Gamma_2} = [\![E_2]\!]_{\Gamma_1}$ and $Act(P)$ does not contain input actions.*

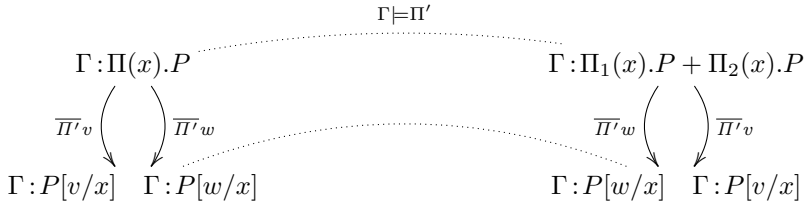

Figure 4: The relationship between the "or" predicate and the non-deterministic choice

It is clear that even if $\Gamma_1 \neq \Gamma_2$*, these components are still bisimilar since the exposed attribute values (i.e.,* $[\![E_1]\!]_{\Gamma_2}$*) are the same. The intuition is that sending components can control what attribute values to be exposed to the communication partners. In some sense the sending component has the power of selecting the criteria in which its communicated messages can be filtered. If the continuation process* P *contains at least one input action, the property cease to hold. The reason is that any incoming message with a predicate, satisfied by* Γ_1 *but not* Γ_2 *can tell them apart.*

Now we show some interesting properties about name restriction in AbC. The next example is simple and the intuition behind it will be used later in a more involved scenario.

Example 5.3. *Let* $C_1 = \Gamma : (v)@\Pi_1.P$ *and* $C_2 = \Gamma : (v)@\Pi.P$ *where* $\Pi \doteq \Pi_1 \vee \Pi_2$*, it holds that:*

$$\nu x C_1 \approx \nu x C_2 \quad \textit{if and only if} \quad \Pi_2 \blacktriangleright x = \mathsf{ff}$$

Clearly $\nu x C_1$ *can apply rule (**Res**) and evolves to* $C_1' = \nu x \Gamma : P$ *with a transition label* $\overline{\Pi_1} v$ *while* $\nu x C_2$ *can apply rule (**Hide2**) and evolves* $C_2' = \nu x \Gamma : P \approx C_1'$ *with a transition label* $\overline{\Pi \blacktriangleright x} v$*. From Table 7, Chapter 4 we have that* $(\Pi \blacktriangleright x) = (\mathsf{ff} \vee \Pi_1) = \Pi_1$*. Now it is easy to see that components* $\nu x C_1$ *and* $\nu x C_2$ *are bisimilar. The hiding mechanism in* AbC *where a predicate can be partially exposed is very useful in describing collective behavior with a global point of view. In the next example we show the expressive power of name restriction in a more involved scenario.*

Example 5.4. *We consider two types of components, a provider component* $CP = \Gamma_p : P$ *and a forwarder component* $CF = \Gamma_i : F$ *where the*

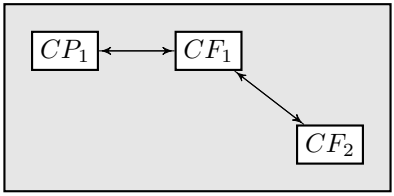

Figure 5: The system with assumptions about the network topology

behavior of processes P and F is defined below.

$$P \triangleq \ (\texttt{this}.role,\ \tilde{v})@(\Pi_1 \vee (role = fwd)).0$$

$$F \triangleq (x = pdr)(x,\ \tilde{y}).(\texttt{this}.grp,\ x,\ \tilde{y})@(role = fwd).(x,\ \tilde{y})@\Pi_1.0$$
$$+$$
$$(x = \texttt{this}.grp \vee x = \texttt{this}.nbr)(x,\ y,\ \tilde{z}).(y,\ \tilde{z})@\Pi_1.0$$

Process P sends an advertisement message to all components that either satisfy predicate Π_1 where $\Pi_1 = (role = client)$ or have a forwarder role (i.e., $(role = fwd)$). Process F may receive an ad from a provider, then it first appends its group id (i.e., $\texttt{this}.grp$) to the message and sends it to nearby forwarders. Process F continues by sending the ad to nearby clients. Alternatively process F may receive a message from one member of its group (i.e., the forwarder that shares the same group id) or from a neighbor forwarder from another group ($x = \texttt{this}.nbr$) and then it will propagate the message to nearby clients. The scenario is simplified to allow at most two hops from the provider. The communication links between providers and forwarders are private (i.e., the name "fwd" and all group and neighbor ids (i.e., $\tilde{n}$) are private names) to avoid interference with other components running in parallel.

The goal of the provider component is to ensure that its advertising message reaches all clients across the network.

To prove if the above specification guarantees this property[1]*, we first need to fix the topology of the network as reported in Figure 5. For the sake of simplicity we will only consider a network of one provider $CP_1 = \Gamma_p : P$ and two forwarders $CF_1 = \Gamma_1 : F$ and $CF_2 = \Gamma_2 : F$. We assume short-range communication where CP_1 messages can reach to CF_1 and CF_2 can*

[1]The results in this scenario holds only for weak bisimulation.

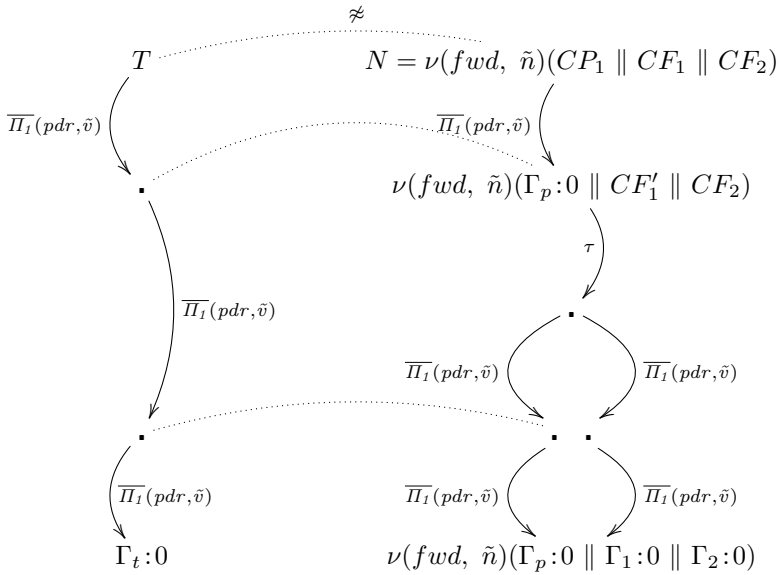

Figure 6: System N simulates the test component T, but initial interference is possible, Hence $N \not\approx T$

only receive the messages when CF_1 forwards them. Assume that initially the attribute environments Γ_p, Γ_1 and Γ_2 are defined as follows:

$$\Gamma_p = \{(grp, n),\ (role, pdr)\}, \quad \Gamma_1 = \{(grp, n),\ (role, fwd),\ (nbr, n')\}$$
$$\Gamma_2 = \{(grp, n),\ (role, fwd),\ (nbr, n'')\}$$

The full system is represented by the component N as defined below:

$$N = \nu(fwd,\ \tilde{n})(CP_1 \parallel CF_1 \parallel CF_2)$$

*The behavior of N without any interventions from other providers is reported on the right side of Figure 6. The provider component CP_1 initiates the interaction by sending an advertisement to nearby clients and forwarders and evolves to $\Gamma_p : \mathsf{0}$. Forwarder CF_1 receives the message and evolves to CF_1'. The overall system N applies rule (**Hide2**) and evolves to $\nu(fwd,\ \tilde{n})(\Gamma_p : \mathsf{0} \parallel CF_1' \parallel CF_2)$ with the label $\overline{(\Pi_1 \vee (role = fwd)) \blacktriangleright fwd}(pdr, \tilde{v})$ which is equivalent to $\overline{\Pi_1}(pdr, \tilde{v})$ according to Table 7, Chapter 4. The forwarder CF_1' adds its group id to the message and sends it secretly to nearby forwarders, in our case this is CF_2. The overall system applies rule (**Hide1**) and evolves to $\nu(fwd,\ \tilde{n})(\Gamma_p : \mathsf{0} \parallel CF_1'' \parallel CF_2')$ with the label $\nu n_1 \overline{((role = fwd)) \blacktriangleright fwd}(n_1, pdr, \tilde{v})$ which is equivalent to $\nu n_1 \overline{\mathsf{ff}}(n_1, pdr, \tilde{v})$. This message is private and is perceived externally as a τ-move. The overall system terminates after emitting the ad, $\overline{\Pi_1}(pdr, \tilde{v})$, two more times, one from CF_1'' and the other from CF_2'. By applying the rule (**Res**) twice, the system evolves to $\nu(fwd,\ \tilde{n})(\Gamma_p : \mathsf{0} \parallel \Gamma_1 : \mathsf{0} \parallel \Gamma_2 : \mathsf{0})$.*

To prove that the advertising message is propagated to all clients in the network it is sufficient to show that each forwarder takes its turn in spreading the message. Formally it is sufficient to prove that the behavior of the overall system is bisimilar to the behavior of a test component T, defined below, which is able to send the same message three times sequentially and then terminates.

$$T = \Gamma_t : (pdr,\ \tilde{v})@\Pi_1.(pdr,\ \tilde{v})@\Pi_1.(pdr,\ \tilde{v})@\Pi_1.0$$

Figure 6 shows that system N weakly simulates component T, but they are not bisimilar, i.e., $T \not\approx N$. This is because forwarders are initially prepared to accept messages from any component with a provider role. For instance if we put another provider, say $CP_2 = \Gamma_h : (\texttt{this}.role,\ \tilde{w})@(\mathsf{tt}).0$ where $\Gamma_h(role) = pdr$, there is a possibility that CF_1 first receives a message from CP_2 and the system evolves as follows:

$$N \| CP_2 \xrightarrow{\overline{\mathsf{tt}}(pdr, \tilde{w})} \xrightarrow{\overline{\Pi_1}(pdr, \tilde{v})} \xrightarrow{\overline{\Pi_1}(pdr, \tilde{w})} \xrightarrow{\overline{\Pi_1}(pdr, \tilde{w})}$$

while

$$T \| CP_2 \xrightarrow{\overline{\Pi_1}(pdr,\tilde{v})} \xrightarrow{\overline{\mathrm{tt}}(pdr,\tilde{w})} \xrightarrow{\overline{\Pi_1}(pdr,\tilde{v})} \xrightarrow{\overline{\Pi_1}(pdr,\tilde{v})}$$

and it is easy to see that $N \| CP_2 \not\approx T \| CP_2$. *One way to avoid interference and ensure that the property holds is shown below:*

$$P' \triangleq (\mathit{this}.grp,\ \mathit{this}.role,\ \tilde{v})@(role = fwd).(\mathit{this}.role,\ \tilde{v})@\Pi_1.0$$

$$F' \triangleq (x = \mathit{this}.grp \wedge y = pdr)(x,\ y,\ \tilde{z}).(\mathit{this}.grp,\ y,\ \tilde{z})@(role = fwd).(y,\ \tilde{z})@\Pi_1.0$$
$$+$$
$$(x = \mathit{this}.grp \vee x = \mathit{this}.nbr)(x,\ y,\ \tilde{z}).(y,\ \tilde{z})@\Pi_1.0$$

Now for components $\hat{CP}_1 = \Gamma_p : P'$, $\hat{CF}_1 = \Gamma_1 : F'$, $\hat{CF}_2 = \Gamma_2 : F'$, *and system* $\hat{N}$ *where* $\hat{N} = \nu(fwd,\ \tilde{n})(\hat{CP}_1 \parallel \hat{CF}_1 \parallel \hat{CF}_2)$ *we have that* $T \approx \hat{N}$. *The interference is avoided by isolating process* F' *from the external world and now it can only receive messages from its group members with a provider role, in our case this is* $\hat{CP}_1$. *To allow* $\hat{CP}_1$ *and* $\hat{CF}_1$ *to interact, process* P' *is adapted so that it first sends a secret message to its group and then continues by sending a public message to nearby clients.*

5.4 Correctness of the encoding

In this section, we provide a proof sketch for the correctness of the encoding presented in Section 3.2.1. We begin by listing the properties that we would like our encoding to preserve. Basically, when translating a term from $b\pi$-calculus into AbC, we would like the translation: to be compositional by being independent from contexts; to be independent from the names of the source term (i.e., name invariance); to preserve parallel composition (i.e., homomorphic w.r.t. '|'); to be faithful in the sense of preserving the observable behavior (i.e., barbs) and reflecting divergence; to translate output (input) action in $b\pi$-calculus into a corresponding output (input) in AbC, and to preserve the operational correspondence between the source and target calculus. This includes that the translation should be complete (i.e., every computation of the source term can be mimicked by its translation) and it should be sound (i.e., every computation of a translated term corresponds to some computation of its source term).

Definition 9 (Divergence)**.** *P diverges, written $P \Uparrow$, iff $P \rightarrowtriangle^{\omega}$ where ω denotes an infinite number of reductions.*

Definition 10 (Uniform Encoding)**.** *An encoding $(\!|\ \mathbf{.}\ |\!) : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ is uniform if it enjoys the following properties:*

1. *(Homomorphic w.r.t. parallel composition): $(\!|\ P \| Q\ |\!) \triangleq (\!|\ P\ |\!) \| (\!|\ Q\ |\!)$*
2. *(Name invariance): $(\!|\ P\sigma\ |\!) \triangleq (\!|\ P\ |\!)\sigma$, for any permutation of names σ.*
3. *(Faithfulness): $P \Downarrow_1$ iff $(\!|\ P\ |\!) \Downarrow_2$; $P \Uparrow_1$ iff $(\!|\ P\ |\!) \Uparrow_2$*
4. *Operational correspondence*
    1. *(Operational completeness): if $P \rightarrowtriangle_1 P'$ then $(\!|\ P\ |\!) \rightarrowtriangle^*_2 \simeq_2 (\!|\ P'\ |\!)$ where $\simeq$ is the strong barbed equivalence of $\mathcal{L}_2$.*
    2. *(Operational soundness): if $(\!|\ P\ |\!) \rightarrowtriangle_2 Q$ then there exists a P' such that $P \rightarrowtriangle^*_1 P'$ and $Q \rightarrowtriangle^*_2 \simeq_2 (\!|\ P'\ |\!)$, where $\simeq$ is the strong barbed equivalence of $\mathcal{L}_2$.*

Lemma 5.14 (Operational Completeness)**.** *if $P \rightarrowtriangle_{b\pi} P'$ then $(\!|P|\!)_c \rightarrowtriangle^* \simeq (\!|P'|\!)_c$.*

Now we provide a proof sketch for the operational completeness and we leave the full proof details in the Appendix A.1.

Proof. (Sketch) The proof proceeds by induction on the shortest transition of $\rightarrowtriangle_{b\pi}$. We have several cases depending on the structure of the term P. We only consider the case of parallel composition when communication happens: $P_1 \| P_2 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{z}} P_1' \| P_2'$. By applying induction hypotheses on the premises $P_1 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{z}} P_1'$ and $P_2 \xrightarrow{a(\tilde{z})} P_2'$, we have that $(\!|\ P_1\ |\!)_c \rightarrowtriangle^* \simeq (\!|\ P_1'\ |\!)_c$ and $(\!|\ P_2\ |\!)_c \rightarrowtriangle^* \simeq (\!|\ P_2'\ |\!)_c$. We can apply rule (Com).

$$\frac{\emptyset : (\!|P_1|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{z})} \emptyset : (\!|P_1'|\!)_p \qquad \emptyset : (\!|P_2'|\!)_p \xrightarrow{(a=a)(a,\tilde{z})} \emptyset : (\!|P_2'|\!)_p}{\emptyset : (\!|P_1|\!)_p \parallel \emptyset : (\!|P_2|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{z})} \emptyset : (\!|P_1'|\!)_p \parallel \emptyset : (\!|P_2'|\!)_p}$$

Now, it is easy to see that: $(\!|P_1' \| P_2'|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2'|\!)_p$. Notice that the $b\pi$ term and its encoding have the same observable behavior i.e., $P_1 \| P_2 \downarrow_a$ and $(\!|P_1 \| P_2|\!)_c \downarrow_{(a=a)}$. □

Lemma 5.15 (Operational Soundness). *if* $(\!|P|\!)_c \twoheadrightarrow Q$ *then* $\exists P'$ *such that* $P \twoheadrightarrow^*_{b\pi} P'$ *and* $Q \twoheadrightarrow^* \simeq (\!|P'|\!)_c$.

Proof. The proof holds immediately due to the fact that every encoded $b\pi$-term (i.e., $(\!|\ P\ |\!)_\emptyset$) has exactly one possible transition which matches the original $b\pi$-term (i.e., P). □

The idea that we can mimic each transition of $b\pi$-calculus by exactly one transition in AbC implies that soundness and completeness of the operational correspondence can be even proved in a stronger way as in corollary 1 and 2.

Corollary 5.2 (Strong Completeness). *if* $P \twoheadrightarrow_{b\pi} P'$ *then* $\exists Q$ *such that* $Q \equiv (\!|P'|\!)_c$ *and* $(\!|P|\!)_c \twoheadrightarrow Q$.

Corollary 5.3 (Strong Soundness). *if* $(\!|P|\!)_c \twoheadrightarrow Q$ *then* $Q \equiv (\!|P'|\!)_c$ *and* $P \twoheadrightarrow_{b\pi} P'$

Theorem 5.3. *The encoding* $(\!|\ .\ |\!) : b\pi \to \mathrm{AbC}$ *is uniform.*

Proof. Definition 10(1) and 10(2) hold by construction. Definition 10(4) holds by Lemma 5.14, Lemma 5.15, Corollary 5.2, and Corollary 5.3 respectively. Definition 10(3) holds easily and as a result of the proof of Lemma 5.14 and the strong formulation of operational correspondence in Corollary 5.2, and Corollary 5.3, this encoding preserves the observable behavior and cannot introduce divergence. □

As a result of Theorem 5.2, Theorem 5.3 and of the strong formulations of Corollary 5.2, and Corollary 5.3, this encoding is sound and complete with respect to bisimilarity as stated in the following corollaries.

Corollary 5.4 (Soundness w.r.t bisimilarity).

- $(\!|P|\!)_c \sim (\!|Q|\!)_c$ *implies* $P \sim Q$
- $(\!|P|\!)_c \approx (\!|Q|\!)_c$ *implies* $P \approx Q$

Corollary 5.5 (Completeness w.r.t bisimilarity).

- $P \sim Q$ *implies* $(\!|P|\!)_c \sim (\!|Q|\!)_c$
- $P \approx Q$ *implies* $(\!|P|\!)_c \approx (\!|Q|\!)_c$

Chapter 6

$Ab^{a}C$uS: A Run-time Environment for the *AbC* Calculus

We present $Ab^{a}C$uS[1], a Java run-time environment for supporting the communication primitives of the *AbC* calculus. Having a run-time environment allows us to assess the practical impact of this young communication paradigm in real applications. In fact, we plan to use the new programming framework to program challenging case studies, dealing with collective adaptive systems, from different application domains.

$Ab^{a}C$uS provides a Java API that allows programmers to use the linguistic primitives of the *AbC* calculus in Java programs. The implementation of $Ab^{a}C$uS fully relies on the formal semantics of the *AbC* calculus. There is a one-to-one correspondence between the *AbC* primitives and the programming constructs in $Ab^{a}C$uS. This close correspondence enhances the confidence on the behavior of $Ab^{a}C$uS programs after they have been analyzed via formal methods, which is made possible by the fact that we rely on the operational semantics of the *AbC* calculus. To facilitate interoperability with other tools and programming frameworks, $Ab^{a}C$uS relies

[1] `http://lazkany.github.io/AbC/`

on JSON[2], a standard data exchange technology that simplifies the interactions between heterogenous network components and provides the basis for allowing $Ab^{a}C$uS programs to cooperate with external services or devices.

The operational semantics of the *AbC* calculus abstracts from a specific communication infrastructure. An *AbC* model consists of a set of parallel components that cooperate in a highly dynamic environment where the underlying communication infrastructure can change dynamically. We start by showing the obvious rendering of *AbC* models as $Ab^{a}C$uS programs and then we show how the underlying communication infrastructure can be implemented to facilitate the interaction between different components. We lay the basis for an efficient implementation of a communication infrastructure that respects that semantics of the *AbC* calculus and we rely on stochastic simulation to evaluate the performance.

The advantages of our programming framework can be summarized by saying that it provides a small set of programming constructs that naturally supports collective-adaptive features including scalability, awareness, adaptation and interdependence, and also permits collaboration to achieve system goals.

6.1 From *AbC* primitives to $Ab^{a}C$uS programming constructs

In this section we show the rendering of the *AbC* primitives in terms of the programming constructs of the $Ab^{a}C$uS framework. These constructs can be viewed as Java macros that mimic the behavior of the *AbC* primitives. We use the running example (Section 2.2.1), and more specifically the description of the participant component, to illustrate the rendering.

Components *AbC* components are implemented via class `AbCComponent`. Instances of this class are executed on either virtual or physical machines that provide access to input/output devices and network connections. An instance of the class `AbCComponent` contains an attribute

[2] `http://www.json.org`

environment and a set of processes that represents the behavior of the component. Components interact via ports supporting either local communication, i.e., components that run on the same machine, or external communication, i.e., components that run on different machines.

The following *Ab*^{a}*C*uS code shows how to create a *participant* component `p`, to set the values of its attributes, to assign the process "`Participant(topic)`" to it, and finally to start its execution.

```
1    AbCComponent p = new AbCComponent("Participant-1");
2    p.setValue(Defs.id, 1);
3    p.addProcess(new Participant(topic));
4    p.start();
```

Notice that the programer is not required to construct the attribute environment for each component from scratch. He is only required to initialize the values of the component attributes via method "`setValue(arg1, arg2)`" as shown at (line 2). This method takes two arguments, "`arg1`", which specifies the type of the attribute, and "`arg2`", which specifies its assigned value. The method checks if the attribute identifier already exists in the attribute environment and assigns the new value to it. Otherwise a new identifier is added to the attribute environment and assigned the initial value. It is a good programming practice to define all constants, attribute identifiers, and other data structures in a single Java class; in our case we use "`Defs`", and then one can refer to any of them from *Ab*^{a}*C*uS programs. This will help creating more clean, compact, and less verbose *Ab*^{a}*C*uS programs.

Attribute Environments *AbC* attribute environments are implemented via the class `AbCEnvironment`. An instance of the class `AbCEnvironment` contains a set of attribute identifiers, implemented via class `AbCAttribute`, and their corresponding values. The attribute environment maintains the attribute values by providing read and update operations via the methods `getValue(attribute)` and `setValue(attribute,value)` respectively. The class `AbCAttribute` implements an *AbC* attribute and provides a mechanism to ensure type compatibility of the possible assigned values. The following *Ab*^{a}*C*uS code shows how to create an *AbC* attribute

and to read and update its value respectively.

```
1       AbCAttribute<Integer> id = new AbCAttribute<>("ID", Integer.class);
2       getValue(id);
3       setValue(id, 1);
```

Processes The generic behavior of an *AbC* process is implemented via the abstract class `AbCProcess`. To create an *AbC* process you have to extend the class `AbCProcess` as shown in the process template, Program 6.1 below. The text between square brackets represents the editable part of the template while the code preceeded by the operator "?" represents the optional part of the template. For instance the text "`[PROCESS-IDENTIFIER]`" is replaced by a process identifier, in our case it is "`ParticipantAgent`". The set of instance variables and the constructor of any process and its parameters are optional unless the process is parametrized and needs to initialize some variables. In our example process `ParticipantAgent` in Specification 2.1 is parametrized with respect to the topic of interest, that is why we defined an instance variable `topic` and initialized it in the constructor of the process `ParticipantAgent` as shown in the $Ab^{a}C$uS implementation in Program 6.2 (Lines 3-8). In Program 6.2 we only report the translation of Specification 2.1 into $Ab^{a}C$uS code. The full translation of the smart conference system can be found in Appendix A.2.

Program 6.1: Process Template

```
1       public class [PROCESS-IDENTIFIER] extends AbCProcess {
2
3        ? [INSTANCE-VARIABLES]
4        ? public [PROCESS-IDENTIFIER](?[PARAMETERS]) {
5
6            [INITIALIZATION]
7         }
8         @Override
9         protected void doRun() throws Exception {
10
11          [PROCESS-BODY]
12        }
13      }
```

The actual behavior of an AbC process is specified by implementing the abstract method "`doRun()`" (lines 8-12). This method is automatically invoked when the process starts executing. The text "`[PROCESS-BODY]`" is replaced with a sequential AbC process. In what follows we show the translation of AbC process constructs into $Ab\overset{a}{C}$uS constructs.

- **Action prefixing,** $\alpha.P$: the sequentialization of the dot operator is naturally translated into the semicolon Java operator which marks the ending of a Java command/statement.

- **Attribute update,** $[a := E]$: The attribute update is implemented via the method "`setValue(Attribute<?>, Object)`".

- **Awareness,** $\langle\Pi\rangle$: The awareness construct "$\langle\Pi\rangle$" is implemented via the method "`waitUntil(AbCPredicate)`".

- **Non-determinism,** $P_1 + P_2$: We only allow for a non-deterministic choice between processes enabling input actions, possibly preceded by attribute updates and/or awareness operators. This is implemented via the method "`receive(`$in_1, \ldots, in_n$`)`". This method takes any finite number of arguments of type `InputAction`, each of which represents a process of the following form $[\tilde{a} := \tilde{E}]\langle\Pi'\rangle\Pi(\tilde{x})$. An instance of the class `InputAction` contains a local attribute environment that represent possible updates (i.e., $[\tilde{a} := \tilde{E}]$), implemented via class `AbCEnvironment`, and two predicates, implemented via class `AbCPredicate`: Π' (the awareness predicate) and Π (the receiving predicate on the selected branch of the choice). When a message arrives to the component, this method only enables the correct branch or blocks the execution in case of unwanted messages. Non-deterministic choice between an input and output or between several output actions is not allowed because output actions are non-blocking.

- **Interleaving,** $P_1|P_2$: This construct is implemented via method "`addProcess(AbCProcess)`". This method is invoked by a component. Adding a process to a component with this method has

the same effect of running that process in parallel with the existing processes in that component.

- **Process call,** K: This is implemented via the method `call(AbCProcess)` which takes an AbC process as a parameter.

- **Recursive call, e.g.,** $P \triangleq \alpha.P$: This construct is naturally translated into a Java "`while(true){}`" loop.

- **Process replication**: Though this is not an explicit AbC process construct, its effect can be obtained when we have a parallel composition in the context of a recursive process call. For instance, process "$P \triangleq \Pi(\tilde{x}).Q|P$" replicates itself every time it receives a new message. The method "`exec(AbCProcess)`" is used to create a new AbC process and run it in parallel. Generally we only allow guarded replication, enabled by either an input action or an awareness construct. This restriction makes it simpler to implement replication, since it becomes clear when one needs to create a new copy of the replicated process. For instance, the process, $P \triangleq a.Q|P$ where a is an output action, is hard to implement because the semantics of output actions is non-blocking, so it would quickly fill up the heap with copies of P.

Actions The AbC communication actions send, $(\tilde{E})@\Pi$, and receive, $\Pi(\tilde{x})$, are implemented via the methods `Send(Predicate, Object)` and `receive(msg->Function(msg))` respectively. The receive operation accepts a message and passes it to a boolean function that checks if it satisfies the receiving predicate.

Program 6.2: The ParticipantAgent in $Ab\overset{a}{C}$uS

```
1  public class ParticipantAgent extends AbCProcess {
2
3   private String topic;
4
5   public ParticipantAgent( String name , String topic ) {
6    super( name );
7    this.topic = topic;
8   }
```

```
9
10  @Override
11  protected void doRun() throws Exception {
12   setValue(Defs.interest, this.topic);
13   send(
14    new HasValue(
15      Defs.ROLE,
16      Defs.PROVIDER
17    ) ,
18    new Tuple(
19      getValue(Defs.interest) ,
20      Defs.REQUEST ,
21      getValue(Defs.id)
22
23    )
24   );
25   Tuple value = (Tuple) receive( o -> isAnInterestReply( o ) );
26   setValue(Defs.destination, (String) value.get(2));
27   while (true) {
28    value = (Tuple) receive( o -> isAnInterestUpdate( o ) );
29    setValue(Defs.destination, (String) value.get(3));
30   }
31  }
32
33  private AbCPredicate isAnInterestReply(Object o){
34   if (o instanceof Tuple) {
35    Tuple t = (Tuple) o;
36    try {
37     if ((getValue(Defs.interest).equals(t.get(0))&&
38       Defs.REPLY.equals(t.get(1)))
39       {
40        return new TruePredicate();
41     }
42    } catch (AbCAttributeTypeException e) {
43     e.printStackTrace();
44    }
45   }
46   return new FalsePredicate();
47  }
48
49  private AbCPredicate isAnInterestUpdate( Object o ) {
50   if (o instanceof Tuple) {
51    Tuple t = (Tuple) o;
52    try {
53     if ((getValue(Defs.interest).equals(t.get(1)))&&
54       (Defs.UPDATE.equals(t.get(2))) {
55      return new TruePredicate();
56     }
57    } catch (AbCAttributeTypeException e) {
58     e.printStackTrace();
```

```
59      }
60     }
61     return new FalsePredicate();
62    }
63
64   }
```

Communication Ports In $Ab^{a}C$uS each component is equipped with a set of ports for interacting with other components. A port is identified by an address which can be used to manage the connection with the underlying communication infrastructure that mediates the interaction between AbC components. It should be noted that these ports are not meant to be used by components to identify the addresses of others, but rather as a way to access the communication infrastructure.

The abstract class `AbCPort` implements the generic behavior of a port. It provides the instruments to dispatch messages to components and partially implements the communication protocol used by AbC components to interact. The send method in `AbCPort` is abstract to allow for different concrete implementations depending on the underlying network infrastructures (i.e., Internet, Wi-Fi, Ad-hoc networks, ...).

In this section we abstract from the existence of a specific communication infrastructure and provide an implementation for a virtual port. This implementation serves as a proof of concept and is also used for running case studies.

The virtual port is used to run components in a single application without relying on a specific communication infrastructure and the interactions are performed via a buffer stored in the main memory. The virtual port is implemented via the class `VirtualPort`. An instance of this class consists of a set of local ports that are used by components to send and receive messages. Every AbC component acquires a unique local port for interaction. The class `LocalPort` implements the class `AbCPort` and provides custom mechanisms for sending and receiving messages. The virtual port manages the use of its local ports by assigning a unique local port to each component and ensures that only a single local port can send at a given moment by relying on a lock, shared between all local ports. If a

component wants to send a message, it acquires the lock through its assigned local port and releases it only when the send operation terminates. This is important to guarantee that messages are handled in the correct order.

Example 6.1. *Assume that the processes P and Q represent the behavior of two different components $C_1 = \Gamma_1 : P$ and $C_2 = \Gamma_2 : Q$ where $P \triangleq (v)@\Pi_1.(x = w)(x).0$ and $Q \triangleq (w)@\Pi_2.(x = v)(x).0$. Assume that $\Gamma_1 \models \Pi_2$ and $\Gamma_2 \models \Pi_1$. If we execute these components in parallel, the possible outcomes according to the interleaving semantics would be either $\Gamma_1 : 0 \parallel \Gamma_2 : (x = v)(x).0$, by executing the sequence $\xrightarrow{\overline{\Pi_1}v}\xrightarrow{\overline{\Pi_2}w}$, or $\Gamma_1 : (x = w)(x).0 \parallel \Gamma_2 : 0$, by executing the sequence $\xrightarrow{\overline{\Pi_2}w}\xrightarrow{\overline{\Pi_1}v}$. If we allow concurrent execution of send operations we might end up in a situation where both processes receive messages from each other which is not sound with respect to the semantics (i.e., $\Gamma_1 : 0 \parallel \Gamma_2 : 0$). This is why a shared lock is needed to program interleaving correctly.*

The following $Ab^{a}C$uS code shows how to create a `VirtualPort`, get a fresh local port, assign it to a participant component "`p`", and start its execution respectively.

```
1      VirtualPort vp = new VirtualPort();
2      AbCPort port=vp.getPort();
3      p.setPort(port);
4      port.Start();
```

6.2 Implementing the Communication Infrastructure

As we have seen in previous chapters, *AbC* has proved to be powerful by means of encoding other classical communication paradigms, however, the question about the tradeoff between its expressiveness and its efficiency, when implemented to program distributed systems, is still to be answered. In the rest of this chapter, we lay the basis for an efficient implementation of a distributed communication infrastructure for *AbC* that respects its formal semantics. The main issue when implementing a communication model, especially for group-based models, is to ensure that every message

is delivered to all possible receivers in the same order. In the operational semantics of AbC, to respect the order of message delivery and as the case in any theoretical model, we abstracted from implementation details and assumed that message exchange is atomic and thus that at a given instant only one component can send while all others block execution until the sent message is delivered to all possible receivers. However, in real implementations we cannot afford atomic message exchange, because it is extremely inefficient. Actually, we wish to reduce synchrony as much as possible, but still we want to mimic the effect of atomic message exchange.

Despite the extensive theoretical results on process calculi, and especially broadcast-based process calculi, distributed coordination infrastructures for managing the interaction of actual computational systems are still scarce. In distributed infrastructures, many servers collaborate asynchronously to deliver messages to available receivers which makes the correctness of their overall behavior not obvious. In this chapter, we propose three possible implementations of a distributed coordination infrastructure to manage multiparty interactions, independently from a specific set of coordination primitives. These include cluster-based, ring-based, and tree-based infrastructures. We prove their correctness and finally we model them as stochastic processes to evaluate their performance. The results show that the tree-based infrastructure outperforms others in terms of minimizing the average delivery time and the average time gap between the delivery of two consecutive messages. Since the decision on whether to accept or discard a message in AbC is resolved at the receiver side, it is easy to see that the expressive power of attribute-based communication does not add any extra complexity to message exchange and actually has a similar complexity when implementing any broadcast process calculus. Thus, these infrastructures can be used efficiently for AbC or any multicast/broadcast process calculus.

In the literature, a plethora of approaches have been proposed to maintain the order of message delivery for multiparty communication. These approaches are however difficult to compare as they often differ in their assumptions, properties, objectives, or target applications. A comprehensive comparative study that highlights the commonalities and the dif-

ferences between these approaches can be found in (DSU04). However, these approaches can be classified according to the ordering mechanisms they adopt. These include: the fixed sequencer-based approach (CM95), the moving sequencer-based approach (CM84), the privileged-based approach (Cri91), and the communication history-based approach (PBS89).

In the fixed sequencer approach, one node takes the responsibility of maintaining the order of message delivery and agents can communicate by sending and receiving messages to/from the sequencer. Our centralized algorithm, used e.g in (ADL16b) is based on this approach. The cluster-based infrastructure is a natural extension of this approach, in the sense that multiple copies of the sequencer can collaborate to deliver messages by sharing an input queue and a counter. Also our tree-based infrastructure can be considered as a generalization of this approach where instead of a single sequencer, we consider a propagation tree which is a common infrastructure for group communication (Pel00). The ordering decisions are resolved along the tree paths. When the depth of the propagation tree is 1, our algorithm behaves just like other fixed sequencer approaches.

The moving sequencer approach tries to avoid the bottleneck induced by a single sequencer, by transferring the sequencer role among several nodes. These nodes form a logical ring and circulate a token that carries a sequence number and a list of all sequenced messages. Upon receipt of the token, a sequencer assigns sequence numbers to its messages, sends all sequenced messages to destinations, updates the token, and passes the token and sequenced messages to the next sequencer. In this way the load is distributed among several nodes. However, the liveness of the algorithm depends on the token and, if the number of senders in one node is larger than others, fairness is hard to achieve. The ring-based infrastructure can be viewed as a generalization of this technique where fairness is resolved by sharing a common counter rather than circulating a token.

In the privilege-based approach, senders circulate a token that carries a sequence number and each sender has to wait for the token. Upon receipt the sender assigns sequence numbers to its messages, sends them to destinations, and passes the token to the next sender. This approach is not suitable for open systems, since it assumes that all senders know

each other. Also fairness is hard to achieve, when some agents send larger number of messages than the others.

In the communication history approach, senders also define the order in which their messages are to be received. Senders can send at anytime by simply attaching timestamps to their messages. The order is maintained by delaying the delivery of messages, i.e., an agent can deliver a message m from another agent only after it has received, from every agent in the network, a message that was sent before reception of m. Liveness problem arises if some agents in the network are unable to send messages. This approach is suitable for synchronous systems with synchronized clocks and physical timestamps.

It should be noted that with the exception of the last approach, which is however not suitable for asynchronous systems, all the above approaches either make assumptions about the topology of the communication infrastructure or require that the communicating partners know each other to guarantee the order of message delivery.

Clearly, centralized infrastructures are not efficient, nonetheless they are easy to develop and maintain, e.g., it is not difficult to respect the order of message delivery in a centralized infrastructure since messages are maintained by a single server that stores incoming messages in a specific order in its FIFO queue and delivers them to receiver in a specific order. For some application, it is much more convenient to rely on a centralized infrastructure rather on a distributed one. For the sake of completeness we proceed by providing a centralized implementation for the communication infrastructure. We discuss its advantages and disadvantages and then we focus more on our distributed proposals.

Centralized communication infrastructure

In this section we provide a centralized implementation of the communication infrastructure which is based on the fixed sequencer approach. We assume the presence of a centralized server/ broker that mediates the interaction between components as shown in Figure 7. In essence the message broker, labeled "X", accepts messages from sending components, and delivers them to all subscribed components with the exception of the

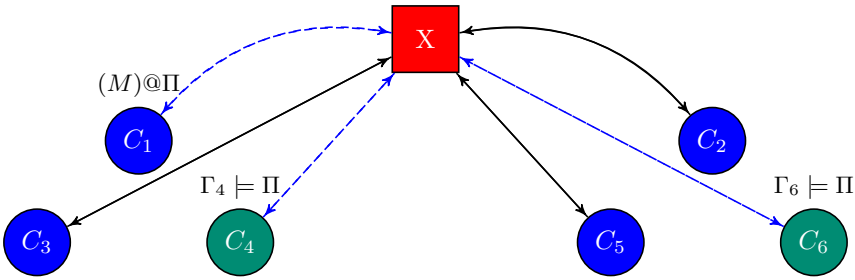

Figure 7: Centralized communication infrastructure

sending ones. This central server plays the role of a forwarder and does not contribute in any way to message filtering. The decision about accepting or ignoring a message is taken when the message is delivered to the receiving components. The implementation of the coordination algorithm is straightforward. Components have to subscribe to server X at any point of time to be considered for future communication. Once a component is subscribed, it will be able to receive messages from other components through the message broker. The broker keeps an entry for each subscribed component which is the tuple $\langle id, addr \rangle$, where id represents a unique id for each component and $addr$ which specifies the address of the subscribed component. During the subscription phase, server X communicates the port number where it accepts data messages to the subscribed component. If a component, say C_1, wants to send a message M to a group of components that satisfy the predicate Π, it just sends the message to server X and blocks execution until server X signals back an acknowledgement to indicate that the message has been processed. Server X forwards message M to all subscribed components with the exception of the sending one and the receiving components decide either to accept or ignore the message based on their satisfaction of the sending predicate Π. In our case, only components C_4 and C_6 are qualified to receive message M because they satisfy the sending predicate Π, while the other components discard the message.

The interleaving semantics is preserved by this algorithm even if mul-

tiple components send messages concurrently. This is because only server X is mediating the interaction and can dispatch messages to components in some order. The order of messages is preserved because each component is blocked until the server signals acknowledgement of processing the message. This means that the next action of the sending process is not enabled until an acknowledgment is received. However, we still need to constrain individual components from concurrently sending and/or receiving messages in the sense that if a component is receiving a message, it is not allowed to send any other messages until the receiving operation is finished and the same applies to the sending operation. Otherwise there would be the chance, in a multi-threaded component, to receive a message that was not supposed to be received according to the operational semantics.

Example 6.2. *Let the behavior of component* $C = \Gamma_1 : P$ *be defined as follows:*

$$P \triangleq (a,\ v)@\Pi_1.(x = a)(x,\ y).0 \mid (x = b)(x,\ y).0$$

which is the parallel composition of two processes, one is attempting to send a message on Π_1 *and then wait for receiving on* $(x = a)$ *and the other is waiting to receive on* $(x = b)$*. Now consider the following:*

1. *Another component* $C' = \Gamma_2 : Q$ *executes in parallel and has the following behavior* $Q \triangleq (a,\ v')@\Pi_2.(x = a)(x,\ y).0$ *such that* $\Gamma_1 \models \Pi_2$ *and* $\Gamma_2 \models \Pi_1$*. Notice that this behavior is similar to the behavior of the subprocess on the left hand side of the parallel composition in process* P*. If both components* C *and* C' *send concurrently and continue executing without waiting for an acknowledgements, they will both receive from each other since the next action is enabled for both of them. However, this is not sound with respect to the operational semantics. That is why acknowledgements are necessary to preserve the semantics.*

2. *If message* $\overline{\Pi_3}(a, v'')$ *reaches component* C *before message* $\overline{\Pi_1}(a, v)$ *is emitted where* $\Gamma_1 \models \Pi_3$*, the sending operation is blocked until the receiving operation is finished. In our case the message is discarded since process* P *is either sending on* Π_1 *or receiving on* $(x = b)$*. If we allow concurrent sending and receiving for co-located processes then there is a chance that the component* C *emits the message* $\overline{\Pi_1}(a, v)$

while processing $\overline{\Pi_3}(a, v'')$ *and evolves to* $\Gamma_1 : 0 \mid (x = b)(x,\ y).0$ *which is not sound with respect to the operational semantics. That is why we do not allow concurrent execution of send and receive operations.*

We would like to stress that, although this implementation is centralized, components interact anonymously and combine their behaviors to achieve the required goals. Components are unaware of the existence of each other, as they only interact with the message broker. The latter can be seen as an access point which mediates the interaction between components. It serves as a forwarder that shepherds the interaction, but it has nothing to do with message filtering. It should be noted that the AbC server and the client ports usually operate from different machines. The following $Ab\overset{a}{C}$uS code shows how to create an `AbCServer` and start its execution.

```
1       AbCServer server = new AbCServer();
2       server.start();
```

The following code shows how to create a client port, register it to an existing server, assign the port for a participant component p_1, and finally, start its execution.

```
1     AbCClient client = new AbCClient(InetAddress.getLoopbackAddress(),
          1234);
2     client.register(
          InetAddress.getLoopbackAddress(),DEFAULT_SUBSCRIBE_PORT );
3     p1.setPort(client);
4     client.start();
```

Centralized infrastructures are easy to develop and deploy. Actually they can be prototyped and tested very quickly. They are also easy to maintain, as there is only a single point of failure which is the main server. However, these advantages come with a price as such infrastructures are not fault tolerant or stable. For instance if the main server fails, the entire infrastructure will be non-functioning. Also relying on a single exchange server may become a bottleneck for performance because these infrastructures do not scale well with a large number of connected components. When the number of messages increases substantially, the server becomes

overloaded and unable to handle further requests and most likely will go down. So there is a trade off between ease of development, maintainability, stability, scalability, and fault tolerance and it really depends on the application of interest. In some applications, with a fixed and relatively small number of components distributed in a small geographic area, this kind of infrastructures might be the optimal choice. On the other hand, if the number of components is very large and can increase at anytime with different factors and the components are distributed in a large area, other kinds of infrastructures are needed.

6.3 Multiparty Interaction Style

In this section, we fix the general multiparty communication model that we consider for our distributed proposals, independently from a specific set of communication primitives. In essence, a system, in this model, consists of a set of parallel agents that interact through message passing. Each agent can perform either send or receive activities. The criteria for deriving the interaction between different agents vary according to the set of communication primitives of interest. For instance, in bπ-calculus, agents can exchange messages only when they agree on a specific channel name while in AbC, agents rely on the content of messages and if the receiver satisfies the sender requirements. Message transmission is autonomous (non-blocking), but reception is not. For instance, an agent can still send a message even there are no receivers, but receive operation can only happen through synchronization with an available sent message. Whenever an agent transmits a message, all agents running in parallel catch the message. Agents can then decide whether to accept or discard the message according to the criterion, adopted by their communication primitives. To respect the order of message delivery in such cases, the multiparty communication model relies on the notion of interleaving semantics or atomic message exchange where at a given instant only one agent can send a message while all other agents block execution until the sent message is delivered to all possible receivers. The following example explains how the communication links are derived.

Example 6.3. *Let us consider three agents A_1, A_2,and A_3, executing in parallel with the following behaviors $A_1 = \overline{a}v.0$, $A_2 = b(x).a(x).P$, and $A_3 = a(x).\overline{b}w.0$ where $\overline{a}v$ (resp. $\overline{b}w$) denotes sending message v (resp. w) on channel a (resp. b) and $a(x)$ (resp. $b(x)$) denotes receiving a message on channel a (resp. b). Messages sent on a channel name, e.g., a, can only be received on the same name, e.g., a. According to the above description, the only legal interaction trace is the following:*

$$A_1 \| A_2 \| A_3 \xrightarrow{\overline{a}v} 0 \| A_2 \| \overline{b}w.0 \xrightarrow{\overline{b}w} 0 \| (a(x).P)[w/x] \| 0$$

When we assume atomic message exchange, agent A_1 takes the first step and sends message $\overline{a}v$ while both A_2 and A_3 block execution until $\overline{a}v$ is delivered to both of them. In this case, only A_3 can receive the message while A_2 stays unchanged. The next step is taken by A_3 which sends $\overline{b}w$ that can only be received by A_2 and after this step no agent is able to send any message.

6.4 Distributed Coordination Infrastructures

In this section, we consider three different implementations of a distributed coordination infrastructure for multiparty communication models that assume atomic message exchange with non-blocking message sending. The idea is to allow modelers to specify systems in a synchronous way, which is more intuitive and simpler in terms of modeling and mathematical reasoning, while the implementation is asynchronous, which is more efficient and desired in distributed settings. The implementation should still provide the same guarantees of the synchronous model. In essence, the infrastructure should guarantee the following properties:

1. A message cannot be lost and should be delivered to all agents connected to the infrastructure.

2. Message sending is non-blocking in the sense that an agent does not have to wait until its message is delivered.

3. Multiple messages may travel concurrently in the infrastructure, but the order, in which they are delivered to every agent, is the same.

The first two properties are obvious and to explain the third one we rely on Example 6.3. For an implementation, we cannot assume atomic message exchange, but we still want to mimic its effect. Now consider the following scenario: assume that after A_1 sent message $\overline{a}v$, the infrastructure sent a copy of $\overline{a}v$ to all connected agents, i.e., A_2 and A_3. Agent A_3 received its copy and then sent $\overline{b}w$. Two different messages are now in transit and if $\overline{b}w$ arrives first to A_2, we are in trouble since A_2 can now consume both messages while it is supposed to consume only $\overline{b}w$. The third property above guarantees that $\overline{a}v$ is delivered to A_2 before $\overline{b}w$.

In the rest of this section we show how these properties can be guaranteed by relying on sequence numbers. The idea is to label each message with a unique id that is globally identified and based on this id, messages can be delivered in a correct order.

6.4.1 A Cluster-based Infrastructure

We consider a set/cluster of nodes, sharing a counter for sequencing messages and an input queue to retrieve messages sent by agents. Cluster nodes can have exclusive locks on both the cluster counter and the input queue. A cluster infrastructure can be viewed as a cloud infrastructure where agents are registered. Formally, the cluster-based infrastructure is defined below:

Definition 11 (Cluster node)**.** *A cluster node c is represented by the tuple $c = \langle addr, \mathcal{C} \rangle$ where addr denotes its address, $\mathcal{C}$ is a reference to the cluster infrastructure where c belongs.*

Definition 12 (Cluster infrastructure)**.** *A cluster infrastructure $\mathcal{C}$ is represented by the tuple $\mathcal{C} = \langle addr, ctr, \mathcal{S}, \mathcal{A}, \mathcal{I} \rangle$ where addr denotes its address, ctr denotes a counter to generate fresh ids, initially the value of ctr equals 0, $\mathcal{S}$ denotes the set of connected cluster nodes, $\mathcal{A}$ denotes the set of connected agents, and $\mathcal{I}$ denotes an input queue.*

In this chapter and for any tuple $T = \langle t_1, \ldots, t_n \rangle$, we shall use the notation $T[t_i]$ to refer to the value of the element t_i in tuple T. Sometimes we abuse the notation and use t_i instead of $T[t_i]$ when it is understood from context. Also if $T[t_i]$ is uniquely identified, we use t_i to refer to it, e.g., we use *ctr* and $\mathcal{I}$ for $\mathcal{C}[ctr]$ and $\mathcal{C}[\mathcal{I}]$ in the cluster infrastructure.

Definition 13 (Cluster agent)**.** *A cluster agent a is represented by the tuple* $a = \langle addr,\ nid,\ mid,\ \mathcal{W}, \mathcal{C}\rangle$ *where addr denotes its address, nid denotes the expected next message id to be processed, initially equals to 0, mid denotes the id of a recent reply message,* $\mathcal{W}$ *denotes a priority waiting queue to store unordered messages where the top of* $\mathcal{W}$ *is the message with the least id, and* $\mathcal{C}$ *is a reference for the cluster infrastructure where a belongs.*

Agents are not aware of the cluster nodes and they only emit messages to be added directly to the cluster's input queue $\mathcal{I}$. Cluster nodes concurrently retrieve messages from $\mathcal{I}$ and handle them by either assigning them unique ids based on the shared counter *ctr* and send them back to the requesters or forwarding them to other agents in the cluster. We consider three labeled messages: a request message {"REQ", src, dest}, a reply message {"RPLY", id, src, dest}, and a data message {"DATA",id, src, dest}.

Algo 1: Cluster Agent

```
1  when (Receive(m))
2      switch m do
3          case { "RPLY", id, src, addr}
4              mid = id;
5          end
6          case { "DATA", id, src, addr}
7              if id ≥ nid then
8                  W ←↩ m;
9              end
10         end
11     endsw
12 when (Send(m))
13     ⟨I ←↩{ "REQ", addr, C[addr]}⟩;
14     mid = −1;
15     when (nid = mid)
16         nid = nid + 1;
17         ⟨I ←↩{ "DATA",mid, addr,C[addr]}⟩;
18 when (!W.isEmpty() ∧ W.top.id == nid)
19     m = W.remove();
20     nid = nid + 1;
21     handle(m);
```

We also use the notation $\langle op_1(obj); \ldots\ op_n(obj);\rangle$ to denote an exclusive lock to object obj where the execution of the block $\langle\rangle$ is atomic and blocking. When a cluster node takes this lock on either *ctr* or $\mathcal{I}$, other nodes can neither have write nor read access to them. The behavior of a cluster agent is reported in Algo 1. A cluster agent can concurrently perform three events, represented by the keyword "when". When a reply message

Algo 2: Cluster node

```
1  while true do
2      ⟨m = I.remove()⟩;
3      switch m do
4          case { "REQ", src, dest}
5              ⟨id = ctr; ctr = ctr + 1⟩;
6              Send({"RPLY",id, addr,
               src},src);
7          end
8          case { "DATA", id, src, dest}
9              ∀a ∈ A : if a ! = src then
10                 Send(m, a);
11             end
12         end
13     endsw
14 end
```

with a fresh id arrives (Lines 3-5), *mid* is assigned the reply's id, this will signal a waiting send activity to proceed. When a data message arrives (Lines 6-9), the message is put in the agent waiting queue. In case an agent wants to send a message (Lines12-17), it emits an id request message to be put in the cluster input queue $\mathcal{I}$, resets *mid* to -1 and waits until the reply signal arrives. The agent needs also to wait until previous messages are received (i.e., $nid = mid$), it then increments its *nid* and emits a data message to be put in the cluster input queue. If the agent is multithreaded, this method ensures that send activities of a single agent are processed sequentially, otherwise the order is violated. Finally, when the id of message m, on top of the waiting queue $\mathcal{W}$, becomes equal to the agent nid (i.e., m is in a correct order) (Lines 18-21), the agent removes the message from $\mathcal{W}$, increments its nid by 1, and handle message m.

The behavior of a cluster node is reported in Algo 2. Cluster nodes always compete to retrieve messages from the shared queue $\mathcal{I}$ by acquiring exclusive locks (Line 2). In case of request messages, (Lines 4-7), a cluster node takes a lock on the shared counter, copies its current value to be sent to the requester, releases it after incrementing it by 1, and finally sends a reply message, carrying a fresh id, to the requester, addressing it by its address, *src*. In case of data messages, (Lines 8-12), except for the sender, a node forwards the message to all registered agents in the cluster.

Algo 3: Ring agent

```
1  when (Receive(m))
2  |  switch m do
3  |  |  case { "RPLY", id, src, addr}
4  |  |  |  mid = id;
5  |  |  end
6  |  |  case { "DATA",id, src, addr}
7  |  |  |  nid = nid + 1;
8  |  |  |  handle(m);
9  |  |  end
10 |  endsw
11 when (Send(m))
12 |  Send({ "REQ", addr, r},r);
13 |  mid = −1;
14 |  when (nid = mid)
15 |  |  nid = nid + 1;
16 |  |  Send({ "DATA",mid, addr, r},r);
```

Proposition 1. *For every agent $a \in \mathcal{C}[\mathcal{A}]$, if a sends a request to $\mathcal{C}$ then eventually some $c \in \mathcal{C}[S]$ will send a reply to a.*

Proof. The proof is immediate from Algo 1, Line 13, where an agent emits a request to be put in the input queue of the cluster $\mathcal{I}$ with an exclusive lock and from Algo 2, Lines 1-7, where cluster nodes are always competing to retrieve messages from $\mathcal{I}$. If the message is a request, then

a node should send a reply to the exact requester.

□

Theorem 6.1. *Any data message is eventually received by all cluster agents.*

Proof. The proof is immediate from Algo 2, Lines 2 and Lines 8-11, where nodes are always retrieving data messages from $\mathcal{I}$ and forwarding them to connected agents. There is no a chance that a message can stay in $\mathcal{I}$ indefinitely.

□

The order of message delivery is guaranteed because the shared counter *ctr* and the shared input queue $\mathcal{I}$ can only be modified and/or read with an exclusive lock which maintains their consistency, thus two different messages can neither take the same id nor be retrieved by two different cluster nodes. Also the priority queue of an agent, $\mathcal{W}$, stores data messages and only allows the agent to handle them when they are in correct order as shown in Algo 1, Lines 18-21.

6.4.2 A Ring-based Infrastructure

We consider a set of nodes, organized in a ring-based topology and sharing a counter for sequencing messages coming from agents. Each node is responsible for a group of agents and can have exclusive locks to the ring counter. Formally, the ring-based infrastructure is defined below:

Definition 14 (Ring node)**.** *A ring node r is represented by the tuple $r = \langle addr,\ nid,\ nxt,\ \mathcal{A},\ \mathcal{W}\rangle$ where addr, nid, $\mathcal{A}$, and $\mathcal{W}$ are defined as before while nxt is a reference for its next ring node.*

Definition 15 (Ring infrastructure)**.** *A ring infrastructure $\mathcal{R}$ is represented by the tuple $\mathcal{R} = \langle \mathcal{S},\ ctr\rangle$ where ctr is defined as before while $\mathcal{S}$ denotes the set of ring nodes. We have that:*

- $\forall r \in \mathcal{R}[\mathcal{S}] :\ r[nxt] \neq \perp \wedge\ r[nxt] \in \mathcal{R}[\mathcal{S}]$.
- $\forall r_1, r_2 \in \mathcal{R}[\mathcal{S}] : r_1[nxt] = r_2[nxt]$ *implies* $r_1 = r_2$.

Definition 16 (Ring agent)**.** *A ring agent a is represented by the tuple $a = \langle addr,\ nid,\ mid,\ r\rangle$ where addr, nid, and mid are defined as before while r is reference for a ring node where the agent is connected.*

The behavior of a ring agent is reported in Algo 3. A ring agent can concurrently perform two events. When a reply message with a fresh id arrives (Lines 3-5), *mid* is assigned the reply's id, this will signal a waiting send activity to proceed. When a data message arrives (Lines 6-9), the agent increments its nid by 1 and handles the message. In case an agent wants to send a message (Line 11-16), it sends an id request message to its ring node where it belongs. the agent waits a reply id, *mid*, and also until previous messages are received (i.e., $nid = mid$), it then increments its *nid* and sends a data message to its ring node.

The behavior of a ring node is reported in Algo 4. A ring node can concurrently respond to two events. When an id request message arrives (Lines 4-7), the node acquires an exclusive lock to the ring counter, copies it current value, releases it after incrementing it by 1, and finally sends a reply message, carrying a fresh id, to the requester agent. When a data message arrives (Lines 8-12), the message is added to the node's waiting queue, $\mathcal{W}$. Finally, when the id of message m, on top of the waiting queue $\mathcal{W}$, becomes equal to the node nid (Lines 14-20), the node removes the message from $\mathcal{W}$, increments its nid by 1, forwards the message to its next node in the ring, and finally forwards m to all of its connected agents except for the sender.

Algo 4: Ring node

```
1  while true do
2  |  when (Receive(m))
3  |  |  switch m do
4  |  |  |  case {"REQ", src, addr}
5  |  |  |  |  ⟨id = ctr; ctr = ctr + 1⟩;
6  |  |  |  |  Send({"RPLY",id, addr,
   |  |  |  |  src} , src);
7  |  |  |  end
8  |  |  |  case {"DATA",id, src, addr}
9  |  |  |  |  if id ≥ nid then
10 |  |  |  |  |  W ↤ m;
11 |  |  |  |  end
12 |  |  |  end
13 |  |  endsw
14 |  when (!W.isEmpty() ∧ W.top.id == nid)
15 |  |  m = W.remove();
16 |  |  nid = nid + 1;
17 |  |  Send(m, nxt);
18 |  |  ∀a ∈ A : if a != m.src then
19 |  |  |  Send(m, a);
20 |  |  end
21 end
```

Proposition 2. *For every node $r \in \mathcal{R}$ and agent $a \in r[\mathcal{A}]$, if a sends a request to r then eventually r will send a reply to a.*

Proof. The proof is immediate from Algo 3, Line 12, where an agent emits a request to the ring node where it belongs and from Algo 4, Lines 4-7, where ring nodes are always competing to get exclusive access to the

shared counter and get fresh ids. A node sends a reply directly to the requester as required. □

Theorem 6.2. *Any data message is eventually received by all ring agents.*

Proof sketch. We sketch the proof here because it is a bit similar to the proof of Theorem 6.3 which we will cover in details. The reason is to avoid repetition and highlight more involved proofs. In any case, to prove this theorem, we need to define a successor relation between ring nodes. e.g., we say that r_2 is a direct successor of r_1, written $r_2 \succ r_1$, if and only if $r_1[nxt] = r_2[addr]$ and $\succ^+$ is the transitive closure of $\succ$. We need to stress that a node cannot be a successor of itself. i.e., for each node $r \in \mathcal{R}$, we have that $r \not\succ^+ r$. We need a similar definition of Definition 20 to define the algorithm execution. We need a similar lemma of Lemma 6.1 where we show that two neighbor nodes will eventually converge to the same *nid*. This lemma can be proved by induction on on the difference between $r_2[nid]$ and $r_1[nid]$. We need also a similar proposition of Proposition 6 to prove that any two nodes in the ring infrastructure will eventually converge to the same *nid*. This proposition can be proved based on the previous lemma and by induction on the distance between two nodes in the infrastructure. The distance can be computed in a similar way of Function $d(t_1, t_2)$ in the proof of Proposition 6. Finally, we need to make sure that messages do not stay in a ring node indefinitely, so we need a similar proposition of Proposition 7 which can be proved by relying on the previous proposition, also Proposition 2, and by induction on the difference by between $r[nid]$ and $m.id$ where $m = \mathcal{W}.top()$. □

The order of message delivery is guaranteed because the shared counter *ctr* again can only be modified and/or read with an exclusive lock. Also requests and replies involve only the agent and its parent node, so duplicate requests and replies are avoided. Data messages, coming from a neighbor node, are only added to the node's queue if their ids are greater of equal to the node's *nid* and thus discarding duplicate data messages (i.e., when $m.id < nid$). Finally, data messages are only forwarded to children and neighbor when they are in correct order, thus incrementing the *nid* counter as shown in in Algo 4, Lines 14-20.

6.4.3 A Tree-based Infrastructure

We consider a set of nodes, organized in a tree-based topology. An agent can be connected to one node in the tree and can interact with other agents in any part of the tree by only sending and receiving messages to/from its parent node. As in the previous infrastructures, when an agent wants to send a message, it asks for a fresh id from its parent node. If the parent node is the root of the tree, it sends a reply message with a fresh id to the requester agent, otherwise it forwards the message to its own parent. Only the root of the tree is responsible for assigning fresh ids to messages. To reduce the number of messages needed to get a reply message with a fresh id from the root, the request and reply messages now have an extra a field, called *route*.

The route field is a linked list, used to backtrack to the requester agent. When an agent sends a request message, it adds its address to the route and every time the request message is received by an intermediate node, its address is added to the route. Finally, when the root receives the request message, it removes the last added address in the route and sends it the reply. The same happen until the reply reaches the requester agent. Formally, the tree-based infrastructure is defined as follows:

Algo 5: Tree agent

```
1  when (Receive(m))
2  |  switch m do
3  |  |  case { "RPLY", id, route, src, addr}
4  |  |  |  mid = id;
5  |  |  end
6  |  |  case { "DATA",id, src, addr}
7  |  |  |  handle(m);
8  |  |  |  nid = nid + 1;
9  |  |  end
10 |  endsw
11 when (Send(m))
12 |  rt = route.add(addr);
13 |  Send({"REQ", rt, addr, t},t);
14 |  mid = −1;
15 |  when (nid = mid)
16 |  |  Send({"DATA",mid, addr, t},t);
17 |  |  nid = nid + 1;
```

Definition 17 (Tree node)**.** *A tree node t is represented by the tuple* $t = \langle addr,\ ctr,\ nid,\ \mathcal{P},\ \mathcal{D},\ \mathcal{W}\rangle$ *where addr, ctr, nid and* $\mathcal{W}$ *are defined as before. The symbol* $\mathcal{P}$ *denotes a reference to a parent node (if any) and* $\mathcal{D}$ *denotes the node's descendants which can be agents and/or tree nodes (i.e.,* $\mathcal{D} = \mathcal{A} \cup \mathcal{S}$*).*

The counter *ctr* is only used in the root node to generate fresh ids and, for the sake of uniformity, we have it in the definition of a tree node.

Definition 18 (Tree infrastructure)**.** *A tree infrastructure $\mathcal{T}$ consists of a set of tree nodes $\{t_0, \ldots, t_k\}$, structured in a tree-based topology. For every pair of nodes $t_1, t_2 \in \mathcal{T}$, we say that:*

- *t_1 is a direct descendant of t_2, written $t_1 \prec t_2$, if and only if $t_1[\mathcal{P}] = t_2[addr]$ and $\prec^+$ denotes the transitive closure of $\prec$*

- *$t[addr] = t'[addr]$ to denote that $t = t'$, with possibly different ctr, nid, and $\mathcal{W}$ (i.e., node t evolved into t' in response to message exchange in the infrastructure).*

Definition 19 (Well formedness)**.** *A tree infrastructure $\mathcal{T}$ is well formed if and only if:*

1. *For each node $t \in \mathcal{T}$, we have that $t \not\prec^+ t$.*

2. *There exists a node $t \in \mathcal{T}$ such that for any $t' \in (\mathcal{T} \backslash \{t\})$, $t' \prec^+ t$ and we have that:*

   - *$t'[nid] \leq t[ctr]$.*
   - *For any message $m \in t'[\mathcal{W}]$ we have that $m.id \leq t[ctr]$.*

3. *For each node $t \in \mathcal{T}$ and for each message $m \in t[\mathcal{W}]$, we have that $m.id \geq t[nid]$.*

4. *If nodes $t, t' \in \mathcal{T}$ and $t[\mathcal{P}] = t'[\mathcal{P}] = \perp$ then we have that $t = t'$.*

The first statement in Definition 19 states that every node is not a descendent of itself. The second statement defines the root of the infrastructure where all nodes are considered to be its descendants. The next id for each node in the infrastructure is at most equal to the value of the counter in the root node (i.e., $t'[nid] \leq t[ctr]$). The value of the counter in the root node gives an upper bound on the number of message ids (i.e., $t'[nid] \leq t[ctr]$). The third statement states that the expected next id for any node cannot be greater than any message in its waiting queue. The last statement ensures that there exists a unique root in the infrastructure.

A tree agent $a = \langle$addr, nid, mid, t$\rangle$, is defined in a similar way of a ring agent with the exception that it has a reference to a tree node t, instead of a ring node. The behavior of a tree agent is reported in Algo 5. It has almost the same behavior of a ring agent with the exception that an agent adds its own address in the route linked list before sending a request as shown at line 12. The behavior of a tree node is reported in Algo 6. A tree node can concurrently respond to two events, represented by the keyword "when". When an id request message arrives (Lines 4-15), if the node is the root of the tree, it generates a fresh id which is equal to the current value of its counter, increments its counter by 1, determines the source of the message by getting the last value added to the route, creates a reply message, carrying the fresh id, and sends to the source. If the node is an intermediate node, it adds its own address to the route of the message and forwards the message to its parent. When a reply message arrives (Lines 16-19), the node determines the destination of the message by getting the last value added to the message's route and forwards the message to that destination. When a data message arrives (Lines 20-27), if the node is not the root and its parent is not the source of the message, the message is directly forwarded to the node's parent and in any case the message is added to the node's waiting queue $\mathcal{W}$. Each time a message arrives to a node. Its source field is set

Algo 6: Tree node

```
1  while true do
2      when (Receive(m))
3          switch m do
4              case { "REQ", route, src, addr}
5                  if P == ⊥ then
6                      id = ctr;
7                      ctr = ctr + 1;
8                      dest = route.remove();
9                      m'={"RPLY", id, route, addr,
                       dest}
10                     Send(m', dest);
11                 else
12                     m.route.add(addr);
13                     Send(m, P);
14                 end
15             end
16             case { "RPLY",id, route, src, addr}
17                 dest = route.remove();
18                 Send(m, dest);
19             end
20             case { "DATA",id, src, addr}
21                 if P != src ∧ P != ⊥ then
22                     m.src = addr;
23                     m.dest = P;
24                     Send(m, P);
25                 end
26                 W.add(m);
27             end
28         endsw
29     when (!W.isEmpty() ∧ W.top.id == nid)
30         m = W.remove();
31         src = m.src;
32         m.src = addr;
33         ∀d ∈ D\{src} : m.dest = d;
34         Send(m, d);
35         nid = nid + 1;
36 end
```

to be the node's address and its destination field is set according where it should go. This is important to make sure that messages do not circulate in the tree indefinitely. Finally, when the id of message m, on top of the waiting queue $\mathcal{W}$, becomes equal to the node nid (Lines 29-35), the node removes the message from $\mathcal{W}$, copies the source of the message, sets its address to be the source of the message, forwards the message to all of its connected nodes and agents with the exception for the source of the message, and increments its nid by 1.

Now we discuss the properties that are preserved by the tree infrastructure.

Definition 20. *We write $\mathcal{T} \rightarrow \mathcal{T}'$ if and only if $\mathcal{T}$ evolves to $\mathcal{T}'$ after the execution of Algorithm 6. The relation $\Rightarrow$ denotes multiple executions of Algorithm 6.*

Proposition 3. *If $\mathcal{T}$ is well formed then each $\mathcal{T}'$ such that $\mathcal{T} \rightarrow \mathcal{T}'$ is well formed.*

Proof. The proof is immediate from Definition 19 and the fact that the algorithm does not change the underlying infrastructure. Also the root node is the only node that can increment the counter, used to assign ids to messages, and nodes can only increment their next id with respect to already generated message ids as shown in Algo 6, Lines 29-35. So it is immediate that the expected next id in any node can be at most equal to the root counter. The same applies for message ids.

□

We will only consider well formed infrastructures. Since the topology of the infrastructure is tree-based, the request messages should be propagated from a node to its parent and so on. The reply messages should be propagated from a parent to its children and so on, as stressed in the following proposition.

Proposition 4.

- *If node t receives a message, m, and $m.type =$ "REQ", we have that $m.src \in t[\mathcal{D}]$.*
- *If node t receives a message, m, and $m.type =$ "RPLY", we have that $m.src = t[\mathcal{P}]$.*

Proof.

- From Algo 6, Lines 11-14, we have that request messages are only forwarded to parent nodes (i.e., $m.src \in t[\mathcal{S}]$) and from Algo 5, Line 13, we have that agents always send messages to their parent nodes (i.e., $m.src \in t[\mathcal{A}]$). From Definition 17, we have that $\mathcal{D} = \mathcal{A} \cup \mathcal{S}$ as required.

- From Algo 6, Lines 4-11, we have the replies are sent to the requesters which are always the children of a node as proved in the first statement and from Lines 16-19, we have that reply messages are only forwarded to the last address in route of the message which only contains the address of requesters as shown in Line 12, Algo 6 and also Line 12, Algo 5. Again by the proof of the first statement, we know that requesters are always children of a node (i.e., $m.src \in n[\mathcal{P}]$) as required.

□

The rest of the section is dedicated to show that our coordination tree infrastructure works as expected. In Proposition 5, we prove that if any node in the infrastructure sends a request for a fresh id, it will eventually get it. In Proposition 6, we prove that for any two nodes in the infrastructure with different expected next id (*nid*), they will eventually converge to the same next id. This is important to show that messages do not circulate indefinitely in the infrastructure. Proposition 7 instead ensures the progress of the infrastructure in the sense that no data message stays in a node's queue indefinitely.

Proposition 5. *For each pair of nodes $t_1, t_2 \in \mathcal{T}$, if t_1 sends a request to t_2 then eventually t_2 will send a reply to t_1.*

Proof. The proof proceeds by induction on the level of n_2, $L(n_2)$.

- Base case: $L(t_2) = 0$, this implies that node t_2 is the root of the infrastructure. By Algo 6, Lines 11-14, when t_1 receives a request, it adds its address to the route of the message and forwards it its parent, in this case t_2. By Algo 6, Lines 5-11, we have that when the root receives a request message, it creates a reply message, carrying a fresh id, and sends it to the source where the request came from. So we have that t_2 will eventually send a reply back to t_1.

- Suppose that $\forall t_2 : L(t_2) \leq k$, if t_1 sends a request to t_2, then t_2 will eventually send a reply to t_1.

  Now it is sufficient to prove the claim for n_2, where $L(n_2) = k+1 \wedge k > 0$. By Algo 6, Lines 11-14, t_1 can send a request to t_2. On the other hand t_2 can also forward the request to its parent node, say t' where $L(t') = k$, by Algo 6, Lines 11-14. By induction hypothesis t' will eventually send a reply to t_2 since the claim holds for level k. When the reply arrives to t_2, it will eventually send it to t_1 by using Algo 6, Lines 16-19, as required.

□

The following lemma ensures that adjacent nodes will eventually converge to the same *nid*. In other words, a node must forward the removed data message from its queue to all immediate neighbors (i.e., tree nodes in its $\mathcal{D}$) before incrementing its *nid*.

Lemma 6.1. *For every two tree nodes t_1 and t_2 and a tree-based infrastructure $\mathcal{T}$ such that $t_1, t_2 \in \mathcal{T}$, we have that:*

- *If $t_1 \prec t_2 \wedge\ t_1[nid] < t_2[nid]$ then $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$.*

- *If $t_2 \prec t_1 \wedge\ t_1[nid] < t_2[nid]$ then $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$.*

Proof. The proof proceed by induction on the difference between $t_2[nid]$ and $t_1[nid]$. We only prove the first statement as the second one is analogous.

- Base case, $t_2[nid] - t_1[nid] = 1$: From Algo 6, Lines 29-35, *nid* is only incremented after message m, where $m.id = t_2[nid] - 1$, is forwarded to children in $\mathcal{D}$ (Lines 33-34) and/or parent, $\mathcal{P}$ (Lines 20-27). So we know that t_1 already received m and added m to its queue by (Lines 20-27), i.e., $t_1[\mathcal{W}] = m :: q$. Notice that this is a priority queue that sorts its messages according to their identities. Since $t_2[nid] - t_1[nid] = 1$, we have that $t_2[nid] - 1 = t_1[nid] = m.id$. This means that m is ordered with respect to t_1 and by Algo 6, Lines 29-35, we have that $t'[nid] = t_1[nid] + 1 = t_2[nid] \wedge t'[addr] = t_1[addr]$ as required.

- Suppose that $\forall t_1[nid], t_2[nid] : t_2[nid] - t_1[nid] \leq k$ where $k > 1$ and given that $t_1 \prec t_2$, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$.

  Now it is sufficient to prove the claim for t_1 and t_2 such that $t_2[nid] - t_1[nid] = k + 1$ where $k > 1$.

  From Algo 6, Lines 29-35, we know that message m, where $m.id = t_2[nid] - 1$, has been already forwarded to children in $\mathcal{D}$ and/or to parent, P (Lines 20-27) and t_1 already received m and added m to its queue, i.e., $t_1[\mathcal{W}] = m :: q$, by Algo 6, Lines 29-35. Since $t_2[nid] - t_1[nid] = k+1$, we have that $m.id - t_1[nid] = k$. This means that k-messages from t_2 already exist in the queue of t_1 and need to be processed first and then after m can be processed. By induction hypothesis, it holds that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$ where $t_2[nid] - t_1[nid] \leq k$ and $k > 1$. So we have that $t'[nid] = t_1[nid] + k$, but $m.id - t_1[nid] = k$. This implies that $m.id = t'[nid]$ which means that message m is ordered with respect to t'. Now again by Algo 6, Lines 29-35, $\mathcal{T}' \Rightarrow \mathcal{T}''$, $t'' \in \mathcal{T}'' : t''[addr] = t'[addr] \wedge t''[nid] = t'[nid] + 1 = t_2[nid] = t_1[nid] + k + 1$ and $t_2[nid] = t''[nid] \wedge t''[addr] = t_1[addr]$ as required.

□

Proposition 6. *Let t_1 and t_2 be two tree nodes and $\mathcal{T}$ be a tree-based infrastructure, $\forall t_1, t_2 \in N \wedge\ t_1[nid] < t_2[nid]$, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$.*

Proof. Since the topology of the infrastructure is tree-based, we have three cases.

- Case 1, $t_1 \prec^+ t_2$: This case can be proved by induction on the distance, $d(t_1, t_2)$, between t_1 and t_2 in the tree. Function $d(t_1, t_2)$ is defined inductively as follows:

$$d(t_1, t_2) = \begin{cases} 0, & \text{for } t_1 \prec t_2 \text{ or } t_2 \prec t_1 \\ 1 + d(t', t_2), & \text{for } t_1 \prec^+ t_2 \text{ where } t_1 \prec t' \\ 1 + d(t_1, t'), & \text{for } t_2 \prec^+ t_1 \text{ where } t_2 \prec t' \end{cases}$$

  - Base case, $d(t_1, t_2) = 0$: directly from Lemma 6.1.
  - Suppose that $\forall t_1, t_2 \in \mathcal{T} : d(t_1, t_2) \leq k$ where $k > 0$ and given $t_1[nid] < t_2[nid]$, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' :$

$t'[addr] = t_1[addr] \wedge t'[nid] = t_2[nid]$. Now it is sufficient to prove the claim for t_1 and t_2 where $d(t_1, t_2) = k+1$ and $k > 1$. From Lemma 6.1 and for t_2 at distance $k+1$ and t_3 at distance k from t_1, i.e., $t_1 \prec^+ t_3 \prec t_2$, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_3[addr] \wedge t'[nid] = t_2[nid]$. But $d(t_1, t') = k$, so by induction hypothesis we have that $\mathcal{T}' \Rightarrow \mathcal{T}''$ and $\exists t'' \in \mathcal{T}'' : t''[addr] = t_1[addr] \wedge t''[nid] = t'[nid] = t_2[nid]$ as required.

- Case 2, $t_2 \prec^+ t_1$: is analogous to the previous case.
- Case 3, $\exists t_3 : t_1 \prec^+ t_3 \wedge\ t_2 \prec^+ t_3$: we have several cases, but here we only consider one case and the others follow in a similar way.

  If $t_1[nid] > t_3[nid] < t_2[nid]$, we first take t_3 and t_2 and by Case 2, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and $\exists t' \in \mathcal{T}' : t'[addr] = t_3[addr] \wedge t'[nid] = t_2[nid] > t_1[nid]$. Now for t_1 and t' and by Case 1, we have that $\mathcal{T}' \Rightarrow \mathcal{T}''$ and $\exists t'' \in \mathcal{T}'' : t''[addr] = t_1[addr] \wedge t''[nid] = t'[nid] = t_2[nid]$ as required.

□

Proposition 7 (Progress)**.** *For any node $t \in \mathcal{T}$ and $t[\mathcal{W}] = m :: q'$, we have that $\mathcal{T} \Rightarrow \mathcal{T}'$ and there exists $t' \in \mathcal{T}' :\ t[addr] = t'[addr]$ and $t'[\mathcal{W}] = q' :: q$".*

Proof. The proof follows from Proposition 5 and Proposition 6 and by induction on the difference between $t[nid]$ and $m.id$ where $m = \mathcal{W}.top()$. □

Theorem 6.3. *Any data message is eventually received by all agents.*

Proof. The proof follows directly from Proposition 5, Proposition 6 and Proposition 7. □

6.5 Performance Evaluation

To compare the three coordination infrastructures presented in the previous section, we model them in terms of a Continuous Time Markov Process (And12). The state of a process represents possible system configurations, while the transitions (that are selected probabilistically) are associated with the events that let the infrastructure progress.

In our system we can consider three kind of events: a new message *sent* by an agent; a message *transmitted* from a node to another in the infrastructure; a message locally *handled* by a node (i.e. removed from an input/waiting queue). Each event is associated with a *rate* that is the parameter of the *exponentially distributed* random variable governing the *event duration*. To perform the simulation we need to fix three parameters: the *agent sending rate* λ_s; the *infrastructure transmission rate* λ_t; and the *handling rate* λ_h. In all of our experiments we fix the values of these parameters as follows: $\lambda_s = 1.0$, $\lambda_t = 15.0$, and $\lambda_h = 1000.0$. As usual we rely on kinetic Monte Carlo simulation (Sch08). In this section, the following notations are used to denote system configurations:

- $C[x, y]$, indicates a *cluster infrastructure* composed by x nodes and y agents;
- $R[x, y]$ indicates a *ring infrastructure* composed by x nodes each of which manages y agents;
- $T[x, y, z]$ indicates a *tree infrastructure* composed by x levels. Each node in the tree (but the leafs) has $y + z$ children: y nodes and z agents. The leaf nodes have z agents each.

Since it is very difficult, if not impossible, to perform statistical analysis on *real* distributed systems when the number of involved participants is large, like in the scenarios considered in this section, we model our infrastructures as Markov processes and evaluate their performance. We consider two scenarios: The first one, named *communication intensive* (CI), is used to estimate the performance of the infrastructures when a *large* number of messages are exchanged among agents. All the involved agents send messages continuously with a fixed rate and the scenario is used to assess the infrastructure performance when it is overloaded.

The second scenario we consider is based on a *fixed number of data providers* (DP): only a fraction of agents sends messages. This scenario models a more realistic configuration where data are only sent by specific agents that, for instance, may acquire these data via sensors from the environment where they operate. Following a typical pattern in CAS,

this data are used by other agent to adapt their behavior. An example could be a *Traffic Control System* where *data providers* are specific devices located in the city, while the *data receivers* are the vehicles traveling in the area.

For each scenario we consider two kind of measures: the time needed by a message to reach all the agents in the system and the message time gap. The first measure indicates how old is a message, while the latter indicates the time between two different messages received by a single agent.

Communication intensive scenario

We consider systems composed by 155 agents that continuously send messages to all the other receivers. Simulations are performed by considering the following configurations:[3]

- Cluster-based infrastructure: $C[10, 155]$, $C[20, 155]$ and $C[31, 155]$;
- Ring-based infrastructure: $R[5, 31]$ and $R[31, 5]$;
- Tree-based infrastructure: $T[5, 2, 5]$ and $T[3, 5, 5]$.

Among the three approaches considered in the previous section, the *cluster-based* is the one with worst performance. This can be appreciated by considering the results in Fig. 8 where the average time (together with the confidence interval) needed by a message to reach all the agents in the system is reported. One can easily notice that when the system reaches the equilibrium (at around time 2000), about 600 time units are needed in order to deliver a message to all agents. These results are not surprising. Indeed, on average, an agent has to wait that all other precedent messages are delivered before it can deliver its own.

We can also observe that the number of nodes in the cluster has a minimal impact on this measure. It is because an agent will send a message only when all the previous ones are received. This aspect can be understood if we consider *Average Message Time Gap* experienced by agents,

[3]The simulator is available at `https://github.com/lazkany/AbCSimulator`

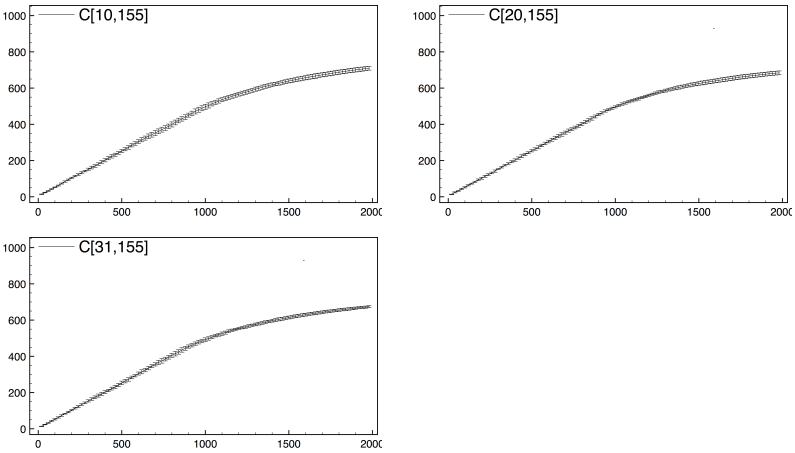

Figure 8: Cluster: Average Delivery Time (155 agents with 10, 20 and 31 cluster elements).

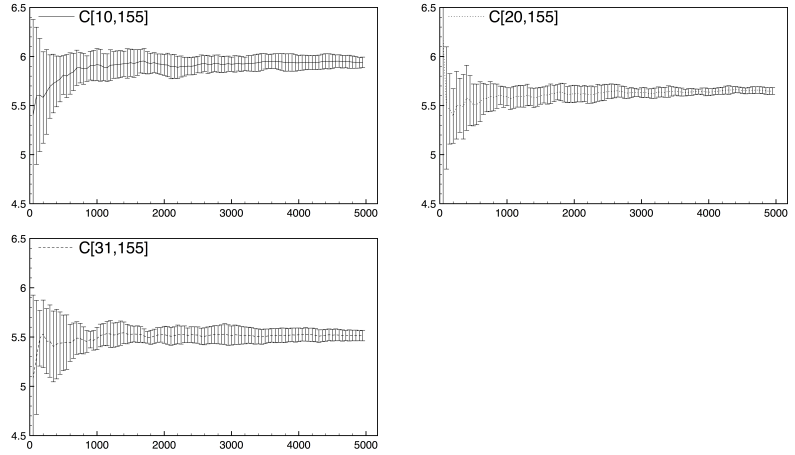

Figure 9: Cluster: Average Message Time Gap (155 agents with 10, 20 and 31 cluster elements).

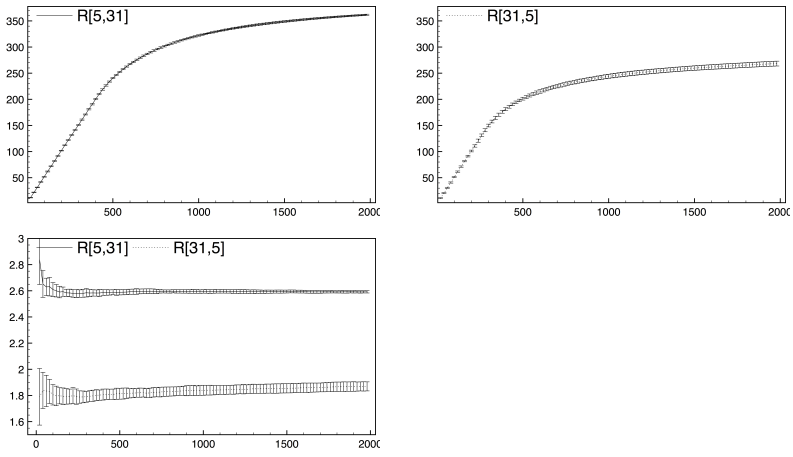

Figure 10: Ring: Average Delivery Time and Average Message Time Gap.

reported in Fig. 9. We can observe that in the long run an agent receives a message every 6/5.5 time units.

Better performance can be obtained if the ring infrastructure is used. In the first two plots of Fig. 10 we report the average delivery time for the configurations $R[5, 31]$ and $R[31, 5]$. The last plot compare the average message time gap of the two configurations. We can observe that in the first configuration a message is delivered to all the agents in 350 time units. In the second configuration this value decreases to 250 time units. This is because in a ring-based infrastructure all the nodes cooperate to deliver the same message to all agents. This affects also the *frequency* of messages delivered to agents. Indeed, in the considered infrastructures a message is received every 2.6 and 1.8 time-units.

The best results are obtained by the tree-based infrastructures. In Fig. 11 we report the how the *average delivery time* changes during the simulation for $T[5, 2, 5]$ and $T[3, 5, 5]$. The two configurations have exactly the same number of nodes (31) with a different arrangement. We can observe that the two configurations work almost in the same way:

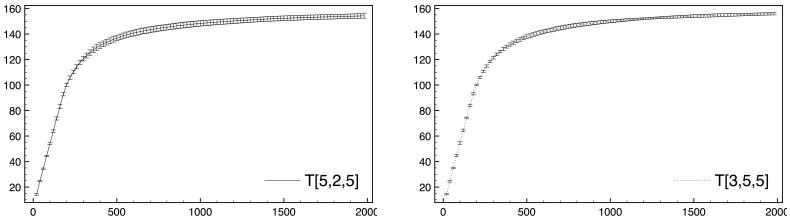

Figure 11: Average Delivery Time: $T[5, 2, 5]$ and $T[3, 5, 5]$.

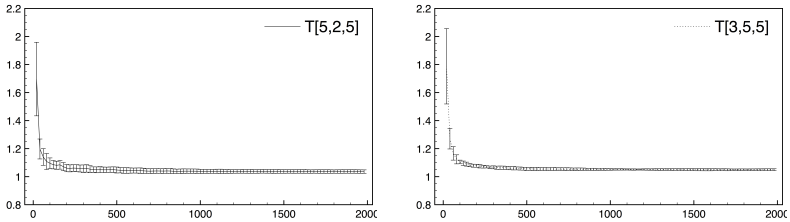

Figure 12: Average Message Time Gap: $T[5, 2, 5]$ and $T[3, 5, 5]$.

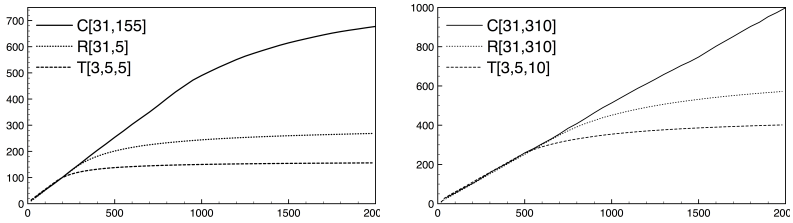

Figure 13: Cluster/Ring/Tree infrastructure results (155 agents, 310 agents).

a message is delivered to all the agents in about 120 time units. This means that, with the same number of *nodes*, the tree-based infrastructure is 5-time faster than the cluster-based and 2-time faster than ring-based infrastructure. Moreover, in the tree-based approach, a message is delivered to agents every $\sim$ 1.1 time units (see Fig. 12. This means that messages in the tree-based infrastructure are constantly delivered after an initial delay.

Simulation results are summarized in the left side of Fig. 13. We can observe that in this *communication intensive* scenario, tree-based infrastructures guarantee better performance; cluster-based infrastructures are overloaded and do not work well while ring-based are in between the two. We can observe that these differences are highlighted when increasing the number of agents to 310 (right side of Fig. 13).

Data provider scenario

In this section we evaluate the proposed infrastructures in a scenario where only a fraction of agents in the system sends data. We consider configurations where the number of nodes is the same (31) while we the number of agents be 155, 310 and 620. We assume that only 10% of agents are sending data.

The average delivery time in the simulated system is reported in Fig. 14 while in Fig. 15 the average message time gap is depicted. We can observe that even when the system is not overloaded, the tree infrastructure is the

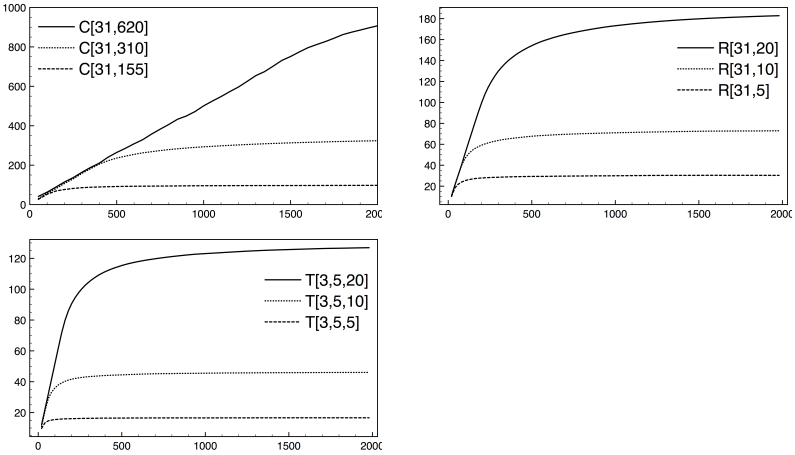

Figure 14: Data provider scenario: Average Delivery Time.

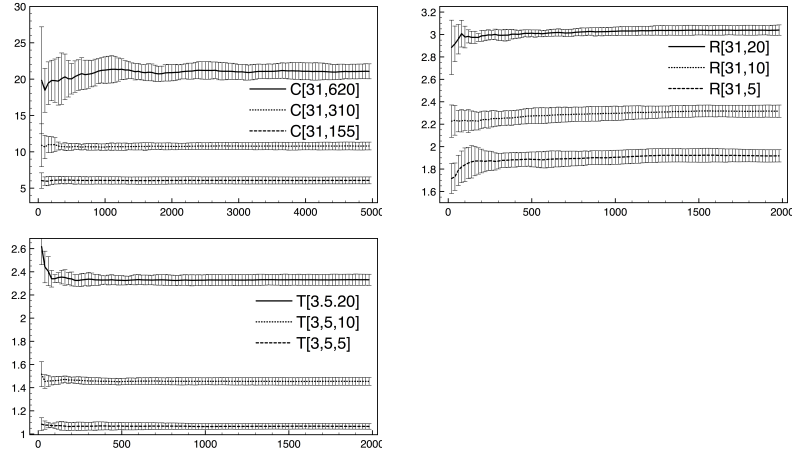

Figure 15: Data provider scenario: Average Message Time Gap.

one with the best performance while the cluster-based infrastructure is the worst. We can also observe that performance of ring-based infrastructure in this scenario is similar to the one obtained in the tree based. Moreover, in the cluster-based approach, the performance degrades soon as the number of agents increases. This does not happen when tree- and ring-based approaches are used. Finally, we can observe that, like in the CI scenario, also in this case messages are delivered more frequently in the ring- and tree-based approach than in the cluster-based.

6.6 A Scalable and relaxed abstract machine for *AbC*

The implementations, presented in the previous section, consider a specific communication infrastructure where servers are structured in either cluster-based, ring-based, or tree-based topology which makes these implementations more suitable for moderate sized systems distributed in a moderate sized geographic area. If we consider systems connected through the Internet, these implementations will not be applicable as the assumptions about the topology do not hold anymore. In the presence of large sized systems, e.g., Internet, we cannot achieve a global ordering of messages in a practical way so it might be useful to consider a notion of "site" in which we require the messages to be ordered. When messages propagate from one site to another through the internet, there is no need to worry about the order in which messages are received. Messages can be received in any order and it is not necessary to be sound with respect to the interleaving semantics since the time interval between sending and receiving can be very long and the atomicity of message exchange cannot be achieved without synchronization points which are not applicable in this context. A graphical representation of the infrastructure that we have in mind is reported in Figure 16.

The basic idea is to extend the tree-based implementation in the previous section in a way that makes it more suitable for other kind of applications. We want to stress that we are interested in the attribute-based interaction whether it is based on point-to-point, multiway synchroniza-

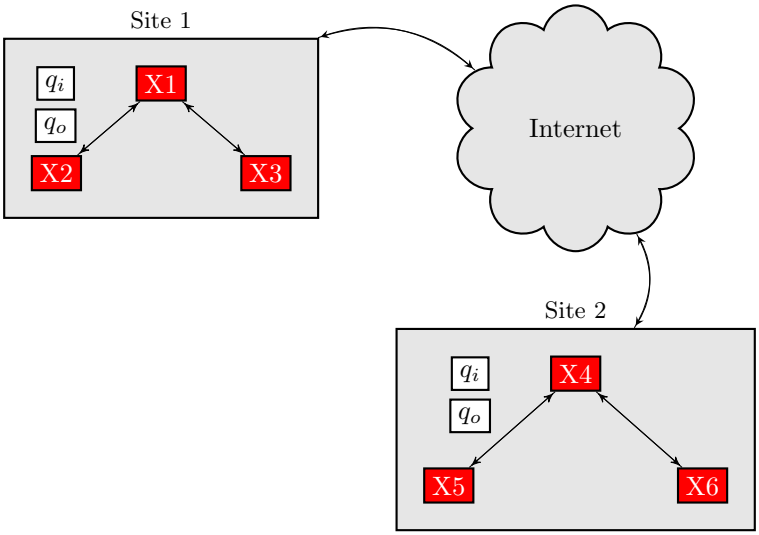

Figure 16: Asynchronous decentralized infrastructure

tion, or multicast. Our choice, influenced by the application domain of interest, was to consider atomic multicast communication but other communication models can be considered too where communication contracts are made in attribute-based fashion. As shown in Figure 16 the internal communication infrastructure of each site is structured in a tree-based topology where the main server is the one which is responsible for communicating messages to other sites. Each site has an input queue which is used to store received messages from other sites and an output queue which is used to store processed data messages to be communicated to other sites. The coordination algorithm in each site is similar to the one we discussed in the previous section with the exception that the root server is also responsible for placing every processed data message in the output queue so that it is communicated later to other sites through the Internet and also for retrieving data messages from the input queue. It should be noted that retrieved data messages are also assigned fresh ids by the root server and are forwarded to other local servers accordingly.

We define a formal abstract machine that specifies the behavior of the communication infrastructure discussed above. Since the internal behavior of each site is sound with respect to the operational semantics of the AbC calculus, its transition rules will be derived by the transition relation $\longrightarrow$ that defines the behavior of an AbC system as reported in Table 6, Page 44. We use the reduction relation $\rightsquigarrow \subseteq Site \times Site$ to define the behavior of the abstract machine where $Site$ denotes the set of sites. The definition of the reduction relation $\rightsquigarrow$ depends on the definition of the relation $\longrightarrow$ in the sense that the effect of site internal behavior is lifted to the global one. We use the notation $\{C\}^{\langle i,\ o\rangle}$ to define an AbC site where C denotes an AbC system operating in a tree-based communication infrastructure, i denotes an input queue, and o denotes an output queue. To simplify the presentation we consider C as a multi-set of components.

Rule **LBrd** states that when a component $C'' \in C$, located at a specific site, sends a message, the root server processes the message and adds it at the end of the output queue to be propagated later to other sites. The internal system C evolves to C' as an effect of this transition.

$$\textbf{LBrd}\ \frac{C \xrightarrow{\overline{\Pi}\tilde{v}} C'}{\{C\}^{\langle i,\ o\rangle} \rightsquigarrow \{C'\}^{\langle i,\ o::\overline{\Pi}\tilde{v}\rangle}}$$

Rule **LRcv** states that if the input queue of a site is not empty, the root server of the site removes the message from the head of the input queue and propagates it to other servers and components in the site. Of course the root server will assign the message a fresh id before propagation, but here we refrain from going deep in details and only provide an abstract view of the transitions. The internal system C evolves to C' as an effect of this transition.

$$\textbf{LRcv}\ \frac{C \xrightarrow{\Pi(\tilde{v})} C'}{\{C\}^{\langle \overline{\Pi}\tilde{v}::i,\ o\rangle} \rightsquigarrow \{C'\}^{\langle i,\ o\rangle}}$$

Rule **OUT** states that if the output queue of a site is not empty, the message at the head of the queue is removed and propagated to other sites. This transition has no effect on the internal system C.

$$\textbf{OUT}\ \{C\}^{\langle i,\ \overline{\Pi}\tilde{v}::o\rangle} \rightsquigarrow \{C\}^{\langle i,\ o\rangle}$$

Rule **IN** states that if a message is received from another site, the message is added at the end of the input queue and the internal system C stays unchanged.

$$\textbf{IN}\ \{C\}^{\langle i,\ o\rangle} \rightsquigarrow\!\!\triangleright\ \{C\}^{\langle i::\overline{\Pi}\tilde{v},\ o\rangle}$$

Rule **Silent** models the internal computation of a single site.

$$\textbf{Silent}\ \frac{C \xrightarrow{\tau}\ C'}{\{C\}^{\langle i,\ o\rangle} \rightsquigarrow\!\!\triangleright\ \{C'\}^{\langle i,\ o\rangle}}$$

Rule **sCom** model the parallel composition of different sites where if the output queue of one site is not empty, the head of the queue is removed and propagated to other sites executing in parallel. The other sites add the received message at the end of their queues to be processed later.

$$\textbf{sCom}\ \frac{\{C\}^{\langle i,\ \overline{\Pi}\tilde{v}::o\rangle} \rightsquigarrow\!\!\triangleright\ \{C\}^{\langle i,\ o\rangle} \qquad \forall j \in J\ (\{C_j\}^{\langle i_j,\ o_j\rangle} \rightsquigarrow\!\!\triangleright\ \{C_j\}^{\langle i_j::\overline{\Pi}\tilde{v},\ o_j\rangle})}{\{C\}^{\langle i,\ \overline{\Pi}\tilde{v}::o\rangle} \parallel \prod_{j\in J}\{C_j\}^{\langle i_j,\ o_j\rangle} \rightsquigarrow\!\!\triangleright\ \{C\}^{\langle i,\ o\rangle} \parallel \prod_{j\in J}\{C_j\}^{\langle i_j::\overline{\Pi}\tilde{v},\ o_j\rangle}}$$

Remark 6.1 (The order of message delivery is not preserved)**.** *Let C_1 and C_2 be two components and $\{\bullet\}^{\langle i,\ o\rangle}$ is a site with input queue i and output queue o, where $C_1 = \Gamma_1 : (v)@\Pi_1.(x = w)(x).0$ and $C_2 = \Gamma_2 : (w)@\Pi_2.(x = v)(x).0$. Assume that $\Gamma_1 \models \Pi_2$ and $\Gamma_2 \models \Pi_1$ and the queues i and o are initially empty.*

- For $C_1 \| C_2$, the possible outcomes according to the interleaving semantics would be either $\Gamma_1 : 0 \parallel \Gamma_2 : (x = v)(x).0$, when executing the sequence $\xrightarrow{\overline{\Pi_1}v}\xrightarrow{\overline{\Pi_2}w}$, or $\Gamma_1 : (x = w)(x).0 \parallel \Gamma_2 : 0$, when executing the sequence $\xrightarrow{\overline{\Pi_2}w}\xrightarrow{\overline{\Pi_1}v}$.

- For $\{C_1\}^{\langle i,\ o\rangle} \| \{C_2\}^{\langle i,\ o\rangle}$, the possible outcomes according to the abstract machine also include $\{\Gamma_1 : 0\}^{\langle \emptyset,\ \emptyset\rangle} \parallel \{\Gamma_2 : 0\}^{\langle \emptyset,\ \emptyset\rangle}$ as shown

below:

$$\{\Gamma_1 : (v)@\Pi_1.(x = w)(x).0\}^{\langle \emptyset,\ \emptyset \rangle} \| \{\Gamma_2 : (w)@\Pi_2.(x = v)(x).0\}^{\langle \emptyset,\ \emptyset \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : (x = w)(x).0\}^{\langle \emptyset,\ \overline{\Pi_1}\tilde{v} \rangle} \parallel \{\Gamma_2 : (w)@\Pi_2.(x = v)(x).0\}^{\langle \emptyset,\ \emptyset \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : (x = w)(x).0\}^{\langle \emptyset,\ \overline{\Pi_1}v \rangle} \parallel \{\Gamma_2 : (x = v)(x).0\}^{\langle \emptyset,\ \overline{\Pi_2}w \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : (x = w)(x).0\}^{\langle \emptyset,\ \emptyset \rangle} \parallel \{\Gamma_2 : (x = v)(x).0\}^{\langle \overline{\Pi_1}v,\ \overline{\Pi_2}w \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : (x = w)(x).0\}^{\langle \overline{\Pi_2}w,\ \emptyset \rangle} \parallel \{\Gamma_2 : (x = v)(x).0\}^{\langle \overline{\Pi_1}v,\ \emptyset \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : 0\}^{\langle \emptyset,\ \emptyset \rangle} \parallel \{\Gamma_2 : (x = v)(x).0\}^{\langle \overline{\Pi_1}v,\ \emptyset \rangle}$$

$$\rightsquigarrow \quad \{\Gamma_1 : 0\}^{\langle \emptyset,\ \emptyset \rangle} \parallel \{\Gamma_2 : 0\}^{\langle \emptyset,\ \emptyset \rangle}$$

Chapter 7

Related Works

In this chapter, we touch on related works concerning languages and calculi with primitives that model either collective interaction, constraint-based interaction, or multiparty interaction. We also report on well-known existing approaches for modeling interaction in distributed systems and show how they relate to AbC. At the end of this chapter we discuss the main differences between AbC and its early version, presented in (ADL$^+$15).

7.1 Channel-based interaction

Channel-based interaction assumes that communicating partners have to agree on specific channels or names to make the interaction possible. Initially process calculi, like CCS (Mil80) and CSP (Hoa78), which adopt channel-based interaction allowed interprocess communication via static structure of channels between processes. Mobility, which is one of the basic features of modern software systems, is hardly possible in these calculi. Robin Milner to developed π-calculus (MPW92) to handle this problem by allowing channels to be communicated during the interaction. This way process interfaces become more dynamic and can change at run-time.

As an example consider the behavior of a component, named C and

modeled in π-calculus as follows:

$$C \triangleq \nu b(\bar{b}a \mid b(x).x(y,z).P) \xrightarrow{\tau} \nu ba(y,z).P$$

Component C changes its interface locally by communicating along a private channel "b" and uses the received channel "a" to communicate with its peers. Here, changing the interface locally requires explicit message-passing between processes, defining the local component behavior, and restricting names to avoid interference with external components. It should be noted that component C can receive a message from its peers on channel "a" only when its input action is enabled, i.e., "a" is received and the substitution $(x(y,z).P)[a/x]$ is applied, otherwise it will discard any incoming messages. This implies that the dynamicity of process interfaces is still limited in the sense that even if we allow generic (bound) input or output actions, these actions are disabled until they are instantiated with specific channel names. This means that a process is only willing to engage in communication when its actions are enabled. In AbC actions are always enabled with respect to the current attribute values of the component where they are executing. Once these values change, the interaction predicates change seamlessly and become available for other communication partners. Name restriction is not needed at the level of processes to model locality since processes can communicate through the shared attribute environment. In this way interdependence between co-located processes can be achieved by changing the shared attribute values at run-time.

A possible rendering of the above example in AbC would be as follows:

$$C = \Gamma : ([\texttt{this}.x := a]()@\texttt{ff}.0 \mid (x = \texttt{this}.x)(x,y).Q)$$

Assume that the initial value of attribute x in Γ is "c". The process on the left hand side of the interleaving operator "|" takes a silent move and updates the value of "x" into "a" and this will implicitly change the receiving predicate of the process on the right hand side. Notice that the difference here is that the input action is always enabled. Before the attribute update, an input action was possible on the predicate $(x = c)$ and after that, it is possible on the predicate $(x = a)$.

7.2 Constraint- and attribute-based interaction

In this section, we discuss approaches that rely on attributes, patterns, or constraints for enforcing interaction.

SCEL (DFLP13; DLPT14) (Software Component Ensemble Language) is a kernel language that has been designed to support the programming of autonomic computing systems. This language relies on the notions of *autonomic components* representing the collective members, and *autonomic-component ensembles* representing collectives. Each component is equipped with an interface, consisting of a collection of attributes, describing its features. Attributes are used by components to dynamically organize themselves into ensembles and as a way to select partners for interaction. SCEL has inspired the development of the core calculus *AbC* (ADL$^+$15; ADL16a) to study the impact of attribute-based communication. Compared with SCEL, the knowledge representation in *AbC* is abstract and is not designed for detailed reasoning during the model evolution. This reflects the different objectives of SCEL and *AbC*. While SCEL focuses on programming issues, *AbC* concentrates on a minimal set of primitives to study attribute-based communication.

The CPC calculus (GWGJ10) adopts the pattern-matching mechanism of the pure pattern calculus (JK06) for concurrent processes. The input and output prefixes are generalized to patterns whose unification enable a two-way, or symmetric, flow of information. The idea is to find a compatible process by matching inputs with outputs and testing for equality. Interaction is driven by unification that allows two processes to exchange information. The main differences with *AbC* calculus are the followings: first, the way processes agree to communicate (i.e., *AbC* uses predicates over attributes rather than exposing patterns in the input/output prefixes); second, the information propagation in *AbC* is mono direction (i.e., from the output to the input process); finally, *AbC* adopts multicast rather than point-to-point communication.

The attributed pi calculus (JLNU08; JLNU10) is an extension of the π-calculus (MPW92) that supports a specific kind of attribute-based com-

munication, and was designed primarily with biological applications in mind. As with AbC, processes may have attributes and these are used to select partners for interaction, but communication is strictly synchronization based and binary. The calculus is equipped with both a deterministic and a Markovian semantics, and in the Markovian case the rates may depend on the values of the attributes involved. The possible attribute values are defined by a language $\mathcal{L}$, and the definition of the attributed pi calculus is parameterized with respect to $\mathcal{L}$. The language $\mathcal{L}$ is also used to model the possible rates and the constraints that can be applied to attributes, thus offering the possibility to capture different behaviors within the framework when rates and probabilities of interaction are all dependent only on local behavior and knowledge. Processes are the top-level entities of the calculus and share the same global store (i.e., attribute environment) ρ , thus processes can communicate through message passing while sharing a global store. The global store cannot be changed at run-time and is only used to evaluate communication constraints on attribute values. This is important to enable only processes located in the same compartment to communicate.

To better illustrate the main differences with respect to AbC we present the main communication rule of the attributed pi calculus and then we show a typical attributed pi calculus scenario from (JLNU08):

$$\frac{\rho \vdash e_1 e_2 \Downarrow r \in Succ}{\rho \vdash x[e_1]?\tilde{y}.P_1 + \ldots \mid x[e_2]!\tilde{v}.P_2 + \ldots \rightarrow P_1[\tilde{v}/\tilde{y}] \mid P_2}$$

The reduction rule above shows how an attributed pi calculus system evolves. The rule matches a receiver $x[e_1]?\tilde{y}.P_1$ and a sender $x[e_2]!\tilde{v}.P_2$ on the same channel x such that the lambda expression $e_1 e_2$ evaluates to some successful value $r \in Succ$ under the shared environment ρ. The set $Succ$ in the non-deterministic semantics is the set that contains a single element which is a binary one, i.e., $Succ = \{1\}$.

Example 7.1. *Consider the binding action of a protein* `Prot(x)` *of type* $x \in \{\text{`}A\text{`}, \text{`}B\text{`}\}$ *to an operator* `Op(y)` *as shown in Figure 17. The protein and the operator are defined as follows:*

$$\texttt{Prot(x)} \triangleq \texttt{bind[x]!().0}$$

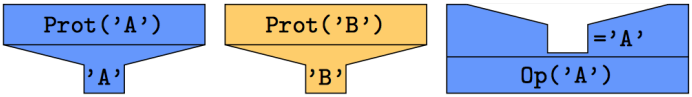

Figure 17: Protein of type 'A' is only permitted to bind to an operator of type 'A'. Type equality is tested by applying the lambda function $\lambda x.x =$ 'A'

$$Op(y) \triangleq bind[\lambda x.x{=}y]?().OpBound(y)$$

This means that the binding action of a protein is only enabled when the value of attribute x *is evaluated. On the other hand the operator* $Op(y)$ *receives the value of* x *and checks if it matches its own type. For instance a system with two proteins* $Prot('A')$ *and* $Prot('B')$ *and operator* $Op('A')$ *can evolves as follows:*

$$Prot('A') \mid Prot('B') \mid Op('A') \rightarrow Prot('B') \mid OpBound('A')$$

Now everything is in place to identify the main differences between AbC and the attributed pi calculus. The primary difference is that processes in AbC share the same attribute environment only when they reside in the same component. However, different components have different attribute environments. In the attributed pi calculus all processes share a common environment that allows them to communicate. As opposed to AbC, changing attribute values at run-time is not allowed in the attributed pi calculus. Also the receiver process is the only one which is responsible for deriving the interaction in the sense that the sender cannot specify the target of interaction. The sender evaluates its expression under the shared environment and sends the evaluation to the receiver which substitutes it in its receiving constraint to decide if the interaction is possible. In AbC, the interaction between senders and receivers is based on mutual interests in the sense that the sender can specify a possible group of receivers through the sending predicate and the receiver afterwards can decide whether the interaction is possible or not. It is worth mentioning that the interaction, in the attributed pi calculus, is still based on communication channels and the attributes are used for further filtering of

communication partners (e.g., only processes in the same compartment are allowed to communicate). Other differences are related to the fact that AbC supports implicit multicast with non-blocking message sending while the attributed pi calculus supports binary communication with fully synchronous communication.

The imperative π-calculus (JLN09) is a recent extension of the attribute π-calculus which allow imperative programming languages $\mathcal{L}$ to serve as attribute languages. In this way, the attribute π-calculus is enriched with a global imperative store. The attribute language has assignment expressions that can be used to change the values of channels at run-time, thus providing a fully dynamic interface. As opposed to AbC, the design choices, of selecting point-to-point synchronization mechanism and channels alongside with constraints as means for deriving the interaction, limit the applicability of this calculus to system biology.

7.3 Broadcast-based interaction

We now report on those calculi, tailored for broadcast and group communication.

CBS (OPT02; Pra95; Pra91) is probably the first process calculus to rely on broadcast rather than on channel-based communication. It captures the essential features of broadcast communication in a simple and natural way. Transmission of messages is autonomous, but reception is not. Whenever a process transmits a value, all processes running in parallel and ready to input catch the broadcast. The $b\pi$-calculus (EM99; EM01) is based on CBS and on the π-calculus (MPW92) in that it extends the former with a channel-passing mechanism. The AbC calculus inherits the style of communication of CBS, but differs from it in that the transmitted data can be names or values, and from $b\pi$-calculus in that it uses attributes rather than channels for communication. Moreover, AbC exploits the attributes attached to processes to select dynamically the set of destinations, and does not necessarily send a message to all listening processes.

The broadcast Quality Calculus of (VNR13) attacks the problem of

denial-of-service by means of *selective* input actions, where an input is consumed only if what is received matches what is expected. Such evaluation, however, is tailored to avoid flooding of improper messages, and therefore inspects the structure of messages to make sure that it agrees with the input contract. Coherently with its purpose, the calculus of (VNR13) only specifies contracts for inputs, and does not provide any means to change the input contracts during the execution, whereas attribute-based broadcast and attribute update are key features of AbC.

7.4 The Actor communication model

The Actor model, originally introduced in (HBS73; Hew77), has been used to support the development of object-based concurrent computation. Actors embody the spirit of objects and extends their advantages to concurrent computations. As with objects where data and behavior are encapsulated to separate what an object can do from how it does it, the actors separate the control (where and when) from the logic of a computation. The early proposals of the Actor model in (HBS73; Hew77) were rather informal. However, the definition of actors that is commonly used today follows Agha's definition in (Agh86). This definition provides an abstract representation for actor systems in terms of what is called *Asynchronous Computation Trees*, building on notions borrowed from Milner's work (Mil89). This representation provides a suitable way of visualizing computation in actors.

An actor system, also called a configuration, consists of autonomous objects, called actors, and of a collection of messages in transit. Messages, what Agha calls "communications", are the driving force of actors. Computations in an actor system are carried out in response to received messages. Each message contains the destination address and the actual content of the message. As opposed to the process algebra approach, only acceptance of a message is interpreted as a transition (Agh86). An actor has an internal state that can be modified externally only by sending it messages. The state is private and persistent and is made up of variables, which contain references to other actors. Each actor has a unique name,

and a unique mailbox address which does not change during the lifetime of the actor. The mailbox of an actor can be used by other actors to send it messages. An actor may not be known to all actors in the system. Messages can be sent only to those actors whose address is known. Furthermore actors are aware of their own mailbox addresses and can send messages to themselves. It should be noted that an actor might also receive addresses of other actors through the contents of incoming messages thus allowing mobility and dynamic configuration of the communication topology. This notion of mobility was not present in CCS (Mil80) and the Actor model was a source of inspiration in the development of the π-calculus as mentioned by Robin Milner in (Mil93; Fre93).

Message passing in actors is based on asynchronous point-to-point communication where message delivery is guaranteed but the arrival order of messages is indeterminate. The guarantee of message delivery relies on a fairness assumption. An actor may change its state, and may perform a finite number of the following actions: send a message to another actor whose mail address is known; create a new actor by relying on the existing behavior definitions while providing initial values for the parameters; become an actor which specifies the replacement behavior to come into effect when the next message is processed. It should be noted that the processing of the current message is not complete until the replacement is specified.

AbC follows an algebraic approach in specifying distributed systems. Process algebra and the actor model can be viewed as different schools for specifying concurrent and distributed systems. Actually the main differences between the two approaches stem from their different goals and objectives. The process algebra approach focuses on understanding elementary communication between processes by providing a rigorous algebraic formulation of distributed systems while abstracting from other programming aspects. On the other hand, the actor model approach focuses on explicitly modeling and programming distributed systems.

Several works for providing formal semantics of actors have been proposed so far (AMST92; AMST97; Tal97; Tal96; MT97). However, these papers concentrate on communication and concurrency aspects and do

not investigate the relationships of the actor approach with the process algebra approach.

The key distinguishing factors can be summarized as follows: Process calculi model the communication medium explicitly (i.e., they send and receive on shared channels) while in the actor model (and also in AbC) the communication medium is abstracted away. The communicable values in actors are actor names while in process calculi these are channels. In process calculi, any number of processes can send or receive on a given channel, and thus processes may interfere. This is not possible in the actor model because two actors cannot have the same name. Interference (i.e., non-deterministic selection of communication partner) is naturally avoided in the AbC calculus and one possible encoding of the name of a component is to consider it as an attribute. When a component wants to send a message to a specific component, the interaction predicate of the sending component can be used to select only the receiver with a specific name as we have seen in the smart conference scenario in Chapter 2, Specification 2.1 and Specification 2.2, where the participant sends its identity/name in the message and the room replies back only to the requester addressing him by his identity.

However, the non-deterministic selection of a communication partner is interesting and helps in achieving some sort of anonymity between communication partners, which is a desired property in distributed systems. In many cases one is interested in communicating with partners, which are capable of offering a specific service, even if one does not know the identity of the partner that is offering the service. This kind of behavior is hardly possible in the actor model since the selection is deterministic. In AbC the situation is different since it supports selective multicast communication. Thus only the group of partners, enjoying specific properties, can receive the message.

In process calculi, processes are stateless entities while actors have associated states. However, in AbC the attribute environment can be used to represent the internal state of a component. Finally, the actor model assumes a fair (reliable) message delivery, which is not the case for process calculi.

An interesting line of research on comparing the Actor model with the process algebra approach, more specifically with the asynchronous π-calculus (HT91), can be found in (TZA02). The authors defined a process calculus for the Actor model, called $A\pi$. The idea is to embed the Actor model within a typed asynchronous π-calculus where the syntax and semantics rules are not altered. The typing rules are then used to enforce properties specific to the Actor model. In this way, not only a direct basis for comparison with the π-calculus can be attained, but also it enables the possibility to adopt concepts and techniques developed for π-calculus to the Actor model.

The properties enforced by the type system include: the persistency property where actors do not dissolve after receiving messages; the freshness property where actor cannot be created with well-known names or names received in a message; the uniqueness property where composed actor configurations do not contain actors with the same name. It should be noted that the fairness property, where the delivery of a message can only be delayed for a finite but unbounded amount of time, is not considered in this calculus. The reason is that the paper is considering a May testing theory (NH84) for the $A\pi$ and compares it with the one introduced for the asynchronous π-calculus (BDP99b). The fairness property requires eventual delivery of messages while May testing is only concerned with the occurrence of an event after a finite computation. Thus, fairness only affects infinite computations and has no effects on the notion of May testing.

To enforce the above mentioned properties, the type system imposes a certain discipline on the use of names. Unlike the π-calculus where names are used to denote communication channels, the names in $A\pi$ denote unique actor names (i.e., the term $x(y).C$ represents a configuration with an actor x whose behavior is $(y)C$ and the term $\bar{x}y.0$ represents a configuration with a single message targeting actor x and with contents y). In this way, the interference resulting from a non-deterministic selection of the receiving component is ruled out in the sense that the receiving component/Actor is deterministically specified using its unique name. To ensure the freshness properties, terms like $x(y).y(z).0$ are not allowed.

New actors cannot be created with names received from a message. In the previous example the message y cannot be used to create a new actor. This means that the above term is not well-typed (i.e., not an $A\pi$ term). Following the terminology of (PS96), only output capability of names can be passed in messages (i.e., the term $x(y).\bar{y}z.0$ is well typed). To ensure uniqueness property, composed configurations like $C_1|C_2$ should not contain actors with the same name. It should be noted that a term like $x(y).0$ is considered well-typed (i.e., an $A\pi$ term). This term represents the actor x which dissolves after receiving the message y. However, if this term is interpreted as the actor x assuming a sink behavior, which simply consumes all messages it receives, the persistence property is not violated.

We conjecture that since the non-deterministic selection of the receiving component/actor is not required, the embedding of the Actor model in both the $b\pi$ and AbC calculi is possible in a similar way as proposed by $A\pi$. Although the communication rule in $b\pi$ and AbC considers multiparty settings, the rule should work as desired because different actors cannot have the same name. In other words, it does not matter if the communication is observed by other configurations as there is always a single unique receiver. However, it is required to rule out terms of the following form $\bar{x}y.C$ (resp. $(\upsilon)@\Pi.C$) where $C \neq 0$. Also acknowledgments of message reception are required as the semantics of output actions in both calculi is non-blocking.

The ActorSpace model (AC93) is a generalization of the actor model to support group-based communication where partners can be selected by pattern matching. This model supports both point-to-point and multicast communication. To the best of our knowledge, the ActorSpace model still lacks a formal semantics. We believe that AbC can be viewed, to some extent, as a generalization of the ActorSpace model. The idea is that AbC extends the ActorSpace pattern-matching mechanism to select partners by predicates on both sides of the communication where not only the sender can select its partner but also the receiver can decide to either receive or discard the message. The notion of spaces/collectives in AbC is more abstract and is only specified at run-time.

7.5 Other approaches for programming adaptive behavior

Programming collective and/or adaptive behavior has been studied in different research communities like those interested in context-oriented programming and in the component-based approach. In Context-Oriented Programming (COP) (HCN08), a set of linguistic constructs is used to define context-dependent behavioral variations. These variations are expressed as partial definitions of modules that can be overridden at runtime to adapt to contextual information. They can be grouped via layers to be activated or deactivated together dynamically. These layers can be also composed according to some scoping constructs. Our approach is different in that components adapt their behavior by considering the runtime changes of the values of their attributes which might be triggered by either contextual conditions or by local interaction. Another approach that considers behavioral variations by building on the Helena framework is considered in (Kla15).

The component-based approach, represented by FRACTAL (BCS04) and its Java implementation, JULIA (BCL$^+$06), is an architecture-based approach that achieves adaptation by defining systems that are able to adapt their configurations to the contextual conditions. System components are allowed to manipulate their internal structure by adding, removing, or modifying connectors. However, in this approach interaction is still based on explicit connectors. In our approach predefined connections simply do not exist: we do not assume a specific architecture or containment relations between components. The connectivity is always subject to change at any time by means of attribute updates. In our view, *AbC* is definitely more adequate when highly dynamic environments have to be considered.

7.6 The old *AbC* Calculus

The old AbC[1] is a very basic calculus and has many limitations. In this thesis, we have fully redesigned the calculus and added essential features needed to effectively control interactions in an attribute-based framework.

The old AbC does not support awareness since its components have no explicit way to read/check their attribute environments and react accordingly. This greatly impacts on expressiveness in that awareness-based applications can hardly be tackled. The old calculus has also problems in modeling adaptation; the impossibility of accessing the attribute environment implies that interaction predicates are static and cannot take into account the changing attributes. For instance, in the robotic scenario in Chapter 4, Section 4.3.3, if a collision with a wall in the arena is detected, the robot cannot adapt by changing its direction because the robot is not aware of its environment. The same applies for its own status e.g., its battery level, so modeling such a scenario in old AbC is hardly possible.

In the old calculus, the whole component state is always exposed during interaction in the sense that different messages sent by a component are characterized by the same set of attributes. A simple behavior like $(\bar{a}c + \bar{b}d) \parallel (a(x) + b(x))$– where action a (resp. b) synchronizes only with action a (resp. b) – cannot be modeled. This behavior can be simply modeled in AbC as follows:

$$\Gamma_1{:}(a,\ c)@(\mathtt{tt}) + (b,\ d)@(\mathtt{tt}) \quad \| \quad \Gamma_2{:}(x = a)(x,\ y) + (x = b)(x,\ y)$$

By including the channel names (i.e., a and b) in the message we are guaranteed that message c will only be received by the process with predicate $(x = a)$ and message d will only be received by the process with predicate $(x = b)$. Clearly, the absence of multi-value passing in the old calculus impacts its expressiveness and makes modeling channel-based communication very hard.

In the end of this chapter, we would like to refer to results from a survey of formal methods for supporting swarm/collective behavior, presented in (RTRH05). The results show that there does not exist a single

[1]The full formal definition can be found in Appendix A.3

formalism to support such kind of behavior, but different formalisms can be combined to reach this goal. Due to the minimality of process calculi, specifications can become large and therefore difficult to read and understand. Most process calculi cannot explicitly deal with data. They do not support modeling and reasoning about persistent information so adaptive behavior can be verified. The goal of AbC is to support modeling adaptive systems with the appropriate level of abstraction that permits a natural modeling and supports verification through compact models. Adaptation is guaranteed by introducing the attribute environment and its operations. However, quantitive variants of AbC are needed to answer questions about model dynamics and steady-state behaviors to ensure that a specific behavior will be reached.

AbC combines the lessons learnt from the above mentioned languages and calculi, in that it strives for expressiveness while aiming to preserve minimality and simplicity. The dynamic settings of attributes and the possibility of inspecting/modifying the environment gives AbC greater flexibility and expressiveness while keeping models as natural as possible.

Chapter 8

Concluding Remarks and Future Works

We have introduced a foundational process calculus, named AbC, for attribute-based communication. We investigated the expressive power of AbC both in terms of its ability to model scenarios featuring collaboration, reconfiguration, and adaptation and of its ability to encode channel-based communication and other interaction paradigms. We defined behavioral equivalences for AbC and finally we proved the correctness of the proposed encoding up to some reasonable equivalence. We demonstrated that the general concept of attribute-based communication can be exploited to provide a unifying framework to encompass different communication models and interaction patterns. We developed a prototype implementation for AbC linguistic primitives to demonstrate their simplicity and flexibility to accommodate different interaction patterns. We studied the impact of centralized and decentralized implementations of the underlying communication infrastructure that mediate the interaction between components.

We plan to investigate the impact of alternative behavioral relations like testing preorders in terms of equational laws, proof techniques, etc. We want to devise an appropriate notion of temporal logic that can be used to specify, verify, and monitor collective adaptive case studies, modeled in AbC. Actually since CAS components usually operate in an open and

changing environment, the spatial and temporal dimensions are strictly correlated and influence each other. So we would like to investigate the impact of spatio-temporal logic approaches in the context of *AbC* models. One promising approach is presented in (NB14).

We want to develop quantitive variants of *AbC* to consider other classes of systems and make reasoning and verification of large systems in *AbC* affordable. Further related work in this direction can be found in (BDG$^+$15), where a specification language was designed based on the *AbC* primitives to support quantitive analysis of large systems.

Another line of research is to investigate anonymity at the level of attribute identifiers. Clearly, *AbC* achieves dynamicity and openness in the distributed settings, which is an advantage compared to channel-based models. In our model, components are anonymous; however the "name-dependency" challenge arises at another level, that is, the level of attribute environments. In other words, the sender's predicate should be aware of the identifiers of receiver's attributes in order to explicitly use them. For instance, the sending predicate ($loc =< 1, 4 >$) targets the components at location $< 1, 4 >$. However, different components might use different identifiers names (i.e., "location") to denote their locations; this requires that there should be an agreement about the attribute identifiers used by the components. For this reason, appropriate mechanisms for handling *attribute directories* together with identifiers matching/correspondence will be considered. These mechanisms will be particularly useful when integrating heterogeneous applications.

Another research direction is to establish a static semantics for *AbC* as a way to discipline the interaction between components. This way we can answer questions regarding deadlock freedom and if the message payload is of the expected type of the receiver.

Our experience in modeling different case studies shows that point-to-point and group-based communication are not rival paradigms but they actually complement each other. As in the case of the ActorSpace model (AC93), we think that binary and multiway communication should be supported in any formalism that is tailored to model the interaction in distributed systems.

Appendix A

Appendix: Additional Materials

A.1 The completeness of the encoding

of Lemma 5.14. The proof proceeds by induction on the shortest transition of $\rightarrow_{b\pi}$. We have several cases depending on the structure of the term P.

- if $P \triangleq \mathtt{nil}$: This case is immediate $(\!|\mathtt{nil}|\!)_c \triangleq \emptyset : 0$
- if $P \triangleq \tau.G$: We have that $\tau.G \xrightarrow{\tau} G$ and it is translated to $(\!|\tau.G|\!)_c \triangleq \emptyset : ()@\mathsf{ff}.(\!|G|\!)_p$. We can only apply rule (Comp) to mimic this transition.

$$\frac{\emptyset : ()@\mathsf{ff}.(\!|G|\!)_p \xmapsto{\overline{\mathsf{ff}}()} \emptyset : (\!|G|\!)_p}{\emptyset : ()@\mathsf{ff}.(\!|G|\!)_p \xrightarrow{\overline{\mathsf{ff}}()} \emptyset : (\!|G|\!)_p}$$

  Now it is not hard to see that $(\!|\ G\ |\!)_c \ \simeq\ \emptyset : (\!|G|\!)_p$. They are even structural congruent. Notice that sending on a false predicate is not observable (i.e., a silent move).

- if $P \triangleq a(\tilde{x}).G$: We have that $a(\tilde{x}).G \xrightarrow{a(\tilde{z})} G[\tilde{z}/\tilde{x}]$ and it is translated to
  $(\!|a(\tilde{x}).Q|\!)_c \triangleq \emptyset : \Pi(y, \tilde{x}).(\!|G|\!)_p$ where $\Pi = (y = a)$. We can only apply rule (Comp) to mimic this transition.

$$\frac{\emptyset : \Pi(y,\tilde{x}).(\!|G|\!)_p \xmapsto{(a=a)(a,\ \tilde{z})} \emptyset : (\!|G|\!)_p[a/y,\ \tilde{z}/\tilde{x}]}{\emptyset : \Pi(y,\tilde{x}).(\!|G|\!)_p \xrightarrow{(a=a)(a,\ \tilde{z})} \emptyset : (\!|G|\!)_p[a/y,\ \tilde{z}/\tilde{x}]}$$

It is not hard to see that: $(\!|G[\tilde{z}/\tilde{x}]|\!)_c \ \simeq\ \emptyset : (\!|G|\!)_p[a/y,\ \tilde{z}/\tilde{x}] \simeq \emptyset : (\!|G|\!)_p[\tilde{z}/\tilde{x}]$ since $y \notin n((\!|G|\!)_p)$.

- if $P \triangleq \bar{a}\tilde{x}.G$: The proof is similar to the previous case but by applying this output transition instead.

- The fail rules for **nil**, τ, input and output are proved in a similar way but with applying (C-Fail) instead.

- if $P \triangleq \nu x Q$: We have that either $\nu x Q \xrightarrow{\gamma} \nu x Q'$, $\nu x Q \xrightarrow{\tau} \nu x \nu \tilde{y} Q'$ or $\nu x Q \xrightarrow{\nu x \nu \tilde{y} \bar{a} \tilde{z}} Q'$ and it is translated to $(\!|\nu x Q|\!)_c \triangleq \nu x \emptyset : (\!|Q|\!)_p$. We prove each case independently.

  - Case $\nu x Q \xrightarrow{\gamma} \nu x Q'$: By applying induction hypotheses on the premise $Q \xrightarrow{\gamma} Q'$, we have that $(\!|Q|\!)_c \twoheadrightarrow^* \simeq (\!|Q'|\!)_c$. We can only use rule (Res) to mimic transition depending on the performed action.

    $$\frac{\emptyset : (\!|Q|\!)_p[y/x] \xrightarrow{\gamma} \emptyset : (\!|Q'|\!)_p[y/x]}{\nu x \emptyset : (\!|Q|\!)_p \xrightarrow{\gamma} \nu y \emptyset : (\!|Q'|\!)_p[y/x]}$$

    And we have that $(\!|\nu x Q'|\!)_c \simeq \nu y \emptyset : (\!|Q'|\!)_p[y/x]$ as required.

  - Case $\nu a Q \xrightarrow{\tau} \nu a \nu \tilde{y} Q'$: By applying induction hypotheses on the premise $Q \xrightarrow{\nu \tilde{y} \bar{a} \tilde{z}} Q'$, we have that $(\!|Q|\!)_c \twoheadrightarrow^* \simeq (\!|Q'|\!)_c$. We can only use (Hide1) to mimic this transition.

    $$\frac{\emptyset : (\!|Q|\!)_p \xrightarrow{\nu \tilde{y} \overline{a=a}(a,\ \tilde{z})} \emptyset : (\!|Q'|\!)_p}{\nu a \emptyset : (\!|Q|\!)_p \xrightarrow{\nu \tilde{y} \overline{\mathsf{ff}}(a,\ \tilde{z})} \nu a \nu \tilde{y} \emptyset : (\!|Q'|\!)_p}$$

    We have that $(\!|\nu a \nu \tilde{y} Q'|\!)_c \simeq \nu x \nu \tilde{y} \emptyset : (\!|Q'|\!)_p$ as required.

  - Case $\nu x Q \xrightarrow{\nu x \nu \tilde{y} \bar{a} \tilde{z}} Q'$: follows in a similar way using rule (Open) .

  - Case $\nu x Q \xrightarrow{\alpha:}$: is similar to the case with (Res) rule.

- if $P \triangleq ((rec\ A\langle\tilde{x}\rangle).P)\langle\tilde{y}\rangle)$: This case is trivial.

- if $P \triangleq G_1 + G_2$: We have that either $G_1 + G_2 \xrightarrow{\alpha} G_1'$ or $G_1 + G_2 \xrightarrow{\alpha} G_2'$. We only consider the first case with $G_1 \xrightarrow{\alpha} G_1'$ and the other case follows in a similar way. This process is translated to $(\!|G_1 + G_2|\!)_c \triangleq \emptyset : (\!|G_1|\!)_p + (\!|G_2|\!)_p$. By applying induction hypotheses on the premise $G_1 \xrightarrow{\alpha} G_1'$, we have that $(\!|\ G_1\ |\!)_c \twoheadrightarrow^* \simeq (\!|\ G_1'\ |\!)_c$. We can apply either rule (Comp) or rule (C-Fail) (i.e., when discarding) to mimic this transition depending on the performed action. We consider the case of (Comp) only and the other case follows in a similar way.

$$\dfrac{\dfrac{\emptyset : (\!|G_1|\!)_p \overset{\lambda}{\mapsto} \emptyset : (\!|G_1'|\!)_p}{\emptyset : (\!|G_1|\!)_p + (\!|G_2|\!)_p \overset{\lambda}{\mapsto} \emptyset : (\!|G_1'|\!)_p}}{\emptyset : (\!|G_1|\!)_p + (\!|G_2|\!)_p \xrightarrow{\gamma} \emptyset : (\!|G_1'|\!)_p}$$

  Again $(\!|G_1'|\!)_c \simeq \emptyset :\ (\!|G_1'|\!)_p$

- if $P \triangleq P_1 \| P_2$: This process is translated to $(\!|\ P_1 \| P_2\ |\!)_c \triangleq \emptyset : (\!|\ P_1\ |\!)_p \| \emptyset : (\!|\ P_2\ |\!)_p$. We have four cases depending on the performed action in deriving the transition $P_1 \| P_2 \xrightarrow{\alpha} \hat{P}$.

  - $P_1 \| P_2 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_1' \| P_2'$: We have two cases, either $P_1 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_1'$ and $P_2 \xrightarrow{a(\tilde{x})} P_2'$ or $P_2 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_2'$ and $P_1 \xrightarrow{a(\tilde{x})} P_1'$. We only consider the first case and the other case follows in the same way. By applying induction hypotheses on the premises $P_1 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_1'$ and $P_2 \xrightarrow{a(\tilde{x})} P_2'$, we have that $(\!|P_1|\!)_c \twoheadrightarrow^* \simeq (\!|P_1'|\!)_c$ and $(\!|P_2|\!)_c \twoheadrightarrow^* \simeq (\!|P_2'|\!)_c$. We only can apply (Com).

$$\dfrac{\emptyset : (\!|P_1|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \quad \emptyset : (\!|P_2'|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_2'|\!)_p}{\emptyset : (\!|P_1|\!)_p \ \|\ \emptyset : (\!|P_2|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \ \|\ \emptyset : (\!|P_2'|\!)_p}$$

    Again we have that: $(\!|P_1'\|P_2'|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2'|\!)_p$. Notice that the $b\pi$ term and its encoding have the same observable behavior i.e., $P_1 \| P_2 \downarrow_a$ and $(\!|P_1 \| P_2|\!)_c \downarrow_{(a=a)}$.

  - $P_1 \| P_2 \xrightarrow{a(\tilde{x})} P_1' \| P_2'$: By applying induction hypotheses on the premises $P_1 \xrightarrow{a(\tilde{x})} P_1'$ and $P_2 \xrightarrow{a(\tilde{x})} P_2'$, we have that $(\!|P_1|\!)_c \twoheadrightarrow^*$

$\simeq (\!|P_1'|\!)_c$ and $(\!|P_2|\!)_c \twoheadrightarrow^* \simeq (\!|P_2'|\!)_c$. We only can apply (Sync) to mimic this transition.

$$\frac{\emptyset : (\!|P_1|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \qquad \emptyset : (\!|P_2'|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_2'|\!)_p}{\emptyset : (\!|P_1|\!)_p \parallel \emptyset : (\!|P_2|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \parallel \emptyset : (\!|P_2'|\!)_p}$$

Again we have that: $(\!|P_1'\|P_2'|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2'|\!)_p$.

– $P_1\|P_2 \xrightarrow{\alpha} P_1'\|P_2$ if $P_1 \xrightarrow{\alpha} P_1'$ and $P_2 \xrightarrow{sub(\alpha):}$ or $P_1\|P_2 \xrightarrow{\alpha} P_1\|P_2'$ if $P_2 \xrightarrow{\alpha} P_2'$ and $P_1 \xrightarrow{sub(\alpha):}$. we consider only the first case and by applying induction hypotheses on the premises $P_1 \xrightarrow{\alpha} P_1'$ and $P_2 \xrightarrow{sub(\alpha):}$, we have that $(\!|P_1|\!)_c \twoheadrightarrow^* \simeq (\!|P_1'|\!)_c$ and $(\!|P_2|\!)_c \twoheadrightarrow^* \simeq (\!|P_2|\!)_c$. We have many cases depending on the performed action:

1. if $\alpha = \tau$ then $P_1\|P_2 \xrightarrow{\tau} P_1'\|P_2$ with $P_1 \xrightarrow{\tau} P_1'$ and $P_2 \xrightarrow{sub(\tau):}$. We can apply (τ-Int) to mimic this transition.

$$\frac{\emptyset : (\!|P_1|\!)_p \xrightarrow{\nu\tilde{y}\overline{\Pi}\tilde{x}} \emptyset : (\!|P_1'|\!)_p \qquad \Pi \simeq \mathsf{ff}}{\emptyset : (\!|P_1|\!)_p \| \emptyset : (\!|P_2|\!)_p \xrightarrow{\tau} \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2|\!)_p}$$

and again we have that: $(\!|P_1'\|P_2|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2|\!)_p$.

2. if $\alpha = a(\tilde{x})$: then $P_1\|P_2 \xrightarrow{a(\tilde{x})} P_1'\|P_2$ with $P_1 \xrightarrow{a(\tilde{x})} P_1'$ and $P_2 \xrightarrow{a:}$. We can apply (Sync) to mimic this transition.

$$\frac{\emptyset : (\!|P_1|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \quad \dfrac{\emptyset : (\!|P_2|\!)_p \xrightarrow{\widetilde{(a=a)(a,\tilde{x})}} \emptyset : (\!|P_2|\!)_p}{\emptyset : (\!|P_2|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_2|\!)_p}}{\emptyset : (\!|P_1|\!)_p \parallel \emptyset : (\!|P_2|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \parallel \emptyset : (\!|P_2|\!)_p}$$

Again we have that: $(\!|P_1'\|P_2|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2|\!)_p$.

3. if $\alpha = \nu\tilde{y}\bar{a}\tilde{x}$ then $P_1\|P_2 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_1'\|P_2$ with $P_1 \xrightarrow{\nu\tilde{y}\bar{a}\tilde{x}} P_1'$ and $P_2 \xrightarrow{a:}$. We can apply (Comp).

$$\frac{\emptyset : (\!|P_1|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \qquad \dfrac{\emptyset : (\!|P_2|\!)_p \xmapsto{\widetilde{(a=a)}(a,\tilde{x})} \emptyset : (\!|P_2|\!)_p}{\emptyset : (\!|P_2|\!)_p \xrightarrow{(a=a)(a,\tilde{x})} \emptyset : (\!|P_2|\!)_p}}{\emptyset : (\!|P_1|\!)_p \parallel \emptyset : (\!|P_2|\!)_p \xrightarrow{\nu\tilde{y}\overline{(a=a)}(a,\tilde{x})} \emptyset : (\!|P_1'|\!)_p \parallel \emptyset : (\!|P_2|\!)_p}$$

Again we have that: $(\!|P_1' \| P_2|\!)_c \simeq \emptyset : (\!|P_1'|\!)_p \| \emptyset : (\!|P_2|\!)_p$. Notice that the $b\pi$ term and its encoding have the same observable behavior i.e., $P_1 \| P_2 \downarrow_a$ and $(\!|P_1 \| P_2|\!)_c \downarrow_{(a=a)}$.

□

A.2 The Smart Conference System in $Ab^{a}CuS$

Program A.1: The set of definitions used in the scenario in $Ab^{a}CuS$

```
1  public class Defs {
2   public static final String REQUEST = "S_REQ";
3   public static final String UPDATE = "INTEREST_UPDATE";
4   public static final String REPLY = "INTEREST_REPLY";
5
6   public static final String SESSION_ATTRIBUTE_NAME = "session";
7   public static final String INTEREST_ATTRIBUTE_NAME = "interest";
8   public static final String ID_ATTRIBUTE_NAME = "id";
9   public static final String ROLE_ATTRIBUTE_NAME = "role";
10  public static final String DESTINATION_ATTRIBUTE_NAME = "dest";
11  public static final String NAME_ATTRIBUTE_NAME = "name";
12  public static final String RELOCATE_ATTRIBUTE_NAME = "relocate";
13  public static final String PREVIOUS_SESSION_ATTRIBUTE_NAME =
        "prevSession";
14  public static final String NEW_SESSION_ATTRIBUTE_NAME = "newSession";
15  public static final String PROVIDER = "Provider";
16
17  public final static Attribute<String> session = new
        Attribute<>(SESSION_ATTRIBUTE_NAME, String.class);
18  public final static Attribute<String> interest = new
        Attribute<>(INTEREST_ATTRIBUTE_NAME, String.class);
19  public final static Attribute<Integer> id = new
        Attribute<>(ID_ATTRIBUTE_NAME, Integer.class);
20  public final static Attribute<String> role = new
        Attribute<>(ROLE_ATTRIBUTE_NAME, String.class);
21  public static final Attribute<String> destination = new
        Attribute<>(DESTINATION_ATTRIBUTE_NAME, String.class);
```

```
22   public static final Attribute<String> name = new
         Attribute<>(NAME_ATTRIBUTE_NAME, String.class);
23   public static final Attribute<Boolean> relocate = new
         Attribute<>(RELOCATE_ATTRIBUTE_NAME, Boolean.class);
24   public static final Attribute<String> previousSession = new
         Attribute<>(PREVIOUS_SESSION_ATTRIBUTE_NAME, String.class);
25   public static final Attribute<String> newSession = new
         Attribute<>(NEW_SESSION_ATTRIBUTE_NAME, String.class);
26
27  }
```

Program A.2: The ParticipantAgent in $Ab^{a}CuS$

```
1   public class ParticipantAgent extends AbCProcess {
2
3    private String topic;
4
5    public ParticipantAgent( String name , String topic ) {
6     super( name );
7     this.topic = topic;
8    }
9
10   @Override
11   protected void doRun() throws Exception {
12    setValue(Defs.interest, this.topic);
13    send(
14     new HasValue(
15       Defs.ROLE,
16       Defs.PROVIDER
17     ) ,
18     new Tuple(
19       getValue(Defs.interest) ,
20       Defs.REQUEST ,
21       getValue(Defs.id)
22
23     )
24    );
25    Tuple value = (Tuple) receive( o -> isAnInterestReply( o ) );
26    setValue(Defs.destination, (String) value.get(2));
27    while (true) {
28     value = (Tuple) receive( o -> isAnInterestUpdate( o ) );
29     setValue(Defs.destination, (String) value.get(3));
30    }
31   }
32
33   private AbCPredicate isAnInterestReply(Object o) {
34    if (o instanceof Tuple) {
35     Tuple t = (Tuple) o;
36     try {
```

```
37      if ((getValue(Defs.interest).equals(t.get(0))&&
38        Defs.REPLY.equals(t.get(1)))
39        {
40         return new TruePredicate();
41       }
42     } catch (AbCAttributeTypeException e) {
43      e.printStackTrace();
44     }
45    }
46    return new FalsePredicate();
47   }
48
49   private AbCPredicate isAnInterestUpdate( Object o ) {
50    if (o instanceof Tuple) {
51     Tuple t = (Tuple) o;
52     try {
53      if ((getValue(Defs.interest).equals(t.get(1)))&&
54        (Defs.UPDATE.equals(t.get(2))) {
55       return new TruePredicate();
56      }
57     } catch (AbCAttributeTypeException e) {
58      e.printStackTrace();
59     }
60    }
61    return new FalsePredicate();
62   }
63
64  }
```

Program A.3: The service process in $Ab^{a}C$uS

```
1   public class Service extends AbCProcess {
2   @Override
3   protected void doRun() throws Exception {
4    while (true) {
5     Tuple value = (Tuple) receive(o -> isARequest(o));
6     exec(new AbCProcess() {
7      @Override
8      protected void doRun() throws Exception {
9       send(
10        new HasValue(Defs.id, value.get(2)),
11        new Tuple(
12         getValue(Defs.session) ,
13         Defs.REPLY ,
14         getValue(Defs.name)
15        )
16       );
17    }});
18    }
```

```
19   }
20   protected AbCPredicate isARequest( Object o ) {
21    if (o instanceof Tuple) {
22     Tuple t = (Tuple) o;
23     try {
24      if (((getValue(Defs.session).equals(t.get(0))&&
25        (Defs.REQUEST.equals(t.get(1))))) {
26       return new TruePredicate();
27      }
28     } catch (AbCAttributeTypeException e) {
29      e.printStackTrace();
30     }
31    }
32    return new FalsePredicate();
33   }
34  }
```

Program A.4: The relocation process in $Ab^{a}C$uS

```
1   public class Relocation extends AbCProcess {
2   @Override
3   protected void doRun() throws Exception {
4    while (true) {
5     waitUntil(new HasValue(Defs.relocate, true));
6     setValue(
7       Defs.previousSession,
8       getValue(Defs.session));
9     setValue(
10      Defs.session,
11      getValue(Defs.newSession));
12    setValue(
13      Defs.relocate ,
14      false);
15    send(
16      new Or(
17       new HasValue(
18        Defs.interest,
19        getValue(Defs.session)
20       ) ,
21       new HasValue(
22         Defs.session,
23         getValue(Defs.session)
24       )
25      ),
26      new Tuple(
27       getValue( Defs.previousSession ) ,
28       getValue( Defs.session ) ,
29       Defs.UPDATE ,
30       getValue( Defs.name )
```

```
31       )
32        );
33     }
34   }
35 }
```

Program A.5: The Updating process in $Ab^{a}CuS$

```
1  public class Updating extends AbCProcess {
2
3  @Override
4  protected void doRun() throws Exception {
5   while (true) {
6    Tuple value = (Tuple) receive( o -> isAnUpdateMessage( o ) );
7    setValue(
8      Defs.previousSession,
9      getValue(Defs.session));
10   setValue(
11     Defs.session,
12   (String) value.get(0));
13    exec(new AbCProcess() {
14      @Override
15      protected void doRun() throws Exception {
16          send(
17           new Or(
18      new HasValue(
19       Defs.interest,
20       getValue(Defs.session)
21      ) ,
22      new HasValue(
23        Defs.session,
24        getValue(Defs.session)
25      )
26     ),
27     new Tuple(
28      getValue( Defs.previousSession ) ,
29      getValue( Defs.session ) ,
30      Defs.UPDATE ,
31      getValue( Defs.name )
32     )
33    );
34    }});
35   }
36   }
37   protected AbCPredicate isAnUpdateMessage( Object o ) {
38   if (o instanceof Tuple) {
39    Tuple t = (Tuple) o;
40    try {
41     if ((getValue(Defs.session).equals(t.get(1)))&&
```

$$
\begin{array}{lll}
\text{(Components)} & C ::= & \Gamma : P \quad | \quad C_1 | C_2 \\
\\
\text{(Processes)} & P ::= & \\
\quad \text{(Inaction)} & & \mathsf{0} \\
\quad \text{(Input)} & | & \Pi(x).P \\
\quad \text{(Output)} & | & (u)@\Pi.P \\
\quad \text{(Update)} & | & [a := u].P \\
\quad \text{(Choice)} & | & P_1 + P_2 \\
\quad \text{(Call)} & | & K \\
\\
\text{(Predicates)} & \Pi ::= & \mathsf{tt} \quad | \quad a = u \quad | \quad \Pi_1 \wedge \Pi_2 \quad | \quad \neg \Pi \\
\\
\text{(Data)} & u ::= & v \quad | \quad x
\end{array}
$$

Table 14: The syntax of the old *AbC* calculus

```
42        (Defs.UPDATE.equals(t.get(2))) {
43       return new TruePredicate();
44      }
45     } catch (AbCAttributeTypeException e) {
46      e.printStackTrace();
47     }
48    }
49    return new FalsePredicate();
50   }
51   }
```

A.3 A Formal Definition for the Old AbC Calculus

The syntax of the old *AbC* calculus is reported in Table 14. The top-level entities of the calculus are *components* (C), a component consisting either of a process P with a set of attributes Γ, denoted $\Gamma : P$, or of the parallel composition $C_1|C_2$ of two components. The attribute *environment* $\Gamma : \mathcal{A} \to \mathcal{V}$ is a map from attribute identifiers $a \in \mathcal{A}$ to values $v \in \mathcal{V}$. No restriction is enforced about the attributes of different components, in the sense that different components can have equal or different sets of attributes depending on the system being modelled.

A *process* is either the inactive process $\mathtt{0}$, an action-prefixed process, the choice $P_1 + P_2$ between two processes, or a possibly recursive call K to a process identified as K in the system. We assume that each process has a unique process definition $K \triangleq P$.

There are three kinds of prefix *actions*. The attribute-based input $\Pi(x)$ receives a message from any process whose attributes satisfy the predicate Π, variable x being a placeholder for the received message; the attribute-based output $(u)@\Pi$ broadcasts the message u to all processes whose attributes satisfy the predicate Π; and the update $[a := u]$ sets the value of an attribute a to u in the local environment. A *predicate* Π either checks the value of an attribute or is the propositional combination of predicates. Predicate $\mathtt{tt}$ is satisfied by all attributes and is used for full broadcast.

Data u can be a constant value, $v \in \mathcal{V}$, or a variables, x. The only binder of the calculus is $\Pi(x).P$, which binds variable x in the continuation process P. The derived notions of *bound* and *free* variables are standard. We assume that our processes are *closed* (i.e., no free variable), while free names can be used whenever needed. For the sake of minimality, the calculus does not equipped with a restriction operator. The need of such primitives will be the subject of further studies.

In the rest of the section, we present the *reduction semantics* of the old *AbC* calculus. The semantic definition relies on a standard structural congruence relation, defined by the rules in Table 16. The first three laws corresponds to the Abelian monoid laws for parallel $\mid$ (with $\Gamma : \mathtt{0}$ as unit). The fourth law permits lifting the relation from processes to components. The rest of the laws are the Abelian monoid laws for sum $+$ (with $\mathtt{0}$ as unit), the unfolding law and the α-conversion ($\equiv_\alpha$) law.

The semantics is formalised by means of an unlabelled transition relation as shown in Table 15. A transition $C \to C'$ denotes that the component C reduces to the component C' by either performing an attribute update $[a := u]$, or performing a one-to-many group communication. The group addressed by communication is determined by the predicates on both the sender and the receiver sides. Rule (Struct) links the congruence to the transition relation: structurally congruent components are

(Struct)

$$\frac{C \equiv C_1 \qquad C_1 \to C_2 \qquad C_2 \equiv C'}{C \to C'}$$

(Upd)

$$\Gamma : [a := v].P + Q \mid C \to \Gamma[a \mapsto v] : P \mid C$$

(Com)

$$\Gamma : (v)@\Pi.P + Q \mid \prod_{i=1}^{m} \Gamma_i : \Pi_i(x_i).P_i + Q_i \mid C \to \Gamma : P \mid \prod_{i=1}^{m} \Gamma_i : P_i[v/x_i] \mid C$$

$$\texttt{s.t.}\ \forall i = 1, \ldots, m \quad (\Gamma_i \models \Pi \wedge \Gamma \models \Pi_i)$$

$$\wedge \quad (C \not\equiv \Gamma' : \Pi'(x).P' + Q' \mid C' \quad \texttt{where} \quad \Gamma' \models \Pi \ \wedge \Gamma \ \models \Pi')$$

Table 15: Reduction semantics of the old AbC calculus

interchangeable. Rule (Upd) defines the ability of the system to evolve by updating the value of an attribute in one of its components. Notation $\Gamma[a \mapsto v]$ denotes the environment update: $\Gamma[a \mapsto v](a') = \Gamma(a')$ if $a \neq a'$ and v otherwise. Rule (Com) defines the ability of the system to evolve by matching an available output $(v)@\Pi$ in one of its components with other components ready to make an input $\Pi_i(x)$, provided that the attributes of the receivers Γ_i satisfy the output predicate Π of the sender, and vice versa the attributes of the sender Γ satisfy the input predicates Π_i. Notation $\prod_{i=1}^{m} C_i$ denotes the parallel composition $C_1 \mid \ldots \mid C_m$; if $m = 0$, this stands for the process $\mathsf{0}$ (this permits modelling the case where the output is executed irrespective of the presence of listeners). Once the value is received, it replaces the free occurrences of the input variable in the continuation process. This is achieved by applying the *substitution* $[v/x_i]$ to P_i. Since we consider closed processes, this entails that an output always ranges on values v. Moreover, a communication reduction also requires a specific condition on the other system components C. Every component in C is excluded from the communication. If,

$$
\begin{aligned}
C_1|C_2 &\equiv C_2|C_1 \\
(C_1|C_2)|C_3 &\equiv C_1|(C_2|C_3) \\
C|\Gamma:\mathsf{0} &\equiv C \\
\Gamma:P_1 &\equiv \Gamma:P_2 \quad \text{if } P_1 \equiv P_2
\end{aligned}
$$

$$
\begin{aligned}
P_1 + P_2 &\equiv P_2 + P_1 \\
(P_1 + P_2) + P_3 &\equiv P_1 + (P_2 + P_3) \\
P + \mathsf{0} &\equiv P \\
K &\equiv P \quad \text{if } K \triangleq P \\
P_1 &\equiv P_2 \quad \text{if } P_1 \equiv_\alpha P_2
\end{aligned}
$$

Table 16: Structural congruence

e.g., we let $\Gamma : P$ be such a component then either its attributes in Γ do not satisfy the sender predicate, or process P is not ready to perform an appropriate input action. Notably, we exploit the structural congruence to check the structure of these components. Clearly, the Structural congruence is decidable because it does not depend on predicate equivalence. Indeed, structural congruence can be easily computed using a static check inductively defined on the syntax of the calculus.

References

[AC93] Gul Agha and Christian J Callsen. *ActorSpace: an open distributed programming paradigm*, volume 28. ACM, 1993. 5, 38, 143, 148

[ADL+15] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1840–1845. ACM, 2015. xiv, 2, 9, 11, 133, 135

[ADL16a] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. On the power of attribute-based communication. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP International Conference, FORTE*, pages 1–18. Springer, 2016. Full technical report can be found on `http://arxiv.org/abs/1602.05635`. xiv, 8, 135

[ADL16b] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. Programming of CAS systems by relying on attribute-based communication. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pages 539–553. Springer, 2016. xiv, 100

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. 139

[AMST92] Gul Agha, Ian A Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In *International Conference on Concurrency Theory*, pages 565–579. Springer, 1992. 140

[AMST97] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997. 140

[And12] William J Anderson. *Continuous-time Markov chains: An applications-oriented approach*. Springer Science & Business Media, 2012. 120

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006. 144

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model. *Draft of specification, version*, 2(3), 2004. 144

[BDG+15] Luca Bortolussi, Rocco De Nicola, Vashti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. Carma: Collective adaptive resource-sharing markovian agents. *Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2015*, pages 16–31, 2015. 148

[BDP99a] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Basic observables for processes. *Inf. Comput.*, 149(1):77–98, 1999. 68

[BDP99b] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. A theory of "may" testing for asynchronous languages. In *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 165–179, 1999. 142

[BN02] Michael A Bass and Frank T Nguyen. Unified publish and subscribe paradigm for local and remote publishing destinations, June 11 2002. US Patent 6,405,266. 5, 38

[Bro06] Manfred Broy. The'grand challenge'in informatics: engineering software-intensive systems. *Computer*, 39(10):72–80, 2006. 2

[CKV01] Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing (CSUR)*, 33(4):427–469, 2001. 5, 38

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2:251–273, August 1984. 100

[CM95] Flaviu Cristian and Shivakant Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Autonomous Decentralized Systems,*

1995. Proceedings. ISADS 95., Second International Symposium on, pages 215–221. IEEE, 1995. 100

[Cri91] Flaviu Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991. 100

[DFLP13] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In *Formal Methods for Components and Objects*, pages 25–48. Springer, 2013. 135

[DLPT14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: the scel language. *ACM Transactions on Autonomous and Adaptive Systems*, pages 1–29, 2014. 2, 135

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36:372–421, December 2004. 100

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. 5, 38, 39

[EM99] Cristian Ene and Traian Muntean. Expressiveness of point-to-point versus broadcast communications. In *Fundamentals of Computation Theory*, pages 258–268. Springer, 1999. 36, 138

[EM01] Christian Ene and Traian Muntean. A broadcast-based calculus for communicating systems. In *Parallel and Distributed Processing Symposium, International*, volume 3, pages 30149b–30149b. IEEE Computer Society, 2001. 9, 36, 38, 138

[Fer15] Alois Ferscha. Collective adaptive systems. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pages 893–895. ACM, 2015. 2

[Fre93] Karen A. Frenkel. An interview with robin milner. *Commun. ACM*, 36(1):90–97, January 1993. Interviewee-Milner, Robin. 140

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962. 56

[GWGJ10] Thomas Given-Wilson, Daniele Gorla, and Barry Jay. Concurrent pattern calculus. In *Theoretical Computer Science*, pages 244–258. Springer, 2010. 135

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973. 139

[HC99] Hugh W Holbrook and David R Cheriton. Ip multicast channels: Express support for large-scale single-source applications. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 65–78. ACM, 1999. 2, 5, 8, 38

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008. 144

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977. 139

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. 5, 46, 133

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *European Conference on Object-Oriented Programming*, pages 133–147. Springer, 1991. 8, 142

[HY95] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995. 69

[JK06] Barry Jay and Delia Kesner. *Pure Pattern Calculus*, pages 100–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. 135

[JLN09] Mathias John, Cédric Lhoussaine, and Joachim Niehren. Dynamic compartments in the imperative π-calculus. In *Computational Methods in Systems Biology*, pages 235–250. Springer, 2009. 138

[JLNU08] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde M Uhrmacher. The attributed pi calculus. In *Computational Methods in Systems Biology*, pages 83–102. Springer, 2008. 135, 136

[JLNU10] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Adelinde M Uhrmacher. The attributed pi-calculus with priorities. In *Transactions on Computational Systems Biology XII*, pages 13–76. Springer, 2010. 135

[JR06] Duncan E Jackson and Francis LW Ratnieks. Communication in ants. *Current biology*, 16(15):R570–R574, 2006. 7

[Kla15] Annabelle Klarl. Engineering self-adaptive systems with the role-based architecture of helena. In *Infrastructure for Collaborative Enterprises, WETICE 2015, Larnaca, Cyprus, June 15-17, 2015*, pages 3–8, 2015. 144

[Kop11] Hermann Kopetz. Internet of things. In *Real-time systems*, pages 307–323. Springer, 2011. 2

[Mil80] Robin Milner. A calculus of communicating systems. 1980. 5, 133, 140

[Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989. 37, 71, 139

[Mil93] Robin Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, January 1993. 140

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and computation*, 100(1):41–77, 1992. 5, 7, 36, 46, 64, 133, 135, 138

[MS92] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Automata, Languages and Programming*, pages 685–695. Springer, 1992. 68, 69

[MT97] Ian A. Mason and Carolyn L. Talcott. A semantically sound actor tranlsation. In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, pages 369–378. Springer, 1997. 140

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 263–273. IEEE, 2004. 2

[NB14] Laura Nenzi and Luca Bortolussi. Specifying and monitoring properties of stochastic spatio-temporal systems in signal temporal logic. In *8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2014, Bratislava, Slovakia, December 9-11, 2014*. Springer, 2014. 148

[NH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1):83 – 133, 1984. 142

[OGCD10] Rehan O'Grady, Roderich Groß, Anders Lyhne Christensen, and Marco Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 28(4):439–455, 2010. 65

[OPT02] Karol Ostrovsky, KVS Prasad, and Walid Taha. Towards a primitive higher order calculus of broadcasting systems. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 2–13. ACM, 2002. 138

[PBMD15] Carlo Pinciroli, Michael Bonani, Francesco Mondada, and Marco Dorigo. Adaptation and awareness in robot ensembles: Scenarios and algorithms. In *Software Engineering for Collective Autonomic Systems*, pages 471–494. Springer, 2015. 65

[PBS89] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7:217–246, August 1989. 100

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. 100

[Pra91] KVS Prasad. A calculus of broadcasting systems. In *TAPSOFT'91*, pages 338–358. Springer, 1991. 2, 36, 138

[Pra95] Kuchi VS Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2):285–327, 1995. 138

[PS96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. 143

[RTRH05] CA Rouff, WF Truszkowski, JL Rash, and MG Hinchey. A survey of formal methods for intelligent swarms. *Greenbelt, MD: NASA Goddard Space Flight Center*, 2005. 145

[SCC+12] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige. Large-scale complex it systems. *Communications of the ACM*, 55(7):71–77, 2012. 3

[Sch08] Tim P Schulze. Efficient kinetic monte carlo simulation. *Journal of Computational Physics*, 227(4):2455–2462, 2008. 121

[SW03] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003. 2, 69, 78

[Tal96] Carolyn L Talcott. An actor rewriting theory. *Electronic Notes in Theoretical Computer Science*, 4:361–384, 1996. 140

[Tal97] Carolyn Talcott. Interaction semantics for components of distributed systems. In *Formal Methods for Open Object-based Distributed Systems*, pages 154–169. Springer, 1997. 140

[TZA02] Prasannaa Thati, Reza Ziaei, and Gul Agha. A theory of may testing for actors. In *Formal Methods for Open Object-Based Distributed Systems V, IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002), March 20-22, 2002, Enschede, The Netherlands*, pages 147–162, 2002. 142

[VDB13] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, pages 114–128. Springer, 2013. 2

[VNR13] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. Broadcast, Denial-of-Service, and Secure Communication. In *10th International Conference on integrated Formal Methods (iFM'13)*, volume 7940 of *LNCS*, pages 410–427, 2013. 138, 139

Unless otherwise expressly stated, all original material of whatever nature created by Yehia Abd Alrahman and included in this thesis, is licensed under a Creative Commons Attribution Noncommercial Share Alike 2.5 Italy License.

Check creativecommons.org/licenses/by-nc-sa/2.5/it/ for the legal code of the full license.

Ask the author about other uses.