Programming Interactions in Collective Adaptive Systems by Relying on Attribute-based Communication☆

Yehia Abd Alrahman[a], Rocco De Nicola[b,c], Michele Loreti[d]

[a] *University of Gothenburg, Gothenburg, Sweden*
[b] *IMT School for Advanced Studies, Lucca, Italy*
[c] *National Cybersecurity Laboratory, CINI Roma, Italy*
[d] *Università di Camerino, Camerino, Italy*

Abstract

Collective adaptive systems are new emerging computational systems consisting of a large number of interacting components and featuring complex behaviour. These systems are usually distributed, heterogeneous, decentralised and interdependent, and are operating in dynamic and possibly unpredictable environments. Finding ways to understand and design these systems and, most of all, to model the interactions of their components, is a difficult but important endeavour. In this article we propose a language-based approach for programming the interactions of collective-adaptive systems by relying on attribute-based communication; a paradigm that permits a group of partners to communicate by considering their run-time properties and capabilities. We introduce AbC, a foundational calculus for attribute-based communication and show how its linguistic primitives can be used to program a sophisticated variant of the well-known problem of Stable Allocation in Content Delivery Networks. In our variant, content providers are assigned to clients based on collaboration and by taking into account the preferences of both parties in a fully anonymous and distributed settings. We also illustrate the expressive power of attribute-based communication by showing the natural encoding of group-based, publish/subscribe-based and channel-based communication paradigms into AbC.

Keywords: Collective-adaptive systems, Attribute-Based Communication, Process calculus, Operational semantics, Computing methodologies

1. Introduction

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 components and featuring complex interaction mechanisms, e.g., *Software-Intensive Systems* [1], *IoT Systems* [2], and *Collective Adaptive Systems* (CAS) [3, 4, 5]. These systems are usually distributed, heterogeneous, decentralised and interdependent, and frequently operate in dynamic and unpredictable environments. Historically, as software systems grew larger, the focus shifted from the complexity of developing efficient algorithms to the complexity of structuring large systems, with the inevitable complications induced by their distributed and concurrent nature [6]. In [7], the authors mentioned that structuring/reconfiguring large software systems is difficult, laborious and error-prone due to (i) the lack of concise abstractions and (ii) the insufficient automatic support to the reconfiguration process. These two points are crucial for an engineering approach to succeed, especially when dealing with additional complexities that arise from the fact that CAS systems operate in and interact with an open and non-deterministic environment.

☆Y. Abd Alrahman is funded by the ERC consolidator grant D-SynMA under the European Union's Horizon 2020 research and innovation programme (grant agreement No 772459). The work of R. De Nicola and M. Loreti is partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

Email addresses: yehia.abd.alrahman@gu.se (Yehia Abd Alrahman), rocco.denicola@imtlucca.it (Rocco De Nicola), michele.loreti@unicam.it (Michele Loreti)

Most of the current communication models and programming frameworks still handle interactions between distributed components by relying on their identities, see e.g., the Actor model [8], or by relying on the communication channels on which they are listening, using point-to-point channel-based communication [9], or multicast with explicit addressing [10], or broadcast communication [11]. These models use entities, such as names or addresses which are totally independent from the run-time properties and from the capabilities of the interacting components, to facilitate interactions. This makes it hard to program complex behaviours and coordination mechanisms, such as reconfiguration, adaptation, opportunistic interactions, etc., that depend on components' status rather than on their identities or addresses.

Indeed, the existing models of concurrent and distributed systems are *communication centric*; the behaviour is defined mainly as a process implementing a specific communication protocol and the programmer has to manage the *channels* or the *addresses* on which components interact. In our view, to deal with CAS, it is instead important to consider *data centric* interaction primitives that abstract from the underlying communication infrastructure (i.e., they are infrastructure-agnostic) and rely on *anonymous one-to-many* interactions to send messages to potential receivers. Data centric interaction primitives provide the illusion of the existence of a shared global state of the whole system. Thus the programmer can base the interactions directly on the states of system components using logical formulae without worrying about how messages are being routed in the actual network. This task is delegated to an underlying communication infrastructure that mediates the interactions between system components.

In this article, we study the impact of a new data centric paradigm that permits a group of partners to interact by considering their properties and capabilities at run-time. Our findings have been triggered by the work on CAS, and by the recent attempts to define linguistic primitives to deal with such systems, see e.g. the Helena framework [12], the field calculus [13], Coordinated Actors [14], and the SCEL language [15].

1.1. Collective-Adaptive Systems

Collective-Adaptive Systems (CAS) [4] consist of a large number of interacting components which combine their behaviours, by forming collectives, to achieve specific goals depending on their attributes, objectives, and functionalities. Decision-making in these systems is complex and components interaction may lead to unexpected behaviours. CAS are inherently scalable [16] and the boundaries between different CAS are fluid in the sense that 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. The need of engineering techniques to address the challenges of developing, integrating, and deploying CAS is advocated in [17]. Also the development of theoretical foundation for CAS has been deemed important to understand their distinctive features and operational principles [18].

As stated before, the existing communication models do not scale with the high level of dynamicity of CAS, and a change of perspective, that takes into account run-time properties, status, and capabilities of communicating systems, is on demand; the key concepts of CAS should be the basis for guiding the development of the new communication modalities. In the following, we summarise these concepts borrowing from [18, 4, 16, 3]:

Distribution: components are distributed on a shared environment and evolve independently, without any centralised control.

Awareness: components are aware of their run-time status (*self-awareness*) and have some (partial) view of their surroundings (*context-awareness*).

Adaptivity: components adapt their behaviours and their interaction policies in response to the changes of their contextual conditions and to the collected awareness data. Thus awareness can be considered as a pre-condition to adaptation.

Interdependence: any change in the shared environment might influence the behaviour of components.

Collaboration: components collaborate and combine their behaviours to achieve system-level goals in response to changes in the environment or according to their predefined roles.

Anonymity: components communicate and exchange information without knowing the existence and/or identity of each other; for example, the identity of a service provider is not relevant, only its ability to provide the desired service is. Notice that existing communication paradigms, which are based on the idea of knowing the identity of the communication partners, are not adequate because they fundamentally contradict the idea of "anonymous" interaction.

Scalability and Open-endedness: components may join or leave a system without perturbing the overall behaviour of the system; senders emit messages without being aware of the presence of receivers and receivers do not rely on specific senders.

1.2. AbC: Attribute-based Communication

In order to capture the above mentioned concepts, we have developed *attribute-based communication*, a paradigm that permits a group of partners to communicate by considering propositional predicates over the attributes they expose. These predicates give the illusion of the existence of a shared global state of the whole system and communications can be based on the states of communicating partners rather than on their identities or physical addresses. Thus communication takes place anonymously in an implicit multicast fashion without a prior agreement among communicating partners. Anonymity of interaction allows programmers to secure scalability, dynamicity, and open-endedness more easily. Sending operations are non-blocking while receiving operations are. This breaks synchronisation dependencies between interacting partners, and permits modelling systems where communicating partners can enter or leave a group at any time without perturbing the overall behaviour. Attributes make it easy to model awareness by locally reading the values of the attributes that may represent either the component status (*self-awareness*) (e.g., the battery level of a robot) or the external environment (*context-awareness*) (e.g., the external humidity). Groups are dynamically formed at the time of interaction by considering the interested receiving components that satisfy sender's predicates, and any run-time changes of attribute values allow opportunistic interactions between components. By parameterising the interaction predicates with local attributes, groups can be implicitly changed and adaptation is naturally captured. Security mechanisms can be placed on top of the attribute-based framework following standard approaches. Indeed security communities have already established techniques to deal with such settings, i.e., attribute-based encryption [19] and attribute-based access control policy languages like `XACML` [1] and `FACPL` [20]. What is missing is actually a programming paradigm that realises these settings and we believe that *AbC* is such paradigm.

Modeling opportunistic interaction in classical communication paradigms like channel-based communication, e.g., π-calculus [21], is definitely more challenging because components have to agree on specific names or channels to interact. Channels have no connection with the component attributes or characteristics; they are specified as addresses where the exchange should happen. Names and channels are static and changing them locally at run-time requires explicit communication and intensive use of scoping mechanisms which do affect readability and compositionality of programs.

We would also like to stress that an attribute-based system is more than just the result of the parallel composition of its interacting components; it also takes into account the environment where components are executed. Indeed, the environment has a great impact on components behaviours and allows modelling interdependence, i.e., a situation where components influence each other unintentionally. For example, in an ant foraging system [22], where an ant disposes pheromone in the shared space to keep track of its way back home, the ant influences other ants behaviour as they are programmed to follow traces of pheromone with higher concentration. In this way, the ant unintentionally influences the behaviour of other ants by only modifying the environment.

Before we discuss the details of our approach, we want to mention that CAS features can be modelled in many different mathematical approaches. Here, we would like to distinguish between two closely related ones, namely the formal language approach (like ours) and the logical approach. For instance, Process algebra [23] follows a formal language approach while Multi-Agent Systems (MAS) [24] follow a logical one.

[1] `https://www.oasis-open.org`

In a nutshell, Process Algebra models distributed systems by algebraic means. The word 'process' here refers to the behaviour of a system. A system is anything showing behaviour, such as the execution of a software system, the actions of a machine or even the actions of a human being. A process is represented by a program with algebraic composition operators and thus compositionality is considered as the main principle in any process algebra. As opposed to Process Algebra, MAS originate as a sub-field of distributed artificial intelligence and model systems in terms of logical formulas. Thus MAS' models are represented in terms of logical formulas that can be used to synthesise correct implementations. Unfortunately the synthesis approach does not lend itself to compositionally. Indeed, the distributed synthesis problem is undecidable in general [25] and is only decidable for a number of fixed communication structures [26]. Logics, however, provide powerful tools to declaratively model the features of complex systems as MAS [24]. Different attempts have been carried out to bridge the gap between MAS and process algebra approaches, see [27, 28, 29], however this is out the scope of this article. We insert our results in the Process Algebra approach and we provide an operational and compositional model that precisely describes the step-by-step admissible evolutions of the system. Operational models ensure easily that implementations are correct with respect to their specifications and based on the evolution rules, system properties can be verified [30].

In this article we will introduce AbC, a process calculus comprising a minimal set of primitives that permits attribute-based communication and it is the outcome of different attempts towards modelling the interaction of CAS scenarios. From our experience, we learnt that any attribute-based paradigm, tailored for modelling the interaction of CAS systems, should at least provide support for the following notions, which are indeed the key ingredients of AbC:

- *Attribute Environment*: to provide a collection of attributes whose values represent the status of a component. These values can be used to control the behaviour of a component at run-time.
- *Attribute-based send and receive operations*: to establish dynamic communication links between different components based on the satisfaction of predicates over components attributes.
- *Attribute update operation*: to change attribute values based on contextual conditions and adapt the behaviour of a component accordingly.
- *Awareness construct*: to collect awareness data and take decisions based on the changes in the attribute environment.

A system is modelled in AbC as a set 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. Namely, a sender broadcasts a message tagged with a partial view of its attributes and a propositional predicate specifying the run-time properties of the targeted components. Accordingly, all other components that satisfy the sender predicate receive the message only if they are interested in its contents and the exposed attributes of the sender satisfy their receiving predicates; otherwise they ignore the message. Thus, communication takes place in an implicit multicast fashion, and communication partners are selected by relying on predicates over the attributes in their interfaces. The main novelty of the AbC interaction model is that low-level details of how communication links are established among interacting components are abstracted away from the programmer and are delegated to the underlying communication infrastructure that mediates the interactions between components. Thus the programmer is given the ability to specify the interactions between communicating components using logical formulae rather than using physical names or addresses as the case of IP multicast [10]. Note that in IP multicast the reference address of the group is explicitly included in the message while in AbC components are unaware of the existence of each other and they receive messages based on mutual interests. In other words, the primitives of AbC are data centric and give the programmer the illusion of a shared global state of the system in which she can specify interactions using logical formulae and thus basing the interactions directly on the states of the different components without worrying about how messages are routed and delivered. Note that the communication infrastructure is usually distributed and interacting components can be connected to different server nodes in the infrastructure where they send and receive messages. Thus a component needs only to know the

server to which it is connected so that it participates in interaction. An infrastructure may serve as a mere message-forwarder (i.e., it does not have knowledge about component states) or it may have knowledge and is involved in filtering messages. This is a choice that has to be made and evaluated with respect to the type of the application being developed. For instance, when the frequency of attribute updates is high the forwarding infrastructure is more appropriate and this is actually our choice in this article.

AbC has been used as the basis for building high-level and formally verifiable communication APIs for CAS. For example, we developed APIs for Java [31], for Go [32], and for Erlang [33]. The actual implementations of these APIs, except for the latter, fully rely on the operational semantics of *AbC*. The formal correspondence between the *AbC* primitives and the programming constructs of the above mentioned APIs is instrumental to increase the confidence on the behaviour of programs by analysing the operational semantics of the original *AbC* specifications. This would also permit exploiting these primitives to program the interaction of CAS applications in different host languages as required by the application domain.

Contributions. The main contribution of this article is the assessment of a language-based approach to program the interactions in collective adaptive systems at a reasonable level of abstraction. We present the *AbC* calculus as a candidate language and concentrate on its linguistic primitives and on their use. We show the expressive power of *AbC* and how it can be used to naturally model non-trivial CAS. We show how to program with *AbC* a sizeable and a sophisticated variant of the well-known problem of Stable Allocation in Content Delivery Networks [34]. We also show how other well-known communication paradigms can be naturally encoded in *AbC*. We present a Go API, named $\mathcal{GoAt}$, and a distributed coordination infrastructure to support the interaction primitives of *AbC* and also to serve as a proof of concept on the feasibility of our proposal. The choice of the implementation language and the type of the infrastructure is based on our results in [35, 32] where we investigated other possible implementations and evaluated their performance. Here we refine these results and present the most efficient and distributed API. Our main focus here is on programming aspects and other more theoretical ones, concerned with formal semantics, behavioural relations, and equational laws, can be found in [36].

Note that this article is an extended and refined version of the conference paper presented in [37]. We have changed and adapted the syntax (and accordingly the semantics) of [37] to enrich the language and also to rule out complex constructions without compromising the expressive power. For instance, in the conference version receivers could not predicate on the attributes of senders. Also received values could not be applied instantly to the attributes of receivers and thus some important behaviours like the ones in the present case studies could not be encoded. Furthermore, scope restriction operators in the early version were a slight adaptation of the ones in [21] while here we introduce novel operators, specifically designed to capture specific features of collective adaptive systems and to model systems with fluid boundaries.

Structure of the article. In Section 2 we describe the *AbC* approach to program CAS interactions and we illustrate its expressive power. In Section 3 we present the case study of Stable Allocation and in Section 4 we discuss our approach to programming and its distinctive features. In Section 5 we introduce the Go API and its distributed infrastructure. In Section 6 we relate our approach to closely related state of arts and finally in Section 7 we report concluding remarks and future directions.

2. Programming CAS with attributes

In this section, we introduce the *attribute-based primitives* of *AbC* and show how they can be used to program CAS features. The semantic foundations of these primitives have been studied in [36] where a behavioural theory for *AbC* has been presented. For the sake of completeness, a detailed description of *AbC* and its operational semantics is reported in the Appendix. To capture the reader's intuition we will use a distributed variant of the *Graph Colouring Problem* [38] as a running example. This variant is designed to cover most of CAS features in a single case study with a well-understood system-level goal.

Example 2.1 (Step 1/4: Distributed Graph Colouring (DGC))**.** *Let us consider a distributed variant of the well known* Graph Colouring Problem *[38]. This problem can be rendered as a typical CAS scenario*

where a collective of agents, executing the same code, collaborate to achieve a system-level goal without any centralised control.

The goal is that of colouring the vertices of a graph in such a way that no two vertices sharing an edge have the same colour. Formally, we have a set of n vertices, each of which is identified by an id *(an integer in our model). Each vertex i has a colour c_i and a set of* neighbours N_i *such that $j \in N_i$ if and only if $i \in N_j$. We have to guarantee that, on termination, $c_j \neq c_i$ for each $j \in N_i$.* □

Components.

The basic building blocks to program a *distributed system* via the *attribute-based* primitives are *components*. A component is a *computational entity*, denoted by $\Gamma :_I P$, equipped with:

- an *attribute environment* Γ that associates attribute identifiers[2] $a \in \mathcal{A}$ to values $v \in \mathcal{V}$, where $\mathcal{A} \cap \mathcal{V} = \emptyset$;
- an *interface* $I \subseteq \mathcal{A}$ consisting of a set of *public attributes* that can be used to control the interactions with other components (elements in $dom(\Gamma) - I$ will be referred as *private attributes*);
- a *process* P describing a *component behaviour*.

The *public attribute environment* of a component $\Gamma :_I P$, is denoted by $\Gamma \downarrow I$ and represents the portion of Γ that can be perceived by the context. It can be obtained from Γ by limiting its domain to the attributes in the interface I:

$$(\Gamma \downarrow I)(a) = \begin{cases} \Gamma(a) & a \in I \\ \bot & \text{otherwise} \end{cases}$$

Example 2.2 (Step 2/4: DGC Components)**.** *Each vertex in DGC consists of a component of the form $C_i = \Gamma_i :_{\{\mathsf{id},\mathsf{N}\}} P_C$. Public attributes* id *and* N *are used to represent the vertex* id *and the set of its* neighbours *N, respectively. These* attributes*, that are part of the* public attribute environment*, will be used to* control *the interactions among partners.*

A group of components can be identified via a *predicate* Π, i.e., by predicating on the attribute values in their interfaces. Formally, a predicate is a *boolean assertion* over *attributes* of the form:

$$\text{(Predicates)} \qquad \Pi \;::=\; \mathsf{tt} \;\mid\; \mathsf{ff} \;\mid\; p_k(E_1, \ldots, E_k) \;\mid\; \Pi_1 \wedge \Pi_2 \;\mid\; \Pi_1 \vee \Pi_2 \;\mid\; \neg\Pi$$

where E is an expression respecting the following syntax:

$$\text{(Expressions)} \qquad E \;::=\; v \;\mid\; x \;\mid\; a \;\mid\; \texttt{this}.a \;\mid\; op(\tilde{E})$$

An *expression* E is built from constant values $v \in \mathcal{V}$, variables x, attribute identifiers a, a reference to the value of a (`this`.a) in the component that is executing the code, or using standard operators $op(\tilde{E})$[3]. The evaluation of expression E under Γ is denoted by $[\![E]\!]_\Gamma$. The definition of $[\![\cdot]\!]_\Gamma$ is standard, the only interesting case is $[\![\texttt{this}.a]\!]_\Gamma = \Gamma(a)$.

A *predicate* Π is built from the boolean constants tt and ff, from a *k-arity predicate* $p_k(E_1, \ldots, E_k)$ and also from standard boolean operators ($\neg$, $\wedge$ and $\vee$). The precise set of k-arity predicates is not detailed here; we only assume that each p_k can be a basic binary relation like $=, >, <, \leq, \geq$, and the predicates $\in$ and $\notin$. For instance the predicate $\in_2 (\mathsf{id}, \mathsf{N})$ is satisfied if and only if $\mathsf{id} \in \mathsf{N}$ for some element id and a set N. In the rest of the article we will use the infix notation for p_k and we will also drop the index k when it is clear from the context.

Intuitively a predicate Π is interpreted over an attribute environment Γ as indicated by the semantics of the satisfaction relation $\models$ (Table 1). We comment on the semantics regarding the case of a k-arity predicate

[2] In the rest of this article, we shall occasionally use the term "attribute" instead of "attribute identifier".

[3] We omit the specific syntax of operators used to build expressions, and use $\tilde{E}$ to denote sequences of expressions.

$\Gamma \models \mathsf{tt}$ for all Γ
$\Gamma \not\models \mathsf{ff}$ for all Γ
$\Gamma \models p_k(E_1, \ldots, E_k)$ iff $([\![E_1]\!]_\Gamma, \ldots, [\![E_k]\!]_\Gamma) \in [\![p_k]\!]$
$\Gamma \models \Pi_1 \wedge \Pi_2$ 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 1: The semantics of the satisfaction relation

$\Gamma \models p_k(E_1, \ldots, E_k)$ and the remaining cases have the standard semantics of propositional formulae. Clearly any predicate can be understood as a relation and thus this case states that a k-arity predicate $p_k(E_1, \ldots, E_k)$ is satisfied by an attribute environment Γ if and only if the evaluations of all expressions $(E_1, \ldots, E_k)$ under Γ are related by the relation associated with the evaluation of p_k ($[\![p_k]\!]$).

We say that a component $\Gamma :_I P$ satisfies predicate Π whenever its *public attribute environment* $\Gamma \downarrow I$ satisfies Π, i.e. $\Gamma \downarrow I \models \Pi$.

Example 2.3 (Step 3/4: DGC Predicates)**.** *The following predicate:*

$$(1 \in \mathsf{N})$$

identifies every component that has the identifier 1 *in its neighbour set* N*, i.e.,* $1 \in \Gamma(\mathsf{N})$.

Programming components

Processes running at a component, in addition to standard programming constructs, can execute two kinds of primitives: *self-awareness* and *interaction commands*. The former, that allows a process to suspend its activity until a given condition is satisfied, is used to coordinate processes executed in the same component. The latter enables interactions among processes located at different components.

Processes running within a component interact with each other locally via the attribute environment Γ that plays the role of a *shared memory*. To coordinate activities it is crucial that a process can *suspended* its computation until a condition on the *attribute environment*, expressed via a predicate, is satisfied. This can be done via the statement

$$\langle \Pi \rangle P$$

A component, with attribute environment Γ and executing $\langle \Pi \rangle P$, would suspend P until $\Gamma \models \Pi$.

Processes on different components, instead, interact via *attribute-based output* and *attribute-based input* actions. Namely, a sender process (executing inside a component with with an interface I and an attribute environment Γ) broadcasts a message "$\Gamma \downarrow I \triangleright \overline{\Pi}(\tilde{v})$", containing a sequence of values $\tilde{v}$ and tagged with its public attribute environment $\Gamma \downarrow I$ and a predicate Π specifying the run-time properties of the targeted components. Accordingly, all other components that satisfy the sender predicate (i.e., with some attribute environment Γ' and interface I' such that $\Gamma' \downarrow I' \models \Pi$) receive the message only if they are interested in its contents and the public attribute environment of the sender satisfy their receiving predicates Π' (i.e., $\Gamma \downarrow I \models \Pi'$); otherwise they ignore the message.

More precisely, an *attribute-based output*, executed inside a component with some interface I and attribute environment Γ, is the following:

$$(\tilde{E})@\Pi$$

It is used to send a message, consisting of the outcomes of the evaluation of a tuple of expressions $\tilde{E}$, i.e., $[\![\tilde{E}]\!]_\Gamma$, to all the components *satisfying* the closure of the predicate Π. The *closure* of Π, denoted by $\{\Pi\}_\Gamma$, is the predicate obtained from Π after replacing the occurrences of the expression `this`.a with their values $\Gamma(a)$. Note that the public environment of the sender $\Gamma \downarrow I$ is automatically attached to every sent message.

For instance a component, with an attribute environment Γ and an interface I in Example 2.2 that wants to send a message to all its *neighbours*, can execute the action:

$$(E)@(\texttt{this.id} \in \mathsf{N})$$

whose execution induces sending the value $[\![E]\!]_\Gamma$ to all components satisfying the *closure* of the predicate $(\texttt{this.id} \in \mathsf{N})$. Let us assume that $\Gamma(\mathsf{id}) = 1$ then we have that the closure predicate is $(1 \in \mathsf{N})$ which targets the neighbours of component 1, i.e., the components where the identifier 1 occurs in their neighbour set N. The public attribute environment $\Gamma \downarrow I = \{(\mathsf{id}, 1), (\mathsf{N}, \{\dots\})\}$ is automatically attached to the message so that the receiver can predicate on it. Note that we did not specify here the set of neighbours $(\mathsf{N}, \{\dots\})$.

Attribute-based output is non-blocking. This means that the action is executed without knowing if there are receivers and their exact number. In this sense the proposed approach implements an *asynchronous interaction* where the sender does not know if the message has been received or not. In particular, when the output predicate Π is equivalent to false, the output action is executed without affecting any other component in the system.

To receive messages a component can execute an *attribute-based input*, denoted by $\Pi(\tilde{x})$:

$$\Pi(\tilde{x})$$

This statement allows to receive a message sent by a component satisfying predicate Π; the sequence of variables $\tilde{x}$ acts as a placeholder for the received values.

The predicates used in input actions can also refer to variable names in $\tilde{x}$ and the received values can be used to check whether specific conditions are satisfied. For instance, the input action

$$((x = \text{“}try\text{”}) \wedge (\texttt{this.id} > \mathsf{id}) \wedge (\texttt{this.round} = z))(x, y, z)$$

can be used to receive a “try” message of the form $(\text{“}try\text{”}, c, r)$ for some colour c received on y and a round r received on z that is equal to $\texttt{this.round}$ and the value of the interface attribute id of the sending component is less than $\texttt{this.id}$. Thus, the predicate can be used to check either the received values or the values in the sender public environment.

The interaction operators we have just described can be combined with standard operators borrowed from process algebras to build processes whose syntax is reported below:

$$\begin{array}{lcl} P & ::= & \mathsf{0} \mid \Pi(\tilde{x}).U \mid (\tilde{E})@\Pi.U \mid \langle\Pi\rangle P \mid P_1 + P_2 \mid P_1|P_2 \mid K \\ U & ::= & [a := E]U \mid P \end{array}$$

A *process* P can be the *inactive* process $\mathsf{0}$, an *action-prefixed* process, $act.U$, where act is an input or output action and U is a process possibly preceded by an *attribute update*, a *context aware* $\langle\Pi\rangle P$ process, a *nodeterministic choice* between two processes $P_1 + P_2$, a *parallel composition* of two processes $P_1|P_2$, or a process call with a unique identifier K used in process definition $K \triangleq P$. The *parallel operator*, $P|Q$, models the interleaving between co-located processes, i.e., processes residing within the same component. The *choice operator*, $P + Q$, indicates a nondeterministic choice among P and Q. Notice that after a communication action is performed a (possibly empty) sequence of updates $([a := E])$ can be executed. These updates do change the attribute environment of a component and their effects appear immediately after the execution of their associated action.

Other process operators can be defined as *macros*, and we will use the following ones:

$$\textbf{if } \Pi \textbf{ then } P_1 \textbf{ else } P_2 \triangleq \langle\Pi\rangle P_1 + \langle\neg\Pi\rangle P_2 \tag{1}$$

$$[a_1 := E_1,\ a_2 := E_2,\ \dots,\ a_n := E_n]P \triangleq [a_1 := E_1][a_2 := E_2]\dots[a_n := E_n]P \tag{2}$$

$$\textbf{set}(a, E)P \triangleq ()@\mathsf{ff}.[a := E]P \tag{3}$$

The role of the macros in equations (1) and (2) is self-explanatory, and (3) is used to express a local computational step by a component that wants to evolve independently. Notice that the action ()@ff models a sending operation on a false predicate which is not satisfied by any communication partner and thus it is not received by other components. The only side effects of this macro is the attached attribute updates; it can be used by components that need to updates their own attribute values.

Building Systems

When the *attribute-based paradigm* is used, the exact underlying communication infrastructure is abstracted away. Components do not interact with each other via their addresses but by relying on their *public interfaces*. Thus the programming constructs (to compose different systems and to allow them to interact) abstract the low-level details of establishing communication links from the programmer and delegate this role to the underlying communication infrastructure where these interaction primitives will be based. As we will see in Section 5 this infrastructure is responsible for routing and ordering messages and also for establishing communication links among components in a reasonable way while preserving the properties of the composition primitives as defined by their semantics.

An *AbC system* is generated by the following grammar:

$$\text{(System)} \qquad C \quad ::= \quad \Gamma :_I P \quad | \quad C_1 \| C_2 \quad | \quad [\ C\ ]^{\triangleleft f} \quad | \quad [\ C\ ]^{\triangleright f}$$

Thus, a *system*, denoted by C, is basically either a *single component* $\Gamma :_I P$, a *parallel composition* of two systems, $C_1 \| C_2$, an *in*-scoped system $[\ C\ ]^{\triangleleft f}$ or an *out*-scoped system $[\ C\ ]^{\triangleleft f}$.

Note that within systems, it is important to have mechanisms that can be used to *restrict* components interactions, i.e., to facilitate private interactions between selected components. To this purpose, we have introduced the *restriction* operators $[\ C\ ]^{\triangleright f}$ and $[\ C\ ]^{\triangleleft f}$, where f is a function associating a predicate Π to a tuple of values $\tilde{v} \in \mathcal{V}^*$ and an attribute environment Γ.

Let us consider a system C that is composed of different components, one of which has a public attribute environment Γ and sends $\tilde{v}$ to components satisfying Π. When the message outgoes $[\ C\ ]^{\triangleright f}$, the target predicate is updated to consider also predicate $\Pi' = f(\Gamma, \tilde{v})$, thus the components satisfying $\Pi \wedge \Pi'$ will receive the message. To prevent the leak of a *secret* s outside C, we can use the following function:

$$f_s(\Gamma, \tilde{v}) = \begin{cases} \mathsf{tt} & s \notin \tilde{v} \\ \mathsf{ff} & s \in \tilde{v} \end{cases}$$

Similarly, $[\ C\ ]^{\triangleleft f}$ can be used to limit the ability of C to receive messages. In particular, if a component with public attribute environment Γ sends a message $\tilde{v}$ to components C satisfying Π, only the components in C that satisfy $\Pi \wedge f(\Gamma, \tilde{v})$ will be eligible to receive the message.

The restriction operators in *AbC* are novel and were designed to capture specific features of collective adaptive systems. For instance the fact that restriction operators can delimit the scope of targeted components by restricting or weakening the predicate of the transmitted message provides an elegant way to model systems with fluid boundaries. Note this is possible because the restriction function $f(\Gamma, \tilde{v})$ is not only parametric to the message contents $\tilde{v}$ but also to the public environment of the sending component Γ, which dynamically evolves at run-time. The dynamic degree of observability of exchanged messages provides a convenient way to model collective behaviour from a global point of view. This means that local interactions are hidden from external observers which can only observe the system as if it was a single huge entity, mimicking a similar notion of superorganism in ecology[4] [39].

Example 2.4 (Step 4/4: DGC System)**.** *In Example 2.1 we introduced the structure of the components modelling the vertices in the considered scenario. Here, we show how the behaviour of our components can*

[4] "superorganism is a collection of the same or similar organisms that are connected in a functional way and behave as a single organism, working to accomplish a set of global goals."

be programmed to assign a colour *(an integer) to each of them while avoiding that two neighbours get the same colour.*

The proposed algorithm consists of a sequence of rounds for colour selection that goes on until the specified goal is reached. At the end of each round at least one component gets assigned a colour.

Components use messages of the form $(\text{“}try\text{”}, c, r)$ *to inform their neighbours that at round* r *they want to select colour* c *and messages of the form* $(\text{“}done\text{”}, c, r)$ *to communicate that colour* c *has been definitely chosen at the end of round* r*. At the beginning of a round, each component* i *selects a colour and sends a* try*-message to all components in* N_i*. Component* i *also collects* try*-messages from its neighbours. The selected colour is assigned to a component only if it has the greatest id among those that have selected the same colour in that round. After the assignment, a* done*-message (associated with the current round) is sent to neighbours. A new round starts when a message, associated with a round* r *such that* $\mathtt{this}.round < r$*, is received.*

This algorithm can be implemented in AbC *by using four processes,* F *for forwarding* try*-messages to neighbours,* T *for handling* try*-messages,* D *for handling* done*-messages, and* A *for assigning a final colour. Process* P_C *of Example 2.2 is now defined as the parallel composition of these four processes:* $P_C = F \mid T \mid D \mid A$*.*

The following private attributes*, local to each component, are used to control the progress of our algorithm:* colour*,* round*,* done*,* assigned*,* used*,* counter*,* send *and* constraints*. The attribute “*colour*” stores the value of the selected colour, attribute “*round*” stores the current round while “*constraints*” and “*used*” are sets, registering the colours used by neighbours. The attribute “*counter*” counts the number of messages collected by a component while “*send*” is used to enable/disable forwarding of messages to neighbours. The private attributes of each component are all initialised with the following values:* $\mathsf{colour} = \bot$*,* $\mathsf{counter} = \mathsf{round} = \mathsf{done} = 0$*,* $\mathsf{constraints} = \mathsf{used} = \emptyset$*,* $\mathsf{send} = \mathsf{tt}$*, and* $\mathsf{assigned} = \mathsf{ff}$*.*

In process F *reported below, when the value of attribute* send *becomes* tt*, a new colour is selected (*$\min\{i \notin \mathtt{this}.used\}$*), and a* try*-message containing this colour and the current round is sent to all the components having* $\mathtt{this}.\mathsf{id}$ *as neighbour. The new colour is the smallest colour that has not yet been selected by neighbours, that is* $\min\{i \notin \mathtt{this}.used\}$*. The guard* $\neg\mathsf{assigned}$ *is used to make sure that components with assigned colours do not take part in the selection anymore.*

$$F \triangleq \langle \mathsf{send} \wedge \neg\mathsf{assigned}\rangle (\text{“}try\text{”}, \min\{i \notin \mathtt{this}.used\}, \mathtt{this}.\mathsf{round})@(\mathtt{this}.id \in \mathsf{N}).[\mathsf{send} := \mathsf{ff},\ \mathsf{colour} := \min\{i \notin \mathtt{this}.used\}]F$$

Note that the values of attributes send *and* colour *are updated simultaneously with the emission of the* try*-message. The attribute* send *is set to false so that this process does not propose a new colour before a decision is made on the proposed one. The attribute* colour *records the proposed colour, i.e.,* $\min\{i \notin \mathtt{this}.used\}$*.*

Process T*, reported below, receives messages of the form* $(\text{“}try\text{”}, c, r)$*. If* $r = \mathtt{this}.round$*, as in the first two branches, then the received message has been originated by another component performing the same round of the algorithm. The first branch is executed when* $\mathtt{this}.\mathsf{id} > \mathsf{id}$*, i.e., the sender has an id smaller than the id of the receiver. In this case, the message is ignored (there is no conflict), simply the counter of received messages (*$\mathtt{this}.\mathsf{counter}$*) is incremented. In the second branch,* $\mathtt{this}.\mathsf{id} < \mathsf{id}$*, the received colour is recorded to check the presence of conflicts. The value of* y *is added to* $\mathtt{this}.\mathsf{constraints}$ *and* $\mathtt{this}.\mathsf{counter}$ *is incremented by* 1*.*

If $r > \mathtt{this}.round$*, as in the last two branches, then the received message has been originated by a component executing a successive round and two possible alternatives are considered,* $\mathtt{this}.\mathsf{id} < \mathsf{id}$ *or* $\mathtt{this}.\mathsf{id} > \mathsf{id}$*. In both cases, round is set to* r*, send and counter are updated accordingly, and* $\mathtt{this}.\mathsf{constraints}$ *is set to the*

value of y if $\mathit{this}.\mathsf{id} < \mathsf{id}$.

$$
\begin{aligned}
T \triangleq\ & ((x = \text{“}try\text{”}) \wedge (\mathit{this}.\mathsf{id} > \mathsf{id}) \wedge (\mathit{this}.\mathsf{round} = z))(x, y, z).\\
& [\mathsf{counter} := \mathsf{counter} + 1]T\\
+\ & ((x = \text{“}try\text{”}) \wedge (\mathit{this}.id < id) \wedge (\mathit{this}.\mathsf{round} = z))(x, y, z).\\
& [\mathsf{counter} := \mathsf{counter} + 1,\ \mathsf{constraints} := \mathsf{constraints} \cup \{y\}]T\\
+\ & ((x = \text{“}try\text{”}) \wedge (\mathit{this}.\mathsf{id} > id) \wedge (\mathit{this}.\mathsf{round} < z))(x, y, z).\\
& [\mathsf{round} := z,\ \mathsf{send} := \mathsf{tt},\ \mathsf{counter} := 1,\ \mathsf{constraints} := \emptyset]T\\
+\ & ((x = \text{“}try\text{”}) \wedge (\mathit{this}.\mathsf{id} < id) \wedge (\mathit{this}.\mathsf{round} < z))(x, y, z).\\
& [\mathsf{round} := z,\ \mathsf{send} := \mathsf{tt},\ \mathsf{counter} := 1,\ \mathsf{constraints} := \{y\}]T
\end{aligned}
$$

Process D, reported below, is used to receive messages of the form $(\text{“}done\text{”}, c, r)$. *These are sent by components that have reached a final decision about their colour. When* $(\text{“}done\text{”}, c, r)$ *is received, we have that either* $\mathit{this}.\mathsf{round} \geq r$ *or* $\mathit{this}.\mathsf{round} < r$. *In the first case, the used colour is registered and the counter* $\mathit{this}.\mathsf{done}$ *is incremented. In the second case, private attributes are updated to indicate the startup of a new round.*

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

Process A, reported below, is used to manage the definitive selection of a colour and can only be executed when messages from all "pending" neighbours have been received ($\mathit{this}.\mathsf{counter} = |\mathit{this}.\mathsf{N}| - \mathit{this}.\mathsf{done}$*) and no conflict has been found (*$\mathit{this}.\mathsf{colour} \notin \mathit{this}.\mathsf{used} \cup \mathit{this}.\mathsf{constraints}$*). When the above conditions are satisfied, message* $(\text{“}done\text{”}, \mathit{this}.\mathsf{colour}, \mathit{this}.\mathsf{round} + 1)$ *is sent to neighbours, the* assigned *attribute is set to true, and the process terminates.*

$$A \triangleq \langle (\mathit{this}.counter = |\mathit{this}.\mathsf{N}| - \mathit{this}.\mathsf{done}) \wedge (\mathit{this}.\mathsf{colour} \neq \bot) \wedge (\mathit{this}.\mathsf{colour} \notin \mathit{this}.\mathsf{constraints} \cup \mathit{this}.\mathsf{used}) \rangle$$

$$(\text{“}done\text{”}, \mathit{this}.\mathsf{colour}, \mathit{this}.\mathsf{round} + 1)@(\mathit{this}.\mathsf{id} \in \mathsf{N}).[\mathsf{assigned} := \mathsf{tt}]0$$

□

Remark. Example 2.4 shows the main advantages of the attribute-based interaction. In essence, components are *infrastructure-agnostic*, i.e., they abstract from the underlying communication infrastructure and rely on *anonymous one-to-many* interaction pattern to communicate; this simplifies the design of component behaviours, because there is no need to manage the communication channels or the addresses on which components interact. The use of communication predicates, to derive the interaction between different components, permits programming *data centric* applications by taking into account the run-time features of the interacting components.

2.1. Expressiveness of the AbC Calculus

In this section, we do provide some evidence of the expressive power of *AbC* by showing how different communication models and interaction patterns can be easily modelled by using its primitives. Indeed, we believe that attribute-based communication can be used as a unifying framework to encompass a number of communication models. Further details regarding the actual implementation of the material presented in this section can be found in the webpage of *AbC*uS[5]: a Java API for the *AbC* calculus.

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

Encoding channel-based interaction. We show how one-to-many *channel-based interaction* can be encoded in the AbC calculus. In the former interaction pattern, a process sends a message on a specific channel name and all processes running in parallel and listening on the same channel should catch the message. A first idea would be to model channel names as AbC attributes, however this does not work. The reason is that in channel-based model, channels where the exchange happens are instantly enabled at the time of interaction and are disabled afterwards. This is not possible in AbC; attributes are persistent in the attribute environment and cannot be disabled at any time (i.e., attribute values are always available to be checked against sender predicates). Instead, the key idea of our encoding is to use structured messages to select communication partners where the name of the channel is the first element in the message; receivers only accept messages containing channels that match their receiving channels. Actually, attributes do not play any role in such interaction so we assume components with empty environments and interfaces i.e., $\emptyset{:}_{\emptyset}\, P$. In what follows, we use $[P]$ to denote $\emptyset{:}_{\emptyset}\, \mathsf{P}$. Thus a pair of processes, with one willing to receive on channel a and the other willing to send on the same channel, can be modelled as follows:

$$[(x = a)(x, y).P] \parallel [(a, msg)@(\mathtt{tt}).Q]$$

To show the feasibility of encoding broadcast channel-based calculi into AbC, we have encoded a calculus inspired by $b\pi$-calculus [40] into AbC. This calculus has been chosen 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 [11]. Furthermore, channels in this calculus can be communicated like in the point-to-point π-calculus [21] which is considered as one of the richest paradigms introduced for concurrency so far.

Based on the separation result of [41] it has been proven that broadcast and binary communication are incomparable in the sense that there does not exist any uniform, parallel-preserving translation from a broadcast into a binary model up to any "reasonable" equivalence. This is because in binary communication a process can non-deterministically choose the communication partner while in broadcast it cannot. Proving the existence of a uniform and parallel-preserving encoding of a channel-based broadcast calculus into AbC up to some reasonable equivalence ensures at least the same separation results between AbC and any binary calculus like the π-calculus. The full encoding, the formal definition which specifies what properties are preserved by this encoding, and a proof of its correctness up to a specific behavioural equivalence can be found in [36]. There we also argue that encoding AbC in channel-based interaction is not possible simply because such formalisms do not have explicit local state representation and a process cannot instantly inspect its local state while reacting to an incoming message. In AbC the behaviour of a process is parametric to its attribute environment and may send or receive messages based on the run-time values of its attributes.

Also *group-based* [42, 43, 10] and *publish/subscribe-based* [44, 45] interaction patterns can be naturally rendered in AbC. These interaction patterns, however, do not have formal descriptions and thus below we proceed by relying on examples. It is worth mentioning that existing group-based (e.g., Actors [42]) and publish/subscribe (e.g., topic-based [45]) frameworks rely on *totally loosely-coupled interaction* mechanisms where the order in which exchanged messages are delivered to interacting components is not unique. Such mechanisms are useful when communicating in distributed settings because they fully break the synchronisation dependencies between interacting components. Unfortunately they cannot cope well when coordination among behaviours is required which is the norm in collective-behaviour. For instance, implementing coordination protocols like the one in Example 2.1 in these frameworks becomes a very tedious and challenging task. This is because that the programmer has to manually manage such coordination by implementing a total order of message delivery, e.g., in Actors [42] Remote Procedure Call (RPC) is used. However, to implement such order one must to know at least the number of interacting components, but that contradicts the open-endedness principle of CAS. In AbC, coordination is a first class entity and it can be enforced syntactically, e.g., a process of the form $(v_1)@\Pi_1.(v_2)@\Pi_2.P$ ensures that the value v_1 is delivered to all receivers before v_2. Furthermore, if two components are concurrently enabled to send two messages, say m_1 and m_2, our interaction model ensures that all receivers equally receive these messages in one of these orders $m_1 \rightarrow m_2$ or $m_2 \rightarrow m_1$. Thus the order of message delivery is unique, i.e., if a component receive m_1 and m_2 in the order $m_1 \rightarrow m_2$ then all other components must receive them in the same order. Note that AbC primitives are totally oblivious to such details and delegate this role to the infrastructure that mediate the

interactions between components as we will explain in Section 5. This infrastructure can therefore establish a unique order of message delivery without involving the interacting components in the process.

Encoding group-based interaction. In the group-based model, when an agent wants to send a message to all elements of a group, it attaches the name or a reference to the group in the message and the message is propagated using this reference.

In the following, we show that when using a group name as an attribute in *AbC*, the constructs for joining or leaving a group can be modelled as attribute updates, like in the following example, where we assume that initially we have $\Gamma_1(group) = b$, $\Gamma_2(group) = a$, and $\Gamma_7(group) = c$:

$$\begin{aligned}\Gamma_1 :_{\{group\}} (msg)@(group = a).0 \parallel \\ \Gamma_2 :_{\{group\}} (group = \texttt{this}.group)(x).0 \mid \mathbf{set}(\texttt{this}.group, c)0 \parallel \ldots \\ \parallel \ \Gamma_7 :_{\{group\}} (group = \texttt{this}.group)(x).0 \mid \mathbf{set}(\texttt{this}.group, a)0\end{aligned}$$

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 as reported on the right hand side of the interleaving operator $\mid$. 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.

Encoding publish-subscribe. In the publish/subscribe model, there are two types of agents: publishers and subscribers and there is an exchange server that mediates their interactions. For instance, in topic-based publish/subscribe models [45], publishers produce messages tagged with topics and send them to the exchange server which is responsible for filtering and forwarding these messages to interested subscribers. Subscribers simply register their interests to the exchange server and receive messages according to their interests. 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 [45] into *AbC* can be obtained by allowing publishers to broadcast messages with true predicates (i.e., satisfied by all subscribers) and requiring subscribers to check compatibility of the exposed publishers attributes with their subscriptions, like in the example below:

$$\begin{aligned}\Gamma_1 :_{\{topic\}} (msg)@(\texttt{tt}).0 \parallel \Gamma_2 :_{\{subscription\}} (topic = \texttt{this}.subscription)(x).P \parallel \\ \ldots \parallel \Gamma_n :_{\{subscription\}} (topic = \texttt{this}.subscription)(x).Q\end{aligned}$$

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

We want to stress again that while *AbC* can be used to encode the group-based and the publish/subscribe-based paradigms the converse is not possible. The validity of this argument stems from the fact that the latter paradigms are more concerned with communication rather than coordination as the case with *AbC*. Clearly, coordination is more challenging and is concerned with maintaining a total order of message delivery. Furthermore, since *AbC* supports message passing, we can use its primitives to program applications with communication and coordination objectives.

In the next sections, we present a sizeable case study inspired by an industrial application of CAS. We focus on the role of the communication primitives and of the external environment in determining the communication between interacting components. Moreover, we use the case study to motivate the new communication primitives of *AbC* and to show that attribute-based communication is appropriate for handling CAS interactions. Additional case studies are presented in the technical report in [46]. In [46], we show how to program a crowd steering scenario and we also show how to program a swarm of robotics, performing a rescuing mission in a disaster arena.

3. Stable Allocation in Content Delivery Networks

This case study is based on the distributed stable allocation algorithm adopted by Akamai's Content Delivery Network (CDN) [34]. Akamai's CDN is one of the largest distributed systems in the world. It has

currently over 170,000 servers located in over 1300 networks in 102 countries and serves 15-30 % of all Web traffic. To avoid dealing with billions of clients individually, Akamai divides the clients of the global internet into groups, called *map units* each having a specific demand, based on their locations and traffic types. Also content servers are grouped into clusters, and each cluster is rated according to its capacity, latency, etc. Map units prefer highly rated clusters while clusters prefer low demand map units. The goal of global load balancing is to assign map units to clusters such that preferences are accounted for and constraints are met.

The allocation algorithm in [34] is a slight variant of the Stable Marriage Problem (SMP), reported in [47]. The goal of the original algorithm is to find a stable matching between two equally sized sets of elements given an ordering of preferences of each element of the two sets. Each element in one set has to be paired to an element in the opposite set in such a way that there are no two elements of different pairs which both would rather have each other than their current partners. When there are no such pairs of elements, the set of pairs is deemed stable. A natural and straightforward *AbC* implementation of the original stable marriage problem can be found in [31].

The variant considered in [34] allows (i) many map units to be assigned to a single cluster and (ii) map units to rank only the top dozen, or so, clusters that are likely to provide good performance. The first feature is a typical generalisation [48] of the original SMP, while the second is a mere simplification of the problem. Implementing these features in *AbC* does not pose any challenge, but it would make the example more verbose. Actually, our implementation of the original problem in [31] needs only to be extended to consider an extra attribute, named capacity, necessary for determining when a cluster should stop engaging with more map units. Moreover, the map units assigned to a cluster should be ordered according to their demands, so that a dissolve message goes first to the most demanding map units when necessary. Furthermore, these features do not add much to the original SMP; they still require map units and clusters to have predefined lists of preferences such that only ranked elements can participate in the algorithm. Obviously, this implies that one cannot take advantage of dynamic creation of new clusters.

In this article, we consider a more interesting variant of stable allocation that is better suited for the dynamicity of CDN. In this variant, the arrival of new clusters is considered, it is not required that elements know each other, and no predefined preference lists are assumed. Note that in these settings point-to-point interaction is not possible because elements are not aware of each other and the choice of implicit multicast is crucial. Indeed, in our variant, elements express their interests in potential partners by relying on their attributes rather than on their identities. In essence, an element of one set communicates with elements of the opposite set using predicates. Two parties that agree on some predicates form a pair, otherwise they keep looking for better partners. A pair splits only if one of its elements can find a better partner willing to accept its offer. In this way, preferences are represented as predicates over the values of some attributes of the interacting partners.

In this scenario, we consider the values of attributes *demand*, for a map unit, and *rating*, for a cluster, as a means to derive the interaction. To simplify the presentation, these attributes can take two different values: high (H) and low (L). Note that such setup does not simplify the problem itself as we will explain later. The reason is that the asymptotic complexity of our proposed solution is insensitive to the number of preference attributes and their evaluations. Of course we consider that such attributes assume a reasonably small number of values. An element in the system can be either a *Unit* or a *Cluster*. Units start the protocol by communicating with clusters in the quest of finding an element that satisfies their highest expectations. If no cluster accepts the offer, a unit lowers its expectations and proposes again until a partner is found. Clusters are always willing to accept proposals from any unit that enhances their levels of satisfaction. In case of a new arrival, some pairs of elements might dissolve if the new arrival enhances their levels of satisfaction. This means that not all pairs in the system are required to split on new arrivals; only those interested will do so. The system level goal (the emergent behaviour) is to construct a set of stable pairs from elements of different types by combining the behaviour of individual elements in the system through message passing. Mathematically speaking, the problem consists of computing a function at the system level by combining individual element behaviours, without relying on a centralised control. Note that since map units initiate the interaction and clusters only react, stable allocation is guaranteed and the solution is a "map-unit-optimal", as shown in [47], which is a property that fits with the CDN's goal of maximising performance for clients.

Allowing new arrivals is crucial to guarantee scalability and open-endedness while communicating based on predicates rather than on identities or ranks is crucial to deal with anonymity. The actual implementation of this algorithm alongside with experiment results can be found in the Webpage of GoAt[6]: a Go API for the AbC calculus.

The system in our attribute-based scenario can be modelled in AbC as the parallel composition of existing units and clusters (i.e., $\mathsf{Unit}_i \parallel \ldots \parallel \mathsf{Unit}_n \parallel \mathsf{Cluster}_i \parallel \ldots \parallel \mathsf{Cluster}_n$). Notice that units and clusters interact in a distributed setting without any centralised control. Each element is represented as an AbC component. A unit, Unit_i, has the form $\Gamma_I :_I \textsc{Pu}$ where Γ_i represents its attribute environment, I represents its interface where $I = \{\mathsf{demand}, \mathsf{id}_i\}$, and the process Pu represents its behaviour. A cluster, $\mathsf{Cluster}_i$, has the form $\Gamma_r :_{I'} \textsc{Pc}$ where Γ_r represents its attribute environment, I' represents its interface where $I' = \{\mathsf{rating}, \mathsf{id}_r\}$, and the process Pc represents its behaviour.

In addition to the attributes demand and rating, mentioned above, the attribute environments of units and clusters contain the following private attributes:

partner: current partner's identity; in case they are not engaged, the value is -1;

exPartner: previous partner;

id_i and id_r: the identity of units and clusters, respectively;

ref: current preference, 0 for high rating and 1 for low rating, initially $\mathsf{ref} = 0$;

success: a boolean attribute which is set to true when an accept message from a cluster is received, initially $\mathsf{success} = \mathsf{ff}$;

arrival: a boolean attribute which is set to true when an arrival message is received, initially $\mathsf{arrival} = \mathsf{ff}$;

dissolve: a boolean attribute which is set to true when a dissolve message is received, initially $\mathsf{dissolve} = \mathsf{ff}$;

rank: an integer attribute used to rank the current partner: 0 for high and 1 for low, initially $\mathsf{rank} = 2$;

bof: an integer attribute used to rank the new arrival: 0 for high and 1 for low. Initially $\mathsf{bof} = 2$;

lock: an integer attribute used to implement a lock within a single component, initially $\mathsf{lock} = 0$;

counter: a counter that is used to implement a monitor inside a component. Initially $\mathsf{counter} = \mathsf{threshold}+1$ where threshold is a constant number representing the number of computational steps a process can take before a specific event can occur.

The behaviour of a unit component is specified by the process Pu which is the parallel composition of the processes Pr, Mt, Mh, and Na. Process Pr defines a proposal process, process Mt defines a monitor process, process Mh defines a message handler process, and process Na defines a negative acknowledgment process. The behaviour of the proposal process Pr is defined below:

$$\begin{aligned}\textsc{Pr} \triangleq\ & \langle \mathsf{ref} = 0 \rangle (\text{“}propose\text{”},\ \texttt{this.demand},\ \texttt{this.id}_i,\ 0)@\Pi_h.[\mathsf{counter} := 0, \mathsf{dissolve} := \mathsf{ff}]\textsc{Ph} \\ & + \\ & \langle \mathsf{ref} = 1 \rangle (\text{“}propose\text{”},\ \texttt{this.demand},\ \texttt{this.id}_i,\ 1)@\Pi_l.[\mathsf{counter} := 0, \mathsf{dissolve} := \mathsf{ff}]\textsc{Ph}\end{aligned}$$

Process Pr sends a proposal message to all components that either satisfy predicate Π_h or predicate Π_l, depending on the current value of the ref attribute. The predicate Π_h represents high expectations where $\Pi_h = (\mathsf{rating} = \text{“}H\text{”})$ while the predicate Π_l represents low expectations where $\Pi_l = (\mathsf{rating} = \text{“}L\text{”})$. Note that the branches of process Pr encode the preferences of a unit and the selection of any of them depends on the run-time value of ref. These branches can be thought of as context-dependent behavioural variations

[6] https://github.com/giulio-garbi/goat

in Context-Oriented Programming [49]. Since the initial value of ref is 0, the process proceeds with the first branch. The proposal message contains a proposal label, "*propose*", the values of attributes demand, id_i, and ref of the unit respectively. The sent value of ref will be used later to decide if an accept message from a cluster is stale (i.e., the received value of ref is different from the current value of ref). By sending a proposal message, the counter and the dissolve attributes are reset. After this step, process PR evolves to PH which is a proposal handler process. Reseting the counter attribute will decide when process PR may propose again.

The monitor process MT is reported below:

$$\begin{aligned}\text{MT} \triangleq\ & \langle \mathsf{counter} < \mathsf{threshold}\rangle()@\mathsf{ff}.[\mathsf{counter} := \mathsf{counter}+1]\text{MT}\\ & +\\ & \langle \mathsf{counter} = \mathsf{threshold}\rangle()@\mathsf{ff}.[\mathsf{counter} := \mathsf{counter}+1,\ \mathsf{ref} := (\mathsf{ref}+1)\%2,\ \mathsf{success} := \mathsf{ff}]\text{MT}\end{aligned}$$

Process MT may increase the value of the counter attribute autonomously by sending a message on a false predicate (i.e., ()@ff) as long as its value is less than the constant threshold. As mentioned before these messages cannot be received by any components and thus behave as silent moves. By assuming fairness and a reasonably large value of threshold, we can guarantee that an accept message from a cluster, that satisfies the sent proposal message (if there is any), is received before the threshold is reached. If the threshold is reached, the monitor process increments the counter, adjusts the preference of the unit according to $(\mathsf{ref}+1)\%2$ and resets the value of the success attribute to false. Note that the adjustment $(\mathsf{ref}+1)\%2$ is used to account for new arrivals of clusters. Thus if the unit is not yet paired with any cluster after proposing with its lowest expectation (i.e., $\mathsf{ref}=1$) the unit is given the opportunity to try again and to propose with highest expectation (i.e., $\mathsf{ref}=(1+1)\%2=0$) until it finds a matching cluster. This is important because our algorithm relax the closed world assumption of [34] and thus we cannot assume that the number of units matches the number of clusters. This way we account for open world requirement of CAS in our solution. As we have seen so far, the code of processes is infrastructure-agnostic, i.e., it does not contain addresses or channel names. It is completely data-centric and relies on the run-time characteristics of the interacting partners.

In the proposal handler process, reported below, we can understand the role of the awareness construct $\langle\Pi\rangle$ as an environmental parameter used to influence the behaviour of a unit at run-time.

$$\begin{aligned}\text{PH} \triangleq\ & \langle \mathsf{lock}=0 \wedge \neg\mathsf{success} \wedge \mathsf{counter} > \mathsf{threshold}\rangle \text{PR}\\ & +\\ & \langle \mathsf{lock}=0 \wedge \mathsf{dissolve}\rangle \text{PR}\\ & +\\ & \langle \mathsf{lock}=0 \wedge \mathsf{arrival} \wedge (\mathsf{bof} \le \mathsf{rank}-1)\rangle\\ & \qquad (\text{“}\mathit{dissolve}\text{”})@(\mathsf{id}_r = \texttt{this.partner}).\\ & \qquad\qquad [\mathsf{arrival} := \mathsf{ff},\ \mathsf{success} := \mathsf{ff},\ \mathsf{ref} := \mathsf{bof},\ \mathsf{bof} := 2,\ \mathsf{rank} := 2,\\ & \qquad\qquad \mathsf{exPartner} := \mathsf{partner},\ \mathsf{partner} := -1]\text{PR}\end{aligned}$$

The process blocks executing until one of three events occurs. If no lock is acquired ($\mathsf{lock}=0$), no accept message from a cluster is received ($\neg\mathsf{success}$), and a threshold is reached ($\mathsf{counter} > \mathsf{threshold}$), process PH calls the proposal process PR again by considering the new value of ref modified by the monitor process. If no lock is acquired and a dissolve message from the current partner is received ($\mathsf{dissolve} = \mathsf{true}$), process PH calls the proposal process PR again. Finally, if no lock is acquired and an arrival message is received ($\mathsf{arrival} = \mathit{true}$), and the rank of the new arrival is better than the rank of the current partner ($\mathsf{bof} \le \mathsf{rank}-1$), the process sends a dissolve message that contains a dissolve label to its partner, sets the value of attribute exPartner to the value of attribute partner and the value of attribute ref to the value of attribute bof, and resets the values of attributes arrival, success, bof, rank, and partner to their initial values. Process PH calls the proposal process PR again.

The negative acknowledgment process NA is reported below:

$$\text{NA} \triangleq (x = \text{“}accept\text{”} \wedge ((z \neq \mathsf{ref}) \vee \mathsf{success}))(x,\ y,\ z).\\ (\text{“}ack\text{”},\ -1)@(\mathsf{id}_r = y).0 \mid \text{NA}$$

NA ensures that after a successful reception of a first accept message from a cluster (i.e., $\mathsf{success} = \mathsf{tt}$) all other accept messages to this unit are discarded (i.e., it sends an acknowledgement message with id=-1 which is interpreted by a cluster as a negative acknowledgement). Note that when the interleaving operator "|" occurs in the scope of a recursive call (e.g., the structure $P \triangleq a.Q|P$ for some action a and processes P and Q) this would be equivalent to a replication process, say $a^*.Q$. According to the semantics of AbC process $a^*.Q$ replicates itself every time action a is executed, i.e., $a^*.Q$ evolves to $Q|a^*.Q$ when action a is executed. Thus process NA in our scenario replicates itself every time an accept message is received to ensure that it is always able to catch incoming accept messages; the condition $(z \neq \mathsf{ref})$ is crucial to discard stale messages.

The message handler process MH is reported below:

$$\begin{array}{l}\text{MH} \triangleq\\ \quad \langle\neg\mathsf{success}\rangle(x = \text{“}accept\text{”} \wedge\ z = \mathsf{ref})(x,\ y,\ z).\\ \qquad [\mathsf{lock} := 1,\ \mathsf{success} := \mathsf{tt},\ \mathsf{rank} := \mathsf{ref}, \mathsf{exPartner} := \mathsf{partner}, \mathsf{counter} := \mathsf{threshold} + 1]\\ \qquad\quad (\text{“}ack\text{”},\ y)@\mathsf{tt}.[\mathsf{lock} := 0,\ \mathsf{partner} := y]\text{MH}\\ \quad +\\ \quad (x = \text{“}dissolve\text{”} \wedge \mathsf{id}_i = \mathsf{partner})(x).\\ \qquad [\mathsf{dissolve} := \mathsf{tt},\ \mathsf{success} := \mathsf{ff},\ \mathsf{ref} := 0,\ \mathsf{rank} := 2,\ \mathsf{partner} := -1]\text{MH}\\ \quad +\\ \quad \langle\mathsf{partner} \neq -1\rangle(x = \text{“}arrived\text{”} \wedge \mathsf{rating} = \text{“}H\text{”})(x).[\mathsf{arrival} := \mathsf{tt},\ \mathsf{bof} := 0]\text{MH}\\ \\ \quad +\\ \quad \langle\mathsf{bof} \neq 0 \wedge\ \mathsf{partner} \neq -1\rangle(x = \text{“}arrived\text{”} \wedge \mathsf{rating} = \text{“}L\text{”})(x).[\mathsf{arrival} := \mathsf{tt},\ \mathsf{bof} := 1]\text{MH}\end{array}$$

MH can respond to one of three events: if no accept message is received yet (i.e., $\neg\mathsf{success}$) and a new one arrives, the process receives the accept message only if the message is not stale ($z = \mathsf{ref}$), acquires a lock ($\mathsf{lock} := 1$), sets the value of attribute success to true, the value of attribute rank to the value of attribute ref, the value of attribute exPartner to the value of attribute partner and resets the counter. The process proceeds by sending an acknowledgement message to the cluster, releasing the lock, setting the value of attribute partner to the received cluster identity, and then the process continues as MH. Notice that the lock is important to ensure that the handler process PH does not proceed before all required attributes are assigned the right values. In this case, the first branch of process MH is executed atomically with respect to the co-located processes in the unit component.

If a dissolve message from the current partner is received, the process sets the value of attribute dissolve to true, resets the values of attributes success, ref, rank, and partner to their initial values and continues as MH. If an arrival message is received, the arrival attribute is set to true and the value of attribute bof is set to 0 if the attached attribute value is high $\mathsf{rating} = \text{“}H\text{”}$ otherwise the value of attribute bof is set to 1 and the process continues as MH. Notice that arrivals with low value of attribute rating cannot override the ones with high values. This is guaranteed by the condition $\mathsf{bof} \neq 0$ on the last branch. Also an arrival message can have an effect on the behaviour of a unit only if the unit is already engaged otherwise the arrival message is just discarded which is ensured by the condition $\mathsf{partner} \neq -1$ in the last two branches of process MH.

The coordination among processes running in a single unit is made possible by relying on shared attributes, awareness constructs, and attribute updates to implement proper lock mechanisms.

The behaviour of a cluster component is specified by the process PC which is the parallel composition of the processes AR, RH and DH. Process AR defines an arrival process, process RH defines a proposal

reception handler process, and process Dh defines a dissolve handler process. The arrival process Ar is defined below:

$$\text{Ar} \triangleq (\text{“}arrived\text{”})@\mathsf{tt}.0$$

This is the first process that takes a step when a cluster component wants to be involved in the allocation procedure. It simply sends an arrival message to all components in the system and then terminates. The arrival message contains an arrival label, “*arrived*”.

The proposal reception handler is reported below:

$$\text{Rh} \triangleq (x = \text{“}propose\text{”})(x,\ y,\ z,\ n).\text{R} \mid \text{Rh}$$

The process simply receives a proposal message and evolves to the reception process R. Note that process Rh replicates itself every time a message is received to ensure that no proposal is lost. The reception process R reported below, checks if no lock is acquired and the rank of the communicated value of attribute demand of the unit is less than the rank of the current partner ($\mathsf{rank}(y) < \mathsf{rank}$), the process sends an accept message to the unit where the proposal message came from, addressing it by its identity and acquires a lock, otherwise the process R terminates. The function $\mathsf{rank}(y)$ takes the value of demand as a parameter and returns 1 if $y =$ “H” and 0 otherwise. The accept message contains an acceptance label, “*accept*”, the identity of the current cluster, and the reference of the proposal message. The process then waits to receive either acknowledgement message or a negative one from the unit. If an acknowledgement is received, the value of attribute partner is set to the received identity of the unit, the rank is set to the value returned by $\mathsf{rank}(y)$, the lock is released and process R terminates. Note that a dissolve message is sent to the current partner before releasing the lock in case a cluster is already engaged. If a negative acknowledgement is received then the lock is released. The lock is needed to ensure that concurrent proposals are handled sequentially which is important to guarantee a consistent state of the attribute environment.

$$\begin{aligned}
&\text{R} \triangleq \langle \mathsf{lock} = 0 \rangle (\\
&\quad \textbf{if } (\mathsf{rank}(y) < \mathsf{rank}) \textbf{ then } \{ \\
&\qquad (\text{“}accept\text{”}, \mathsf{id}_r, n)@(\mathsf{id}_i = z).[\mathsf{lock} := 1] \\
&\qquad (\\
&\qquad \textbf{if } (\mathsf{partner} \neq -1) \textbf{ then } \{ \\
&\qquad\quad (e = \text{“}ack\text{”} \wedge f = \mathsf{id}_r \wedge \mathsf{id}_i = z)(e,\ f). \\
&\qquad\quad [\mathsf{exPartner} := \mathsf{partner},\ \mathsf{partner} := z,\ \mathsf{rank} := \mathsf{rank}(y)] \\
&\qquad\quad (\text{“}dissolve\text{”})@(\mathsf{id}_i = \texttt{this}.\mathsf{exPartner}).[\mathsf{lock} := 0]0 \\
&\qquad\quad \} \textbf{ else } \{ \\
&\qquad\quad (e = \text{“}ack\text{”} \wedge f = \mathsf{id}_r \wedge \mathsf{id}_i = z)(e,\ f). \\
&\qquad\quad [\mathsf{partner} := z,\ \mathsf{rank} := \mathsf{rank}(y),\ \mathsf{lock} := 0]0 \\
&\qquad\quad \} \\
&\qquad + \\
&\qquad (e = \text{“}ack\text{”} \wedge f \neq \mathsf{id}_r \wedge \mathsf{id}_i = z)(e,\ f).[\mathsf{lock} := 0]0 \\
&\qquad) \\
&\quad \} \textbf{ else } 0)
\end{aligned}$$

The dissolve handler process is reported below:

$$\begin{aligned}
\text{Dh} \triangleq (x = \text{“}dissolve\text{”} \wedge \mathsf{id}_i = \mathsf{partner})(x). \\
[\mathsf{rank} := 2,\ \mathsf{exPartner} := \mathsf{partner},\ \mathsf{partner} := -1]\text{Dh}
\end{aligned}$$

When a dissolve message from a cluster's partner is received, the value of attribute exPartner is set to the value of attribute partner, the values of attributes rank and partner are reset to their initial values, and the process calls itself.

Remark 3.1. *In [47], the authors showed that their algorithm terminates with a matching that is stable after no more than n^2 proposals, where n is the number of proposing elements, i.e., the algorithm has $O(n^2)$ worst-case complexity. In our variant, it should be clear that the worst case complexity is also $O(n^2)$ even after relaxing the assumptions of the original algorithm, i.e., no predefined preference lists and components are not aware of the existence of each other, so point-to-point communication is not possible. Interestingly, the complexity is still quadratic even if we consider a blind broadcast mechanism where proposals are sent to all components in the system except for the sender unit. In this way, for n-units, n-clusters, and a constant L related to the preferences of a unit where L is computed based on the number of branches in the proposal process* Pr*, we have that each unit can send at most $L * (2n - 1)$ proposals.*

The example below shows a possible execution of our variant of the stable allocation problem.

Example 3.1. *Let us consider two map units M_0 and M_1 and two clusters C_0 and C_1 with attribute environments defined as follows: $\Gamma_{m_0} = \{\mathsf{demand} = \text{“}H\text{”},\ \mathsf{id}_i = m_0\}$, $\Gamma_{m_1} = \{\mathsf{demand} = \text{“}L\text{”},\ \mathsf{id}_i = m_1\}$, $\Gamma_{c_0} = \{\mathsf{rating} = \text{“}H\text{”},\ \mathsf{id}_r = c_0\}$, and $\Gamma_{c_1} = \{\mathsf{rating} = \text{“}L\text{”},\ \mathsf{id}_r = c_1\}$. Below we show the execution of the stable allocation protocol, presented above, until a stable matching is reached.*

- *C_0 broadcasts an arrival message through process* Ar*.*
- *C_1 broadcasts an arrival message through process* Ar*. Because no unit is engaged yet, all arrival messages are discarded.*
- *M_0 proposes for a cluster with a high rating and waits for a response.*
- *C_0 receives the proposal because its rating is high.*
- *M_1 proposes for a cluster with a high rating and waits for a response.*
- *C_0 receives the proposal because its rating is high.*
- *C_0 handles the first proposal by process* R*. It acquires a lock and sends an accept message to M_0. Note that the second proposal cannot be handled before the lock is released.*
- *M_0 receives the message by process* Mh*, acquires a lock and sets the attributes success, rank and exPartner appropriately.*
- *M_0 sends an acknowledgement message to all possible components and includes the identity of C_0 in the message. It sets its partner to C_0 and releases the lock.*
- *Clearly, all components are potential receivers for this message, however, only C_0 receives it because it is the only component waiting an acknowledgement that contains its identity. By doing so, C_0 sets its partner to M_0, its rank to 1 (because $\mathsf{rank}(\text{“}H\text{”}) = 1$) and releases the lock.*
- *C_0 handles the second proposal by process* R *again. It acquires a lock and sends an accept message to M_1 because $\mathsf{rank}(\text{“}L\text{”}) = 0 < 1$, and waits for an acknowledgement.*
- *M_1 is the only component that can receive the accept message because it satisfies the sending predicate. It receives the message by process* Mh*, acquires a lock and sets the attributes success, rank and exPartner appropriately.*
- *M_1 sends an acknowledgement message to all possible components and includes the identity of C_0 in the message. It sets its partner to C_0 and releases the lock.*
- *C_0 receives the acknowledgement, sets its exPartner to current partner, its partner to M_1 and its rank to 0.*
- *C_0 sends a dissolve message to its exPartner (i.e., M_0) and releases the lock.*

- *M_0 receives the dissolve message by the second branch of process* Mh *which sets the dissolve attribute to true and resets the attributes* success, ref, rank, *and* partner *to their initial values. By doing so, the second branch of process* Ph *will be enabled and M_0 is ready to propose again.*
- *M_0 proposes again for a cluster with a high rating and waits for a response.*
- *C_0 receives the proposal because its rating is high.*
- *C_0 drops the proposal by the last branch of process* R*, because it is satisfied with its current partner, i.e., its* rank¡*1.*
- *No component can actually respond to the proposal. M_0 keeps waiting until a timeout occurs, i.e., the second branch of process* Mt *is enabled. The counter is reseted, the ref attribute value is set to 1 and the success attribute is turned off. M_0 can now propose for a cluster with a low rating. In this case, only C_1 accepts the proposal in a similar way as described before. Note that (M_0, C_1) and (M_1, C_0) are actually stable pairs.*

4. Discussion

In this section, we explain how *AbC* has helped us in dealing with the collective adaptive features that we introduced in Section 1.1 and also were evident in the previous case studies. We also comment on two interesting features of *AbC* code, namely *Compositionality* and *Code Extensibility*.

Distribution is naturally obtained because an *AbC* system has no centralised control; and it is represented as the parallel composition of independent components that can mutually influence each other only through message-exchange. For instance in Section 3, the overall system is defined as the parallel composition of existing units and clusters.

Awareness is supported by the attribute environment that plays a crucial role in orchestrating the behaviour of *AbC* components. It makes components aware of their own status and provides partial views of the surrounding environment. Components behave differently under different environmental contexts. This is possible because the behaviour of *AbC* processes is parametric with respect to the run-time attribute values of the component in which they are executed. The awareness construct, $\langle \Pi \rangle$, is used as an environmental parameter to influence the behaviour of *AbC* components at run-time. For instance, in Section 3, the behaviour of the proposal handler process, Ph, is totally dependent on the values of the attributes lock, success, counter, dissolve, arrival, bof, and rank. In case the value of just one of these attributes changes, the process will change its behaviour accordingly.

Adaptivity is obtained by means of the interaction predicates (both for sending and receiving) of *AbC* components that can be parametrised with their local attribute values; any run-time changes of these values might influence/change the possible set of interaction partners. Note that the target of the dissolve message, in the third branch of the proposal handler process Ph in Section 3, depends on the identity of the current partner (`this.partner`).

Interdependence among co-located processes can be obtained by modifying, with the attribute update construct, the attribute environment shared by processes within a single component. A branch of process Ph, in Section 3, is mainly selected depending on the attribute updates performed by the monitor process Mt and the message handler Mh.

Collaboration is obtained by combining individual component behaviours, through message exchange, to achieve a global goal for which a single component would not be sufficient. For instance, in Section 3, the goal was to construct a set of stable pairs without any centralised control. The goal was achieved by allowing different components to collaborate through message passing; each component contributed, and the combined behaviour of all components was necessary, to reach the overall system goal.

Anonymity is obtained 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. Thus the qualification needed to receive a message no longer depends on available channels or addresses, but on the values of dynamic attributes. For instance, in Section 3, the process Pr sends a proposal to a group

of clusters whose attributes satisfy a specific predicate (i.e., Π_h or Π_l). There is no prior agreement between the units and the other clusters and the set of candidate receivers is specified at the time of interaction in the sense that any change in the value of attribute ref changes the set of targeted components.

Scalability and Open-endedness are guaranteed by the adoption of multiparty communication instead of the binary one. Actually *AbC* supports an implicit multicast communication in the sense that the multicast group is not specified in advance, but rather it 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 the anonymity of interaction, breaks the synchronisation dependencies between senders and receivers; components can thus join and leave a system without disrupting its overall behaviour. For instance, new clusters, as defined by process Ar of Section 3, can join the allocation procedure at run-time without causing any disruption to the overall system behaviour. Actually, the third branch of process Ph of a unit is used as an adaptation mechanism to handle new arrival of clusters. Note that the condition $\mathsf{bof} \leq \mathsf{rank} - 1$ of that branch ensures that a unit responds only to new clusters that enhance its satisfaction. This means that new arrivals affect only specific components in the system.

It is worth mentioning that the initial behaviour of all components with the same type (e.g., Unit or Cluster in Section 3) is exactly the same. However, since this behaviour is parametrised with respect to the run-time attribute values of each component, components might exhibit different behaviours. Thus the context where a component is executing has a great impact on its behaviour, in the sense that specific behaviours can be enabled or disabled in presence of environmental changes. The behaviour of a component evolves based on contextual conditions; components do not need to have complex behaviour to achieve adaptation at system level, those behaviours can be achieved by combining local behaviours of individual components.

In addition to supporting the above distinguishing features of CAS, we have also that *AbC* code naturally supports compositionality and extensibility.

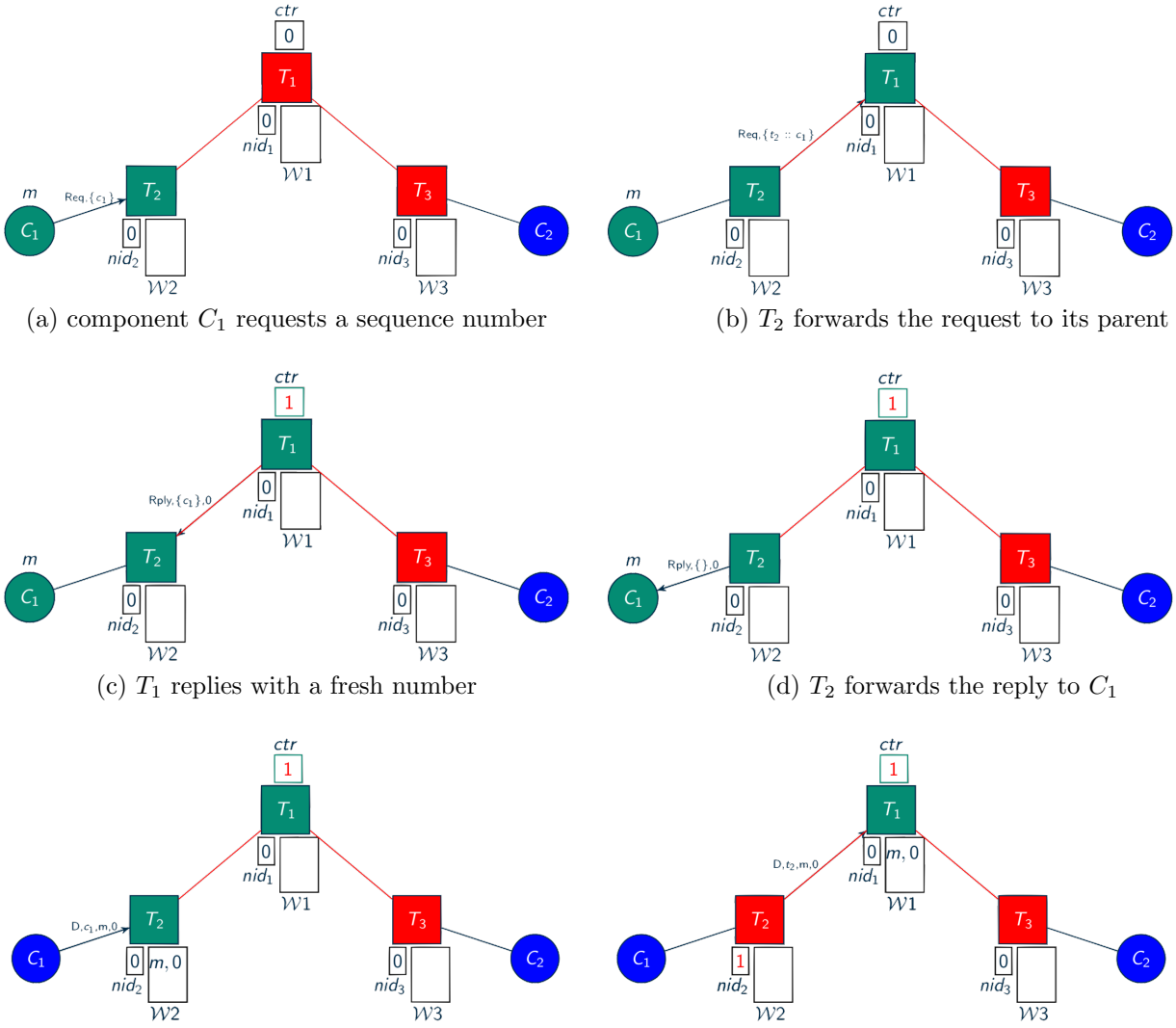
Compositionality: a component is the basic building block of *AbC* programs and any program can be broken down into a set of individual components which can only interact by exchanging messages. This simplifies verification because components can be analysed individually and their external behaviour can be assessed by their observable communication capabilities. Actually, we can abstract from the internal behaviour of an *AbC* component and only consider its observable behaviour when it interacts with other components. Thus, a component can be treated as a blackbox that acts and reacts to its environment. We refer interested readers to our theoretical results in [36].

Code Extensibility: *AbC* code can be easily extended in the sense that new alternative behaviours can be added and removed by modifying the attribute environments and the interfaces of components. Also component behaviours can be adapted without changing the internal structure of their running processes. For instance in Section 3, a unit can reverse the order of its preferences by directly modifying the required value of attribute ref in the awareness constructs and the following output actions of process Pr. Also if we add a public attribute to a cluster, say location, the code of a unit can be easily adapted to consider new possible preferences. One way is to add another alternative/branch, that considers the location of a cluster, to process Pr. Of course some attribute values have to be adjusted accordingly. As mentioned before, these branches can be thought of as context-dependent behavioural variations in Context-Oriented Programming [49].

Note that all of the above interesting properties of *AbC* code are only attainable as a result of adopting data centric interaction primitives. These primitives allow the programmer to base the interaction directly on the states of components rather than on their fixed addresses or on their low-level details. However, to be able to use such flexible primitives in real applications we need to realise them on a high-level programming language and (most importantly) on a coordination infrastructure that respects the properties of the composition operators of *AbC* as we will explain in the next section.

5. An Infrastructure supporting Attribute Based Interactions

In this section, we provide a distributed coordination infrastructure that realises the interaction primitives presented in Section 2. As mentioned in the introduction, *AbC* primitives are data centric and abstract from



(a) component C_1 requests a sequence number

(b) T_2 forwards the request to its parent

(c) T_1 replies with a fresh number

(d) T_2 forwards the reply to C_1

(e) C_2 sends m with its sequence number to T_2

(f) T_2 forwards the message since $nid_2 = 0$

Figure 1: The tree coordination infrastructure

the underlying coordination infrastructure that mediates the interaction. Actually, the semantics of *AbC* assumes atomic message-exchange and that a message is delivered to all components in a single step. Once the message is delivered, each component has the responsibility to use or discard the message. This provides a convenient interface to specify distributed systems from a high-level of abstraction without worrying about how these specifications are actually realised when implementing real applications. It is therefore crucial to design a coordination infrastructure to realise such high-level specifications in a reasonable way. Such infrastructure must break the atomic message-exchange assumption and allow components to execute asynchronously while establishing the communication links among them in an efficient way. It must also respect the properties of the composition operators in our calculus.

We introduce a distributed *tree-based* infrastructure in Google Go [50]. The choice of the infrastructure is based on our results in [35, 32] where we developed different implementations and evaluated their performance. Here we refine these results and present the most efficient and distributed API. Our infrastructure ensures that messages are delivered to components in an order that preserves their causal dependencies.

The approach consists of labelling each message with an id that is uniquely determined at the infras-

tructure level. Components work asynchronously while the semantics of the parallel composition operator is preserved by relying on the unique identities of exchanged messages.

The infrastructure is composed of a set of servers, organised as a logical tree. Each server can be connected to another server in the tree (its parent) and can interact with others in any part of the tree by only interacting with its parent or children. Each *AbC* component is connected to a server (possibly the root of the tree) that, in turns, manages multiple components. When a component wants to send a message, it asks its server for a fresh *id*. Only the root of the tree is responsible for sequencing messages. Hence, if it is the root of the tree, it replies with a fresh *id*, otherwise it forwards the message to its parent in the tree. Following this approach we are guaranteed that messages are ordered in the infrastructure and that each component receives messages in the appropriate order. *Out of order* messages are enqueued in waiting queue until the proper order is satisfied.

Fig. 1 illustrates the underlying logic of the tree infrastructure. Note that this figure does not include all details of the tree structure and is only intended to show its high-level behaviour. For instance input queues of the different nodes (where exchanged messages are stored and retrieved) are not detailed here and the examples in the figure assume that exchanged messages are handled immediately. Also tree agents that mediate the interaction between *AbC* components and tree servers are abstracted and components are assumed to communicate directly with the tree servers. We will explain these details in the rest of the section. *AbC* components are represented as circles and tree servers are represented as rectangles; each server has a counter, named *nid*, and a waiting queue to store sequenced messages. The waiting queue is a priority queue and it keeps the message with the least sequence number on top of it. The counter *nid* contains the sequence number of the next message to be handled. Normally the server comes to know the initial value of *nid* at the time when it registers to join the infrastructure. The root of the tree includes an additional counter, named *ctr* (initially equals to 0), that is used to sequence messages.

The subfigures (a)-(f) show a scenario where component C_1 wants to send a message m to other components in the tree, e.g., C_2. Note that C_1 is not aware of the existence of C_2 and can only communicate with its parent server T_2. Thus C_1 sends a request message $\mathsf{Req}, \{\mathsf{c_1}\}$ including its identity c_1 to T_2 (Fig. 1 (a)). Because T_2 is not the root of the tree it adds its identity on top of the list of addresses in the message and forwards the request $\mathsf{Req}, \{\mathsf{t_2} :: \mathsf{c_1}\}$ to its parent T_1 (Fig. 1 (b)). The root T_1 sends a reply $\mathsf{Rply}, \{\mathsf{c_1}\}, \mathsf{0}$ including the value of its counter *ctr* to T_2 and increments its counter (Fig. 1 (c)). Furthermore T_2 forwards the reply $\mathsf{Rply}, \{\}, \mathsf{0}$ to C_1 accordingly (Fig. 1 (d)). Note that the list of addresses $\{\mathsf{t_2} :: \mathsf{c_1}\}$, built during the trip of the request message to the root, is used to trace the reply back to C_1. Once C_1 receives the sequence number 0 it sends a data message $\mathsf{D}, \mathsf{c_1}, \mathsf{m}, \mathsf{0}$ containing its address, the *AbC* message and the sequence number to its parent T_2 and T_2 stores it accordingly in its waiting queue $\mathcal{W}_2$ (Fig. 1 (e)). The message stays in the waiting queue until it is ordered with respect to nid_2, i.e., the sequence number of the message matches nid_2. When this happens T_2 removes the message form its waiting queue and forwards it to all directly connected *AbC* components and server nodes and as a result T_2 increments its nid_2 (Fig. 1 (f)). The message is then propagated in the infrastructure until it reaches all connected components. Note that C_1 and C_2 may send messages concurrently and the root decides the order in which these messages are received by other components, depending on the fact which request of C_1 and C_2 reached the root before.

Note that since *AbC* components can be connected to any server in the tree (and yet communicate and coordinate their behaviour by only interacting with their directly connected server), the distribution principle of CAS is preserved. Furthermore, we want to stress that since the main message here is to provide a proof of concept on the feasibility of our programming approach we only deal with reliable infrastructures and we leave the reliability issue for a future work, which in anyway can be enforced by replicating the tree servers using standard approaches.

We have provided a Go programming API, named $\mathcal{G}o\mathcal{A}t$[7], that supports this infrastructure. We refer the reader to [35] for details about formal semantics, the proof of correctness of the proposed infrastructure, and its performance evaluation. In [35], we also compare the performance of the tree infrastructure with other kind of infrastructures, namely ring and cloud. The results show that the tree infrastructure outperforms the others in terms of average message delivery time and throughput.

[7] `https://giulio-garbi.github.io/goat/`

In what follows we discuss the Go implementation of the tree infrastructure and we also go through the $\mathcal{GoAt}$ programming API.

5.1. A Go Attribute-based Interaction API

$\mathcal{GoAt}$ [32] is a *distributed* programming API for supporting attribute-based interaction directly in Google Go [50]. Go is a new programming language, developed by Google to handle the complexities of networked systems and massive computation clusters, and to make working in these environments more productive. Go supports concurrency and inter-process communication through a compact set of powerful primitives and lightweight processes, called *goroutines*. It has an intuitive and lightweight concurrency model with a well-understood semantics. It extends the CSP model [51] with mobility by allowing channel-passing, like in π-calculus [21]. However, channel-passing in Go is possible only locally between goroutines. Go also supports buffered channels with a finite size. When the buffer size is 0, goroutines block execution and can only communicate by means of synchronisation. Otherwise, channels behave like mailboxes in Erlang which for interaction relies on identities rather than on channels. The generality and the clean concurrency model of Go make it an appropriate language for programming CAS. Thus, we integrated attribute-based interaction in Go via the *distributed* $\mathcal{GoAt}$ API to move the mobility of Go concurrency to the next level.

In this article, we use Go as an implementation language over Erlang (which supports the centralised *AbC* implementation *AErlang* [33]) to explore the capabilities of the Go language under the attribute-based interaction primitives. Also due to the fact that the concurrency model of Go is expressive enough to model the supported interactions of the actor model as mentioned above. Furthermore since we are not in a position to decide which language is better, we may only regard our choice as a good compromise. It would be interesting, for a future work, to find out what class of applications each language supports the best.

The projection of a $\mathcal{GoAt}$ system with respect to a specific component is reported in Fig. 2. It mainly consists of three parts: (i) infrastructure, (ii) agent, and (iii) component. The agent provides a standard interface between a $\mathcal{GoAt}$ component and the underlying coordination infrastructure and mediates message-exchange between them. Actually, this agent hides the details of the infrastructure from a component. An agent can be seen as a piece of software that handles the interaction between a component and the infrastructure server connected to it. In other words the agent decouples the behaviour of the component from the one of the underlying infrastructure. This way the behaviour of $\mathcal{GoAt}$ components is parametric to the underlying infrastructure that mediates the interaction.

Below we provide a brief description of the implementation details of the tree coordination infrastructure and the $\mathcal{GoAt}$ API.

The Tree Coordination Infrastructure. Our tree infrastructure is only responsible for forwarding messages to components and also for message-sequencing. It does not have access to the states of components and thus is not involved in filtering messages. Components decide to receive or discard messages based on the run-time values of their attributes. The tree infrastructure consists of a *registration node* and a set of servers organised in a logical tree. The *registration node* handles the construction of the infrastructure and the registration of components. When a component registers to the infrastructure through its agent, the registration node associates the agent to a specific server by assigning it communication ports to manage interaction with the selected server. The root of the tree is the only server that is responsible for generating sequence numbers. Each server is responsible for a group of agents and has its own input queue. The agent forwards its component messages to the input queue of its parent server. The server gets a message from its input queue: if it is a request message and the server is the root of the tree, the server assigns it a fresh id and sends a reply back to the requester, otherwise the server forwards the message to its parent until the message reaches the root. Every time a request message traverses a server, it records its address in a linked list to help trace back the reply to the original requester with a minimal number of messages. If the server receives a reply message, it will forward it to the address on top of the message's linked list storing the path. As a consequence, this address is removed from the linked list. Finally, when a data message is received, it is forwarded to all connected agents and servers except for the sender.

The method `NewTreeAgentRegistration(port, nodeAddresses)` is used to create a registration node. This method takes a port number and a set of node addresses. Furthermore the method `NewTreeNode(port,`

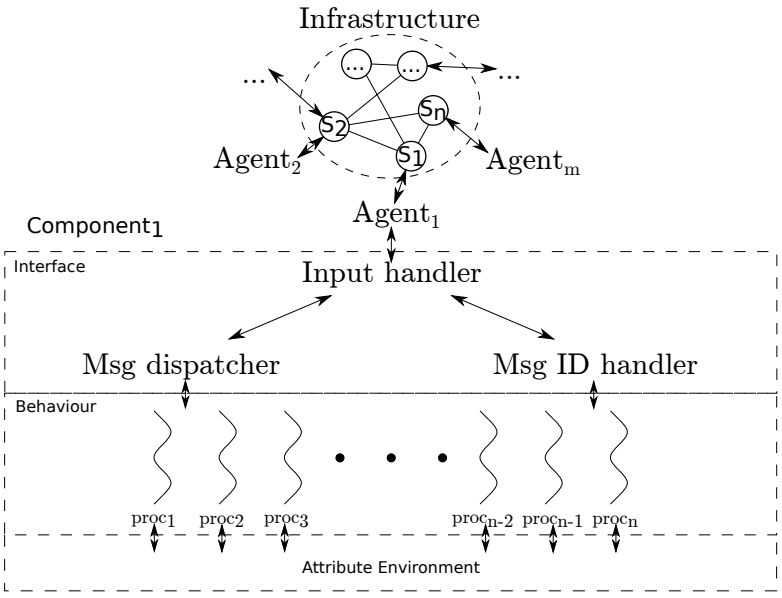


Figure 2: A Component interface to a *GoAt* system

`parentAddress, registrationAddress, childNodesAddresses)` is used to created a server node. It takes a port number, a parent address (it is empty for the root), the address of the registration node and the addresses of connected children (servers and components). We use the method `WorkLoop()` to register a server node to the infrastructure and starts its execution.

Example 5.1. *In this example we create a registration node on Machine* 1 *(Lines 2-6). We also create a server node on Machine* 2 *and register it to the registration node in the former machine (Lines 8-14).*

```
1
2   // Machine 1: creating a registration node on a machine with address "192.168.2.1"
3
4   port := 17000 // listening port
5   nodesAddresses := []string{} // initially empty list of server nodes addresses
6   go goat.NewTreeAgentRegistration(port, nodesAddresses).WorkLoop()
7
8   // Machine 2: creating a root server on another machine and register it to the registration node
9
10  port := 17002 // listening port
11  childNodesAddresses := []string{}  // initially empty
12  parentAddress := ""     // this address can be assigned later after registration
13  registrationAddress := "192.168.2.1:17000"
14  go goat.NewTreeNode(port, parentAddress, registrationAddress, childNodesAddresses).WorkLoop()
```

Note that the `go` *keyword (Line* 6 *and Line* 14*) is used to create lightweight go routines, assuming the role of server nodes.*

The Component. As reported in Fig. 2, a *GoAt* component consists of a behavioural part (represented by its running processes) and an interface (represented by its agent) to deal with the infrastructure's server connected to it. The interface consists of three entities: Input handler, Msg dispatcher and Msg ID handler.

The *Input handler* is used to collect all incoming messages from the infrastructure's server and to forward reply messages to the Msg ID handler.

The *Msg dispatcher* stores a message in the waiting queue of the component until all messages with smaller id have been sent/delivered. Once this condition is satisfied, the Msg dispatcher forwards the message to a process; if the process accepts, the message is considered as delivered, otherwise, the Msg dispatcher tries with another process. The procedure continues until either the message is accepted by some

process or all processes have rejected it. In both cases, the message is considered as delivered and the new id is notified to the Msg ID handler which updates the id of the next message to receive. It is important to note that any change to the attributes during the decision of accepting or rejecting the message can be only committed if the message is accepted, otherwise it will be rolled-back.

The *Msg ID handler* deals with requests of processes wanting to send a message, and provides them with fresh ids. The handler forwards the request to the infrastructure's server. While the process is waiting to send its message, dispatched messages are added to the waiting queue of the component. Once a reply message with a fresh id is received, the Msg ID handler forwards it to the process only when all messages with smaller id have been processed. The process can now manipulate the attributes environment and send a new message to the Msg ID handler which will forward it to the infrastructure's server. All attribute updates are committed and the msg dispatcher is notified about the new id.

The method `NewTreeAgent(registrationAddress)` is used to create a dedicated agent for a component and register it to the registration node of the infrastructure. Furthermore, method `NewComponent(agent, environment)` is used to generate a $\mathcal{GoAt}$ component and assign it an agent and an attribute environment. The attribute environment is simply a Go map from attribute identifiers to values.

Example 5.2. *For instance we can define a graph vertex component from the DGC Example 2.1 and register it to the registration node in Example 5.1 as follows:*

```
environment := map[string]interface{}{"round": 0, "used": {}, ...}
agent := goat.NewTreeAgent("192.168.2.1:17000")  // registration address
vertex := goat.NewComponent(agent, environment)
```

The method `NewProcess(Component).Run(func(proc *goat.Process)` is used to assign a behaviour to a $\mathcal{GoAt}$ component and also to start its execution. This method takes a $\mathcal{GoAt}$ process and executes it within the scope of the current component. The code inside the Run method represents the actual behaviour of a component. The generic behaviour of a $\mathcal{GoAt}$ process is implemented via a Go function `func(proc *goat.Process){body}`. This function takes a reference to a $\mathcal{GoAt}$ process and executes its body.

Example 5.3. *We now assign a behaviour to the graph vertex created in Example 5.1. Note that the behaviour of a graph vertex is the parallel composition of processes:* F, T, D *and* A *as explained early in Example 2.1. These processes are created inside the vertex (using* `Spawn`*) as follows:*

```
goat.NewProcess(vertex).Run(
        func(proc *goat.Process) {
                proc.Spawn(F)
                proc.Spawn(T)
                proc.Spawn(D)
                proc.Spawn(A)
            }
        )
```

5.2. The programming Interface

In this section, we briefly introduce the programming constructs of the $\mathcal{GoAt}$ API and show how they relate to the *AbC* primitives. For a detailed exposition of the $\mathcal{GoAt}$ implementation, we refer the reader to [32]. There we also provide an eclipse plugin[8] that is used to write specification in *AbC* syntax and automatically generate the corresponding Go code.

$\mathcal{GoAt}$ Components are parametric with respect to the infrastructure that mediates their interactions and the programmer needs only to know the registration address of components in the infrastructure as reported in the previous subsection. We provide the programmer with a small and high-level set of programming constructs to manipulate the attribute environments of components and also to specify their actual behaviours.

[8] `https://github.com/giulio-garbi/goat-plugin`

Components attributes can be retrieved and set via the methods `Comp(attribute)` and `Set(attribute, value)` respectively. The method `NewProcess(Component).Run(func(proc *goat.Process)` (as explained before) assigns a behaviour to a $\mathcal{GoAt}$ component and also starts its execution. This method takes a $\mathcal{GoAt}$ process `proc` and executes it within the current component. Note that a process can also be a parallel composition of other processes. This is implemented using the `Spawn(proc *goat.Process)` method which is used to dynamically create a process and execute it in parallel with the main process at run time. Though this does not correspond to an explicit AbC command, it is a natural macro in AbC where the interleaving operator occurs in the scope of a process definition, i.e., $P \triangleq a.Q|P$ as we explained in Section 3.

The behaviour of a $\mathcal{GoAt}$ process is implemented via a Go function `func(proc *goat.Process) {proc.Cmd1...proc.Cmdn}` that takes a reference to a $\mathcal{GoAt}$ process and executes its commands. Note that beside $\mathcal{GoAt}$ commands, which will be explained later, the usual loop and branching statements of Go can also be used. For instance we implement AbC recursive definitions using infinite loops `for{}`.

The main interaction actions, send and receive, are implemented via `Send(Tuple, Predicate)` and `Receive(accept func(attr *Attributes, msg Tuple) bool)` respectively. The rendering of a send command is obvious while the receive command accepts a message and passes it to a function that checks whether it satisfies the receiving predicate. We also use a guarded send command with atomic attribute updates, i.e., `GSendUpd(Guard, Tuple, Predicate, updFnc)`. In addition to the normal send command, this one is only activated when its guard evaluates to true and as a result of its execution a possible sequence of attribute updates may apply.

The command Call(Process) implements a process call while the awareness operator is implemented via the command `WaitUntilTrue(Predicate)`. Finally, the non-deterministic choice of several guarded processes are implemented via the command `Select(cases ...selectcase)`. This command takes a finite number of arguments of type `selectcase`, each of which is composed of an action guarded by a predicate and a continuation process, i.e., `Case(Predicate, Action, Process)`. When the guarding predicate of one branch is satisfied, the method enables it and terminates the other branches.

AbC predicates are implemented via `Equals, And, Belong`, and `Not` which correspond to $=$, $\wedge$, $\in$ and $\neg$ respectively. Other standard predicates are also supported. We use `Receiver(a)` in the sender predicate to indicate that we are evaluating the predicate based on the value of attribute `a` in the receiver side. For instance, the predicate `Belong(goat.Comp("id"), goat.Receiver("N"))` is equivalent to `this.id` $\in$ N. We also use `Evaluate(expression)` to evaluate an expression under the attribute environment of a component.

Example 5.4. *Below, we show how to program process* F *of Example 2.4 in the $\mathcal{GoAt}$ API. Despite the verbosity of the $\mathcal{GoAt}$ syntax, the correspondence in terms of syntax with respect to* AbC *is evident.*

```
1  func F (proc *goat.Process) {
2      for {
3          proc.GSendUpd(
4              goat.And(goat.Equals(goat.Comp("assigned"), false), goat.Equals(goat.Comp("send_try"), true)),
5              goat.NewTuple("try", goat.Evaluate(minColorNot, goat.Comp("used")), goat.Comp("round"), goat.Comp("id")),
6              goat.Belong(goat.Comp("id"), goat.Receiver("N")),
7              func(attr *goat.Attributes){
8                  attr.Set("colour", minColorNot(attr.GetValue("used")))
9                  attr.Set("send_try", false)
10                 }
11         )
12     }
13 }
```

6. Related Work

In this section we touch on related works concerning languages and calculi with primitives that model either collective interaction, or multiparty interaction. We also report on other approaches to interaction in distributed systems and show how they relate to AbC.

Several frameworks have been proposed to target the problem of collective (or ensemble) formation. These approaches usually differ in the way they represent collectives and in their generality.

The SCEL language [15], proposed in the ASCENS project [52], is designed to program autonomic computing systems. It combines different mechanisms from different communication paradigms. For instance, it combines message passing, pattern matching, tuple spaces and interaction policies. It is actually a meta-language that can be instantiated to serve different purposes. *AbC* on the other hand is based on a set of primitives to model CAS interactions. Furthermore and as opposed to *AbC*, SCEL's concurrency model is complicated and is not equipped with a behavioural theory to be able to reason about the interactions of collective-adaptive systems.

DEECo [53] and Helena [54] are component-based frameworks that handle ensemble (or collective) formation, component cooperation and knowledge distribution concerns at the level of software architecture. They model ensemble as a first class entity. An ensemble determines its member components and their roles and thus ensembles can be considered as distribution units that centrally control their members. It is very hard to push this idea into action to develop complex systems where ensembles may overlap, be nested, dynamically formed and dismantled in a distributed environment. In *AbC* these architectural complexities are avoided by relying on dynamic interpreted predicates and thus providing a clean way to specify nested and overlapping ensembles at runtime.

Blackboard design pattern [55] is an architecture-based approach that adopts a Linda-like [56] interaction pattern. Actually they are being used in various industry tuple-space architectures, such as IBM TSpaces [57]. A blackboard architecture consists of different independent agents implementing parts of the application logic and interact among each other by using the blackboard component. The latter is a data structure that is used as the general communication and coordination mechanism and is managed by a controller component. Note that an architecture typically has multiple blackboards and agents may move from one to another at run-time. This overwhelms the controllers that have to shepherd the interaction among agents. Since the interactions among agents is mediated by the blackboard and agents may move between blackboards while engaging in conversations, these controllers have to keep track of the new locations of agents to be able to forward them messages (sent to them in previous locations). In *AbC* we avoid the complexity of managing architectural issues arising as side-effects of mobility, by adopting an architecture-free interaction model. We assume a flat infrastructure and thanks to predicates we can discipline the interaction dynamically in a straightforward way.

The field calculus [13] relies on a notion of computational fields to describe distributed systems from a system or an aggregate-level. A computational field can be considered as an aggregate-level distributed data structure that is maintained by physically distributed components. These components manage a global and dynamically evolving computational fields by defusing, recombining and composing information injected by one or a few other components. This is done iteratively in asynchronous computational rounds, consisting of message reception from neighbours, computing the local value of the fields, and spreading messages to neighbours. The field calculus is useful for implementing self-organising systems that adaptively regulate their behaviours in response to changes in the surrounding environment. Note, however, that the field calculus is designed specifically for spatially distributed systems and has assumptions on the distance between devices. While the field calculus approach might not be optimal for sparse physically distributed systems, it is very useful for dense ones like sensor networks. In *AbC* we describe systems from the individual point of view. These individual behaviours are combined to engineer the behaviour at system-level. Furthermore, *AbC* assumes no restrictions on the distance between devices or even on their existence.

Other well-known approaches to program distributed systems include: channel-based models [58, 51, 21], group-based models [42, 43, 10], and publish/subscribe models [44, 45]. 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., binary communication CCS [58], multiway synchronisation CSP [51], and broadcast communication CBS [59]. The π-calculus [21] was developed as a way to mitigate this problem by allowing the exchange of channel names and thus providing dynamic interfaces and additional flexibility. However, the dynamicity of interfaces is still limited because even if generic input or output actions are allowed, they are disabled until they are instantiated with specific channel

names. This means that a process can engage in a communication only when its actions are enabled. The broadcast $b\pi$-calculus [41, 40] is based on CBS and the π-calculus in that it extends the former with a channel-passing mechanism. Furthermore, π-calculus and most process calculi rely on synchronous communication that creates dependencies between senders and receivers and affects the overall scalability of the system. AbC actions are always enabled with respect to the current attribute values of the component where they are executing. When these values change, the interaction predicates change seamlessly and become available for other communication partners.

The class of mobile ambient calculi [60, 61] aim to model mobile computing systems both in terms of process- and device-mobility (in the physical sense). An ambient defines a scope of interactions. A process can enter/leave an ambient. Also ambients can be nested and dissolved dynamically at run-time. These calculi deal with mobility in terms of physical location, rather than coding it in terms of wiring as in channel-based calculi. However, the evolution of ambient structures is complicated and is hard to trace and control when the system gets larger; and thus greatly affects code readability and extensibility. In AbC we abstract from the physical movement of devices and abstract the surrounding environment by means of local attributes, i.e, some attributes might get their values from sensors. Since the behaviour of a component is parametric to these attributes, any changes in their values would be reflected in the component behaviour. Thus physical movement can be accounted for in AbC models while maintaining a clean structure of localities.

In *group-based* models, like the one used for IP multicast [10], the group is specified in advance, in fact the reference address of the group is explicitly included in the message. Groups or spaces in the ActorSpace model [42] are regarded as passive containers of processes (actors) and may be created and deleted with explicit constructs. Spaces may be nested or overlap and can be created dynamically at run-time. Actually, the notion of space is a first class entity in the ActorSpace model. AbC extends the ActorSpace pattern-matching mechanism to select partners by predicates from both sides of the communication, because not only the sender can select its partner but also the receiver can decide to receive or discard messages. The notion of spaces/collectives in AbC is indeed more abstract and only specified at run-time.

Coordinated Actors [14] is a framework that relies on Actors [8] as the main units of concurrency. As opposed to AbC in which we design system goals offline, this framework also relies on run-time techniques [62]. It is based on the idea that a self-adaptive system is realised through a collection of (centralised) feedback loops (i.e., MAPE-K [63]) to control adaptation of the system. A feedback loop consist of Monitor, Analyze, Plan and Execute components together with a knowledge part. The knowledge consists of information about both the system and the environment. To guarantee correctness of the overall behaviour this technique is applied both offline and online. Thus an abstract model of the system and the environment is stored in the knowledge component and can be updated and analysed periodically to check for correctness and also to be used for re-planning. The main novelty of this approach is that it provides one solution to construct the MAPE-K loops and models@runtime while providing runtime analysis to detect or predict violation of system's goals. Clearly, this is an engineering framework that combines different techniques to deal with the complexities of collective behaviour. AbC, on the other hand, is a core computational framework that provides a minimal set of primitives to be used to program such behaviours at design time.

Another Actors-based [8] specification language is bRebecca [64] and it is introduced to model complex interactions among distributed systems. bRebecca actors can interact with each other via *asynchronous anonymous broadcast communication*. The focus of bRebecca is more on system specifications than on system programming. Indeed, specific abstraction mechanisms are introduced to mitigate the state space explosion problem and to enable the use of model-checking techniques. However, in bRebecca a sender is not able to select the class of receivers. Differently, in AbC, thanks to the use of predicates, both senders and receivers can identify the counterparts of an interaction. We would like to mention that in general AbC is closer to Actors than other Process Algebra approaches. In fact, messages can be used in AbC to communicate components ids and thus use them in the same way of the Actor model.

In the *publish/subscribe* model, like the one used in NASA Goddard Mission Services Evolution Centre (GMSEC)[9], each component can take the role of a publisher or a subscriber. Message propagation is obtained

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

by introducing an exchange server that mediates the interaction; it stores the subscriptions of subscribers, receives messages from publishers, and forwards them to the appropriate 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, scalability issues are moved to the implementation 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. This is not the case in our implementation of the underlying infrastructure for *AbC*. Our infrastructure is completely distributed and does not filter messages and thus does not have to deal with evolving subscriptions. It only serves as a forwarding service. The publish/subscribe model can be considered a special case of *AbC* where publishers can attach attributes to messages and send them with empty predicates (i.e., satisfied by all). Only subscribers can check the compatibility of the attached publishers attributes with their subscriptions.

AErlang [33] is a middleware enabling attribute-based communication among programs in *Erlang*, a concurrent functional programming language originally designed for building telecommunication systems and recently successfully adapted to broader contexts. *AErlang* is not, however, a faithful implementation of *AbC*; it can be considered more as an extension of Erlang with linguistic abstraction for modelling attribute based communication. The latter mentioned semantical mismatch is very critical because it hinders the possibility of taking advantage of the formal semantics of *AbC* for proving programs properties. As opposed to $\mathcal{GoAt}$, AErlang only supports a centralised coordination infrastructure while $\mathcal{GoAt}$ supports different distributed coordination infrastructures, i.e., tree-based, ring-based and cluster-based infrastructures. It would be interesting, though, to implement the proposed distributed infrastructure in Erlang and compare it with $\mathcal{GoAt}$ in terms of intuitiveness and performance. The choice to provide a Go implementation as an alternative to *AErlang* [33] is due to two main reasons. Firstly, to the expressiveness of the Go concurrency model and to our confidence that this language will have a substantial impact in future development of distributed applications. Secondly, we do believe that it is currently essential to experiment with attribute based extension of different full fledged programming languages in order to assess which of them best support the development of Collective adaptive Systems trough further experiments and to evaluate the performance of different implementations and approaches.

Programming collective and/or adaptive behaviour has also been studied in different research communities like those interested in Context-Oriented Programming (COP) [49] and in the Component-Based approach [65]. In Context-Oriented Programming (COP), a set of linguistic constructs is used to define context-dependent behavioural variations. These variations are expressed as partial definitions of modules that can be overridden at run-time 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 behaviour by considering the run-time changes of the values of their attributes which might be triggered by either contextual conditions or by local interaction. Another approach that considers behavioural variations is adopted in the Helena framework [12].

The component-based approach, represented by FRACTAL [65] and its Java implementation, JULIA [65], 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.

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 manipulating the environment gives *AbC* greater flexibility and expressiveness while keeping *AbC* models as natural as possible.

7. Concluding Remarks

We have proposed a language-based approach for programming the interaction of collective-adaptive systems by relying on attribute-based communication. We have introduced the programming constructs of the *AbC* calculus and we exploited them to show how complex and interesting case studies, from the realm of collective-adaptive systems, can be programmed in an intuitive way. We illustrated the expressive power of attribute-based communication by showing the natural encoding of other existing communication paradigms. We argued that the general concept of attribute-based communication can be exploited to provide a unifying framework to encompass different communication models and interaction patterns. Since the focus of this article was to show the expressive power of attribute-based communication and their applicability in the context of CAS systems, we refrained from presenting theoretical results. However, full details about behavioural theory, equational laws, and a formal proof of encoding can be found in [36]. We also provided an *AbC* API to enable attribute-based interaction in real programming languages, e.g., the $\mathcal{GoAt}$ programming API in Google Go [32].

For future work, we plan to develop formal tools based on *AbC* semantics to analyse the generated code of our *AbC* APIs for ensuring safety and liveness properties. We want to study the possibility of using static analysis to discipline the interaction in *AbC* and thus producing a correct by construction programs and we will also consider the more challenging problem of specifying and verifying collective properties of *AbC* programs. The ReCiPe framework [66] (an extension and symbolic representation of *AbC* specifications) is the first step in this direction where also the linear-time temporal logic LTL is extended to be able to specify collective properties.

References

[1] M. Broy, The 'grand challenge' in informatics: engineering software-intensive systems, Computer 39 (10) (2006) 72–80. doi:10.1109/MC.2006.358.

[2] H. Kopetz, Internet of Things, Springer US, Boston, MA, 2011, pp. 307–323. doi:10.1007/978-1-4419-8237-7_13.

[3] A. 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, ACM, 2015, pp. 893–895.

[4] S. Anderson, N. Bredeche, A. Eiben, G. Kampis, M. van Steen, Adaptive collective systems: Herding black sheep, Bookprints, 2013.

[5] M. Wirsing, M. M. Hölzl, N. Koch, P. Mayer (Eds.), Software Engineering for Collective Autonomic Systems - The ASCENS Approach, Vol. 8998 of Lecture Notes in Computer Science, Springer, 2015. doi:10.1007/978-3-319-16310-9.

[6] M. Wirsing, M. Holzl, Software intensive systems, Draft Report on ERCIM Beyond the Horizon Thematic Group 6.

[7] R. Behjati, S. Nejati, L. C. Briand, Architecture-level configuration of large-scale embedded software systems, ACM Trans. Softw. Eng. Methodol. 23 (3) (2014) 25:1–25:43. doi:10.1145/2581376.

[8] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.

[9] D. Sangiorgi, D. Walker, PI-Calculus: A Theory of Mobile Processes, Cambridge University Press, New York, NY, USA, 2001.

[10] H. W. Holbrook, D. R. Cheriton, Ip multicast channels: Express support for large-scale single-source applications, in: ACM SIGCOMM Computer Communication Review, Vol. 29, ACM, 1999, pp. 65–78.

[11] K. V. S. Prasad, A calculus of broadcasting systems, in: S. Abramsky, T. S. E. Maibaum (Eds.), TAPSOFT '91, Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 338–358.

[12] R. Hennicker, A. Klarl, Foundations for ensemble modeling - the helena approach - handling massively distributed systems with elaborate ensemble architectures, in: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi, 2014, pp. 359–381. doi:10.1007/978-3-642-54624-2_18.

[13] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Trans. Model. Comput. Simul. 28 (2) (2018) 16:1–16:28. doi:10.1145/3177774.

[14] M. Bagheri, I. Akkaya, E. Khamespanah, N. Khakpour, M. Sirjani, A. Movaghar, E. A. Lee, Coordinated actors for reliable self-adaptive systems, in: O. Kouchnarenko, R. Khosravi (Eds.), Formal Aspects of Component Software - 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers, Vol. 10231, 2016, pp. 241–259. doi:10.1007/978-3-319-57666-4_15.

[15] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: the scel language, ACM Transactions on Autonomous and Adaptive Systems (2014) 1–29.

[16] D. Sanderson, N. Antzoulatos, J. C. Chaplin, D. Busquets, J. Pitt, C. German, A. Norbury, E. Kelly, S. Ratchev, Advanced manufacturing: An industrial application for collective adaptive systems, in: 2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, 2015, pp. 61–67. doi:10.1109/SASOW.2015.15.

[17] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, R. Paige, Large-scale complex it systems, Communications of the ACM 55 (7) (2012) 71–77.

[18] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al., Software engineering for self-adaptive systems: A research roadmap, in: Software engineering for self-adaptive systems, Springer, 2009, pp. 1–26.

[19] A. B. Lewko, B. Waters, Decentralizing attribute-based encryption, in: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings, 2011, pp. 568–588. doi:10.1007/978-3-642-20465-4_31.

[20] A. Margheri, M. Masi, R. Pugliese, F. Tiezzi, A rigorous framework for specification, analysis and enforcement of access control policies, IEEE Trans. Software Eng. 45 (1) (2019) 2–33. doi:10.1109/TSE.2017.2765640.
URL https://doi.org/10.1109/TSE.2017.2765640

[21] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, ii, Information and computation 100 (1) (1992) 41–77.

[22] D. E. Jackson, F. L. Ratnieks, Communication in ants, Current Biology 16 (15) (2006) R570 – R574. doi:https://doi.org/10.1016/j.cub.2006.07.015.

[23] J. Baeten, D. Van Beek, J. Rooda, Process algebra for dynamic system modeling, Eindhoven University of Technology, The Netherlands, Tech. Rep. CS-Report (2006) 06–03.

[24] M. J. Wooldridge, Reasoning about rational agents, Intelligent robots and autonomous agents, MIT Press, 2000.

[25] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989, 1989, pp. 179–190. doi:10.1145/75277.75293.

[26] B. Finkbeiner, S. Schewe, Uniform distributed synthesis, in: 20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings, 2005, pp. 321–330. doi:10.1109/LICS.2005.53.

[27] F. Bergenti, G. Rimassa, M. Viroli, Operational semantics for agents by iterated refinement, in: Declarative Agent Languages and Technologies, First International Workshop, DALT 2003, Melbourne, Australia, July 15, 2003, Revised Selected and Invited Papers, 2003, pp. 37–53. doi:10.1007/978-3-540-25932-9_3.

[28] F. S. de Boer, W. de Vries, J. C. Meyer, R. M. van Eijk, W. van der Hoek, Process algebra and constraint programming for modeling interactions in MAS, Appl. Algebra Eng. Commun. Comput. 16 (2-3) (2005) 113–150. doi:10.1007/s00200-005-0173-0.

[29] D. Kinny, Algebraic specification of agent computation, Appl. Algebra Eng. Commun. Comput. 16 (2-3) (2005) 77–111. doi:10.1007/s00200-005-0172-1.

[30] M. Viroli, A. Omicini, Process-algebraic approaches for multi-agent systems: an overview, Appl. Algebra Eng. Commun. Comput. 16 (2-3) (2005) 69–75. doi:10.1007/s00200-005-0170-3.

[31] Y. A. Alrahman, R. De Nicola, M. 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, Springer, 2016, pp. 539–553. doi:10.1007/978-3-319-47166-2_38.

[32] Y. A. Alrahman, R. De Nicola, G. Garbi, *GoAt*: Attribute-based interaction in google go, in: Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III, 2018, pp. 288–303. doi:10.1007/978-3-030-03424-5_19.

[33] R. De Nicola, T. Duong, O. Inverso, C. Trubiani, Aerlang: Empowering erlang with attribute-based communication, Sci. Comput. Program. 168 (2018) 71–93. doi:10.1016/j.scico.2018.08.006.

[34] B. M. Maggs, R. K. Sitaraman, Algorithmic nuggets in content delivery, SIGCOMM Comput. Commun. Rev. 45 (3) (2015) 52–66. doi:10.1145/2805789.2805800.

[35] Y. A. Alrahman, R. De Nicola, G. Garbi, M. Loreti, A distributed coordination infrastructure for attribute-based interaction, in: Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings, 2018, pp. 1–20. doi:10.1007/978-3-319-92612-4_1.

[36] Y. A. Alrahman, R. D. Nicola, M. Loreti, A calculus for collective-adaptive systems and its behavioural theory, Information and Computation 268 (2019) 104457. doi:https://doi.org/10.1016/j.ic.2019.104457.

[37] Y. A. Alrahman, R. De Nicola, M. Loreti, On the power of attribute-based communication, in: Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, 2016, pp. 1–18. doi:10.1007/978-3-319-39570-8_1.

[38] T. R. Jensen, B. Toft, 25 pretty graph colouring problems, Discrete Mathematics 229 (1-3) (2001) 167–169. doi:10.1016/S0012-365X(00)00206-5.

[39] B. Hlldobler, E. Wilson, The Superorganism: The Beauty, Elegance, and Strangeness of Insect Societies, 1st Edition, W.W. Norton & Company, 2008.

[40] C. Ene, T. Muntean, A broadcast-based calculus for communicating systems, in: Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001, 2001, p. 149. doi:10.1109/IPDPS.2001.925136.

[41] C. Ene, T. Muntean, Expressiveness of point-to-point versus broadcast communications, in: Fundamentals of Computation Theory, 12th International Symposium, FCT '99, Iasi, Romania, August 30 - September 3, 1999, Proceedings, 1999, pp. 258–268. doi:10.1007/3-540-48321-7_21.

[42] G. Agha, C. J. Callsen, Actorspaces: An open distributed programming paradigm, in: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May

19-22, 1993, 1993, pp. 23–32. doi:10.1145/155332.155335.

[43] G. V. Chockler, I. Keidar, R. Vitenberg, Group communication specifications: a comprehensive study, ACM Comput. Surv. 33 (4) (2001) 427–469. doi:10.1145/503112.503113.

[44] M. A. Bass, F. T. Nguyen, Unified publish and subscribe paradigm for local and remote publishing destinations, US Patent 6,405,266 (2002).

[45] P. T. Eugster, P. Felber, R. Guerraoui, A. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131. doi:10.1145/857076.857078.

[46] Y. A. Alrahman, R. De Nicola, M. Loreti, Programming the interactions of collective adaptive systems by relying on attribute-based communication, CoRR abs/1711.06092. arXiv:1711.06092.

[47] D. Gale, L. S. Shapley, College admissions and the stability of marriage, The American Mathematical Monthly 120 (5) (2013) 386–391. doi:10.4169/amer.math.monthly.120.05.386.

[48] K. Iwama, S. Miyazaki, A survey of the stable marriage problem and its variants, in: Proceedings of the International Conference on Informatics Education and Research for Knowledge-Circulating Society (Icks 2008), ICKS '08, IEEE Computer Society, 2008, pp. 131–136. doi:10.1109/ICKS.2008.7.

[49] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151. doi:10.5381/jot.2008.7.3.a4.

[50] R. Pike, Go at google, in: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, ACM, New York, NY, USA, 2012, pp. 5–6. doi:10.1145/2384716.2384720.

[51] C. A. R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677. doi:10.1145/359576.359585.

[52] M. Wirsing, M. M. Hölzl, N. Koch, P. Mayer (Eds.), Software Engineering for Collective Autonomic Systems - The ASCENS Approach, Vol. 8998 of Lecture Notes in Computer Science, Springer, 2015. doi:10.1007/978-3-319-16310-9.
URL https://doi.org/10.1007/978-3-319-16310-9

[53] T. Bures, F. Plasil, M. Kit, P. Tuma, N. Hoch, Software abstractions for component interaction in the internet of things, IEEE Computer 49 (12) (2016) 50–59. doi:10.1109/MC.2016.377.
URL https://doi.org/10.1109/MC.2016.377

[54] A. Klarl, L. Cichella, R. Hennicker, From helena ensemble specifications to executable code, in: Formal Aspects of Component Software - 11th International Symposium, FACS 2014, 2014, pp. 183–190. doi:10.1007/978-3-319-15317-9_11.
URL https://doi.org/10.1007/978-3-319-15317-9_11

[55] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-oriented software architecture, 4th Edition, Wiley series in software design patterns, Wiley, 2007.

[56] S. Ahuja, N. Carriero, D. Gelernter, Linda and friends, IEEE Computer 19 (8) (1986) 26–34. doi:10.1109/MC.1986.1663305.

[57] P. Wyckoff, S. W. McLaughry, T. J. Lehman, D. A. Ford, T spaces, IBM Systems Journal 37 (3) (1998) 454–474. doi:10.1147/sj.373.0454.
URL https://doi.org/10.1147/sj.373.0454

[58] R. Milner, A Calculus of Communicating Systems, Vol. 92 of Lecture Notes in Computer Science, Springer, 1980. doi:10.1007/3-540-10235-3.

[59] K. V. S. Prasad, A calculus of broadcasting systems, Sci. Comput. Program. 25 (2-3) (1995) 285–327. doi:10.1016/0167-6423(95)00017-8.

[60] L. Cardelli, A. D. Gordon, Mobile ambients, Electr. Notes Theor. Comput. Sci. 10 (1997) 198–201. doi:10.1016/S1571-0661(05)80699-1.

[61] L. Cardelli, A. D. Gordon, Mobile ambients, in: Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, 1998, pp. 140–155. doi:10.1007/BFb0053547.

[62] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, R. Mirandola, Self-adaptive software needs quantitative verification at runtime, Commun. ACM 55 (9) (2012) 69–77. doi:10.1145/2330667.2330686.

[63] J. O. Kephart, D. M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50. doi:10.1109/MC.2003.1160055.

[64] B. Yousefi, F. Ghassemi, R. Khosravi, Modeling and efficient verification of broadcasting actors, in: M. Dastani, M. Sirjani (Eds.), Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers, Vol. 9392 of Lecture Notes in Computer Science, Springer, 2015, pp. 69–83. doi:10.1007/978-3-319-24644-4_5.

[65] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J. Stefani, The FRACTAL component model and its support in java, Softw., Pract. Exper. 36 (11-12) (2006) 1257–1284. doi:10.1002/spe.767.

[66] Y. Abd Alrahman, G. Perelli, N. Piterman, Reconfigurable interaction for MAS modelling, in: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020, 2020, (To Appear).

Appendix A. AbC Operational Semantics

The operational semantics of AbC is based on two relations. The transition relation $\longmapsto$ that describes the behaviour of individual components and the transition relation $\longrightarrow$ that relies on $\longmapsto$ and describes system behaviours.

Appendix A.1. Operational semantics of components

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

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

The λ-labels are used to denote AbC output $\Gamma \triangleright \overline{\Pi}(\tilde{v})$ and input $\Gamma \triangleright \Pi(\tilde{v})$ actions. The former contains the sender's predicate Π, that specifies the expected communication partners, the transmitted values $\tilde{v}$, and the portion of the sender *attribute environment* Γ that can be perceived by receivers. The latter label is just the complementary label selected among all the possible ones that the receiver may accept.

The α-labels include an additional label $\widetilde{\Gamma \triangleright \Pi(\tilde{v})}$ to model the case where a component is not able to receive a message. As it will be seen later, this kind of *negative* labels is crucial to appropriately handle dynamic operators like choice and awareness. In the following we will use $fn(\lambda)$ to denote the set of names occurring in λ.

The transition relation $\longmapsto$ is defined in Table A.2 and Table A.3 inductively on the AbC syntax. For each process operator we have two types of rules: one describing the actions a term can perform, the other one showing how a component discards undesired input messages.

$$\frac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad \{\Pi_1\}_\Gamma = \Pi}{\Gamma :_I (\tilde{E})@\Pi_1.U \xmapsto{\Gamma\downarrow I \triangleright \overline{\Pi}(\tilde{v})} \{\!|\Gamma :_I U|\!\}} \text{ BRD} \qquad \frac{}{\Gamma :_I (\tilde{E})@\Pi.U \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I (\tilde{E})@\Pi.U} \text{ FBRD}$$

$$\frac{\Gamma' \models \{\Pi_1[\tilde{v}/\tilde{x}]\}_{\Gamma_1} \quad \Gamma_1 \downarrow I \models \Pi}{\Gamma_1 :_I \Pi_1(\tilde{x}).U \xmapsto{\Gamma' \triangleright \Pi(\tilde{v})} \{\!|\Gamma_1 :_I U[\tilde{v}/\tilde{x}]|\!\}} \text{ RCV} \qquad \frac{\Gamma' \not\models \{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma \vee \Gamma_1 \downarrow I \not\models \Pi'}{\Gamma_1 :_I \Pi(\tilde{x}).U \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma_1 :_I \Pi(\tilde{x}).U} \text{ FRCV}$$

$$\frac{\Gamma \models \Pi \quad \Gamma :_I P \overset{\lambda}{\mapsto} \Gamma' :_I P'}{\Gamma :_I \langle\Pi\rangle P \overset{\lambda}{\mapsto} \Gamma' :_I P'} \text{ AWARE} \qquad \frac{\Gamma \not\models \Pi}{\Gamma :_I \langle\Pi\rangle P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I \langle\Pi\rangle P} \text{ FAWARE1}$$

$$\frac{\Gamma \models \Pi \quad \Gamma :_I P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P}{\Gamma :_I \langle\Pi\rangle P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I \langle\Pi\rangle P} \text{ FAWARE2}$$

Table A.2: Operational Semantics of Components (Part 1)

The behaviour of an *attribute-based output* is defined by rule BRD in Table A.2. This rule states that when an output is executed, the sequence of expressions $\tilde{E}$ is evaluated, say to $\tilde{v}$, and the *closure* Π of predicate Π_1 under Γ is computed. Hence, these values are sent to other components together with $\Gamma \downarrow I$. This represents the portion of the *attribute environment* that can be perceived by the context and it is obtained from the local Γ by limiting its domain to the attributes in the interface I as defined below:

$$(\Gamma \downarrow I)(a) = \begin{cases} \Gamma(a) & a \in I \\ \bot & \text{otherwise} \end{cases}$$

$$\frac{\Gamma :_I P_1 \xmapsto{\lambda} \Gamma' :_I P_1'}{\Gamma :_I P_1 + P_2 \xmapsto{\lambda} \Gamma' :_I P_1'}\ \textsc{SumL} \qquad \frac{\Gamma :_I P_2 \xmapsto{\lambda} \Gamma' :_I P_2'}{\Gamma :_I P_1 + P_2 \xmapsto{\lambda} \Gamma' :_I P_2'}\ \textsc{SumR}$$

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

$$\frac{\Gamma :_I P_1 \xmapsto{\lambda} \Gamma' :_I P'}{\Gamma :_I P_1 \mid P_2 \xmapsto{\lambda} \Gamma' :_I P' \mid P_2}\ \textsc{IntL} \qquad \frac{\Gamma :_I P_2 \xmapsto{\lambda} \Gamma' :_I P'}{\Gamma :_I P_1 \mid P_2 \xmapsto{\lambda} \Gamma' :_I P_1 \mid P'}\ \textsc{IntR}$$

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

$$\frac{\Gamma :_I P \xmapsto{\lambda} \Gamma' :_I P' \quad K \triangleq P}{\Gamma :_I K \xmapsto{\lambda} \Gamma' :_I P'}\ \textsc{Rec} \qquad \frac{\Gamma :_I P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P \quad K \triangleq P}{\Gamma :_I K \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I K}\ \textsc{FRec}$$

$$\frac{}{\Gamma :_I 0 \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I 0}\ \textsc{FZero}$$

Table A.3: Operational Semantics of Components (Part 2)

Afterwards, possible updates U, following the action, are applied. This is expressed in terms of a recursive function $\{\!|C|\!\}$ defined below:

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

where $\Gamma[a \mapsto v]$ denotes an attribute update such that $\Gamma[a \mapsto v](a') = \Gamma(a')$ if $a \neq a'$ and v otherwise. Rule Brd is not sufficient to fully describe the behaviour of an output action; we need another rule (FBrd) to model the fact that all incoming messages are *discarded* in case only output actions are possible.

Rule Rcv governs the execution of input actions. It states that a message can be received when two *communication constraints* are satisfied: the local attribute environment restricted to interface I ($\Gamma_1 \downarrow I$) satisfies Π, the predicate used by the sender to identify potential receivers; the sender environment Γ' satisfies the receiving predicate $\{\Pi_1[\tilde{v}/\tilde{x}]\}_{\Gamma_1}$. When these two constraints are satisfied the input action is performed and the update U is applied under the substitution $[\tilde{v}/\tilde{x}]$.

Rule FRcv states that an input is *discarded* when the local attribute environment does not satisfy the *sender's predicate*, or the *receiving predicate* is not satisfied by the sender's environment.

The behaviour of a component $\Gamma :_I \langle\Pi\rangle P$ is the same as of $\Gamma :_I P$ only when $\Gamma \models \Pi$, while the component is inactive when $\Gamma \not\models \Pi$. This is rendered by rules Aware, FAware1 and FAware2.

Rules SumL, SumR, and FSum describe behaviour of $\Gamma :_I P_1 + P_2$. Rules SumL and SumR are standard and just say that $\Gamma :_I P_1 + P_2$ behaves nondeterministically either like $\Gamma :_I P_1$ or like $\Gamma :_I P_2$. A message is *discarded* by $\Gamma :_I P_1 + P_2$ if and only if both P_1 and P_2 are not able to receive it. We can observe here that the presence of discarding rules is fundamental to prevent processes that cannot receive messages from evolving without performing actions. Thus *dynamic operators*, that are the ones *disappearing* after a transition like awareness and choice, persist after a message refusal.

The behaviour of the interleaving operator is described by rules IntL, IntR and FInt. The first two are standard process algebraic rules for parallel composition while the discarding rule FInt has a similar interpretation as of rule FSum: a message can be discarded only if both the parallel processes can discard it.

Finally, rules Rec, FRec and FZero are the standard rules for handling process definition and the inactive process. The latter states that process $\mathsf{0}$ always discards messages.

Appendix A.2. Operational semantics of systems

The behaviour of an *AbC* system is described by means of the transition relation $\longrightarrow \subseteq \text{Comp} \times \text{SLAB} \times \text{Comp}$, where Comp denotes the set of components and SLAB is the set of transition labels, λ, generated by the following grammar:

$$\lambda \ ::= \Gamma \triangleright \overline{\Pi}(\tilde{v}) \quad | \quad \Gamma \triangleright \Pi(\tilde{v})$$

The definition of the transition relation $\longrightarrow$ is provided in Table A.4.

$$\frac{\Gamma :_I P \overset{\lambda}{\mapsto} \Gamma' :_I P'}{\Gamma :_I P \xrightarrow{\lambda} \Gamma' :_I P'} \text{ Comp} \qquad \frac{\Gamma :_I P \overset{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}}{\longmapsto} \Gamma :_I P}{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \Pi'(\tilde{v})} \Gamma :_I P} \text{ FComp}$$

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

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

$$\frac{C \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C' \quad f(\Gamma, \tilde{v}) = \Pi'}{[\ C\ ]^{\triangleright f} \xrightarrow{\Gamma \triangleright \overline{\Pi \wedge \Pi'}(\tilde{v})} [\ C'\ ]^{\triangleright f}} \text{ ResO} \qquad \frac{C \xrightarrow{\Gamma \triangleright \Pi \wedge \Pi'(\tilde{v})} C' \quad f(\Gamma, \tilde{v}) = \Pi'}{[\ C\ ]^{\triangleleft f} \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} [\ C'\ ]^{\triangleleft f}} \text{ ResI}$$

$$\frac{C \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C'}{[\ C\ ]^{\triangleright f} \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} [\ C'\ ]^{\triangleright f}} \text{ ResOPass} \qquad \frac{C \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C'}{[\ C\ ]^{\triangleleft f} \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} [\ C'\ ]^{\triangleleft f}} \text{ ResIPass}$$

Table A.4: Operational Semantics of Systems

Rules Comp and FComp depends on relation $\longmapsto$ and they are used to lift the effect of local behaviour to the system level. The former rule states that the relations $\longmapsto$ and $\longrightarrow$ coincide when performing either an input or an output actions, while rule FComp states that a component $\Gamma :_I P$ can discard a message and remain unchanged. However, we would like to stress that the system level label of FComp coincides with that of Comp in case of input actions, which means that externally it cannot be observed whether a message has been accepted or discarded.

Rule Sync states that two parallel components C_1 and C_2 can receive the same message. Rule ComL and its symmetric variant ComR govern communication between two parallel components C_1 and C_2.

Rules ResO and ResI show how *restriction operators* $[\ C\ ]^{\triangleright f}$ and $[\ C\ ]^{\triangleleft f}$ limit *output* and *input* capabilities of C under function f.

Rule ResO states that if C evolves to C' with label $\Gamma \triangleright \overline{\Pi}(\tilde{v})$ and $f(\Gamma, \tilde{v}) = \Pi'$ then $[\ C\ ]^{\triangleright f}$ evolves with label $\Gamma \triangleright \overline{\Pi \wedge \Pi'}(\tilde{v})$ to $[\ C'\ ]^{\triangleright f}$. This means that when C sends messages to all the components satisfying Π, the *restriction operator* limits the interaction to only those that also satisfy Π'.

Rule ResI is similar. However, in this case, the *restriction operator* limits the input capabilities of C. Indeed, $[\ C\ ]^{\triangleleft f}$ will receive the message $\tilde{v}$ and evolve to $[\ C'\ ]^{\triangleleft f}$ with a label $\Gamma \triangleright \Pi(\tilde{v})$ only when $C \xrightarrow{\Gamma \triangleright \Pi \wedge \Pi'(\tilde{v})} C'$ where $f(\Gamma, \tilde{v}) = \Pi'$. Thus, message $\tilde{v}$ is delivered only to those components that satisfy both Π and Π'. Note that, both $[\ C\ ]^{\triangleright f}$ and $[\ C\ ]^{\triangleleft f}$ completely hide input/output capabilities whenever $f(\Gamma, \tilde{v}) \wedge \Pi \doteq \mathsf{ff}$.

Rule ResOPass (resp. ResIPass) states any *input transition* (resp. *output transition*) performed by C is also done by $[\ C\ ]^{\triangleright f}$ (resp. $[\ C\ ]^{\triangleleft f}$).