Science of Computer Programming 202 (2021) 102567

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Provably correct implementation of the *AbC* calculus ☆

Rocco De Nicola [a,b,*], Tan Duong [a,*], Michele Loreti [c,*]

[a] *IMT School for Advanced Studies Lucca, Italy*
[b] *CINI – Cyber Security Laboratory, Rome, Italy*
[c] *University of Camerino, Italy*

ARTICLE INFO

Article history:
Received 21 February 2020
Received in revised form 16 October 2020
Accepted 20 October 2020
Available online 6 November 2020

Keywords:
Process calculi
Formal methods
Distributed computing
Erlang
Correctness proofs

ABSTRACT

Building open, distributed systems while guaranteeing a specific behaviour is difficult because of the dynamicity of the operating environments and the complexity of the interactions of their components. The *AbC* calculus provides a novel communication mechanism to select interacting partners based on their runtime capabilities, making it naturally to model complex interactions and adaptive behaviour in such systems. The formal account of this calculus has enabled constructing formally verifiable models and proving their properties. In this paper, we i) propose an implementation of *AbC* using the *Erlang* language ii) formalize the operational semantics of our implementation; iii) propose a set of rules that given an *AbC* specification, automatically generate *Erlang* executable code; and iv) prove that the proposed translation is correct by establishing a simulation relation between source and target specifications. This enables us to guarantee that any property proved for a given *AbC* specification is preserved by the corresponding implementation.

1. Introduction

Attribute-based communication, originally proposed in [1], is a novel paradigm that allows the dynamic formation of interaction groups while taking into account run-time properties and status of individual members. This paradigm was formalized in the *AbC* kernel calculus [2] to study the impact of attribute-based communication in the realm of so-called collective-adaptive systems (CAS) [3]. In *AbC*, components are equipped with a set of *attributes*, describing their features, whose values may change locally at runtime. The calculus comes with specifically conceived primitives that permit components to communicate on the basis of satisfactions of *predicates* over the attributes that they expose. Communication among components takes place in a broadcast fashion [4] in which only the components that satisfy sender's predicate can receive the sent message; receivers can ignore messages from components that do not satisfy their predicates. By parameterising the interaction predicates with attributes, communication groups can be implicitly changed and adaptation is naturally modelled; run-time changes of attribute values allow additional opportunistic interactions between components.

The expressive power of *AbC* in modelling complex interactions has been demonstrated in [5,6]; there it has been shown that *AbC* can be an alternative to other traditional communication models such as publish/subscribe-based, group-based, broadcast and channel-based interactions. The use of *AbC* to specify and program CAS has also been discussed in other

☆ This work is partially funded by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems).

* Corresponding authors.
E-mail address: tan.duong@imtlucca.it (T. Duong).

https://doi.org/10.1016/j.scico.2020.102567
0167-6423/

papers with illustrative examples, such as classical distributed systems [7,8], communication protocols [9], and real-world systems [2,10]. The paradigm itself has also been employed and studied in other contexts, see e.g., [11–13]. [14].

Despite the promising features of *AbC* in dealing with complex systems, when considering actual CAS programming, implementability, scalability, and proof of correctness of systems specified by resorting to the new communication primitives are still a concern. Researchers have diligently developed implementations for *AbC* in different languages, for example, AbaCus [15] in Java, GoAt [8] in Go, and AErlang [16] in *Erlang* with a common goal of adopting the new communication paradigm for practical use. By design, both AbaCus and GoAt tried to stay aligned with *AbC* whereas AErlang can be seen more as an attribute-based variant of *Erlang* rather than an implementation of *AbC*. These frameworks provide some sort of programming interface that allows writing *AbC*-like programs in their host languages. However, there are still syntax gaps between the supported APIs and *AbC* primitives that make it difficult to use them to directly capture complex forms of *AbC* processes. This means that there may not be a uniform way to express an executable system from an *AbC* specification. Another problem with the existing implementations is that they lack proof of correctness of preserving the *AbC* semantics. These efforts hence fall short of serving as a runtime environment for *AbC*.

The research problem we are after is how to reliably build systems with their behaviour guaranteed correct with respect to *AbC* specifications. We envisage several key ingredients that are needed to tackle this problem. The first is an adequate runtime environment which faithfully respects *AbC* operational semantics. The second is a systematic translation of *AbC* specifications. The third is a rigorous correctness proof of the implementation and of the translation.

In response, we propose an implementation of *AbC* in *Erlang*, called *ABEL* that combines the lessons learned from previous proposals to preserve the semantics of calculus. *ABEL*'s API is designed to closely mimic *AbC*'s syntax, making it easy to write *AbC* specifications. For the actual implementation, we need to encode message broadcasting of *AbC* by relying on *Erlang* point-to-point asynchronous communication.

As pointed out in [17], it is not possible to uniformly encode atomic broadcast using point-to-point communication and thus to model *AbC* atomic broadcast, we follow the approach presented in [18]. This approach relies on building an infrastructure used to forward messages among components that uses a fixed sequencer protocol [19]; messages to be sent are labelled with sequence numbers and components are required to deliver the forwarded messages according to the attached numbers.

To reason about our implementation, we equip also *ABEL* with an operational semantics. In general there are different ways for clearly stating the formal semantics of an implementation of a source language into a host language. The most obvious one would be to rely on the semantics of the host language and to reason on the specified systems by using the tools developed for that language. Another approach would be to introduce an intermediate language capturing the essence of the host language but ignoring its specific syntax.

Different formal semantics of *Erlang* [20], our host language, have been proposed for different purposes, such as model checking [21–23], debugging [24] and reversible computation [25] and it would be certainly possible to adapt one of those for reasoning on the *Erlang* implementation of *AbC*. But, in this case, we would need to examine the entire code of the implementation, which would mean considering also libraries such as OTP [26] which often go beyond the plain language primitives. Thus, we follow the second approach and introduce an intermediate language (*ABEL*), equipped with a formal semantics, capturing the essence of *Erlang*. Correctness of the implementation is proved by showing that the interaction between components and the infrastructure guarantees messages delivery in the desired order.

Instead of writing *ABEL* code manually, we advocate a model-driven approach to the development of *AbC* systems by providing a translation from *AbC* to *ABEL*. A natural question arises to as whether the behaviour of the translated *ABEL* system is the one dictated by the original *AbC* specification. We establish the correspondence between an *AbC* system and an *ABEL* by proving that their respective labelled transition systems are operationally correspondent. Our notion of correspondence is based on simulation in the sense that *ABEL* executions are present in the corresponding *AbC* model. The other direction does not hold because the evolution of *ABEL* components is constrained by sequencing numbers, and thus some executions allowed by an *AbC* term could not be present in the corresponding *ABEL* program. In fact, the mentioned sequencer protocol implicitly introduces a specific scheduling policy for *ABEL* executions whereas *AbC* executions are non-deterministic at each evolution step. Nevertheless, we show that if an *AbC* system can evolve then also the corresponding *ABEL* program does so.

Now, given an *AbC* specification, we can first rely on an external verifier to establish the correctness of the model using verification methods developed for *AbC* [14,9] and then translate the verified specifications into *ABEL* code for actual execution. This approach is particularly effective since correctness of specifications is typically independent of the number of components. Thus, even if we may only verify a system model with a small number of components; its translation in *ABEL* can be executed with a larger system size while still being confident about the correctness of the executable code. With the work in this paper, we bridge the gap between *AbC* specifications and *ABEL* implementations, and make another step toward constructing reliable systems.

This paper extends [27] where *ABEL* was first introduced, and [28] where a translation from *AbC* into *ABEL* was first presented. In a nutshell, the current paper provides the key elements that were missing in earlier attempts, namely guaranteeing correctness of the *ABEL* implementation and of the proposed translation. The new contributions are thus the formal semantics of *ABEL* and the proofs that the infrastructure faithfully models attribute-based communication and that the actual translation is correct. The *ABEL* implementation and the translator have been significantly refined and are available at https://github.com/ArBitral/ABEL, together with several illustrative examples.

Table 1
AbC syntax.

(Components)	$C ::= \Gamma :_{\mathrm{I}} P \mid C \parallel C$
(Processes)	$P ::= Q \mid P \vert P \mid K$
	$Q ::= \Sigma_J \alpha^i_j.U_j \mid \Sigma_J \alpha^o_j.U_j \mid \langle\Pi\rangle Q$
(Input Actions)	$\alpha^i ::= \Pi(\tilde{x})$
(Output Actions)	$\alpha^o ::= (\tilde{E})@\Pi$
(Updates)	$U ::= [a := E]U \mid P$
(Expressions)	$E ::= v \mid x \mid a \mid this.a \mid f(\tilde{E})$
(Predicates)	$\Pi ::= \mathrm{tt} \mid \mathrm{ff} \mid p(\tilde{E}) \mid \Pi \vee \Pi \mid \Pi \wedge \Pi \mid \neg\Pi$

The rest of the paper is organized as follows. Section 2 introduces the fragment of *AbC* calculus of interest. Section 3 contains a description of the *ABEL*'s API for *AbC*, of the translation from *AbC* to the API, and an informal presentation of *ABEL* implementation. The formal semantics of *ABEL* and some of its properties are presented in Section 4. Section 5, instead, is concerned with the correctness of the translation from *AbC* to *ABEL*. Section 6 discusses the related works, while Section 7 concludes the paper and suggests topics for future research.

2. *AbC*: a calculus for attribute-based communication

In this paper we work with a subset of *AbC*; the full calculus is presented in [6]; in fact, this section is mainly a rephrasing of [6] that we report here for the sake of completeness. The considered subset does not allow so-called *mixed choice*, i.e. non-deterministic composition of output and input prefixes and does not permit using the (blocking) awareness predicate to condition the evolution of parallel processes.

2.1. The syntax

Let $\mathcal{A}, \mathcal{V}, \mathcal{X}$ be disjoint countable sets of attribute names, values, and variables, ranged over by a, v and x, respectively. An attribute environment Γ is a partial mapping from $\mathcal{A}$ to $\mathcal{V}$. An interface I, with respect to an attribute environment Γ is a subset of $\mathcal{A}$.

An *AbC* system is a collection of parallel components, each component C is either a process P associated with an environment Γ and an interface I, or the parallel composition of two components. The syntax of *AbC* processes (P, Q) is detailed in Table 1. There, we use $(\tilde{\cdot})$ to denote a finite sequence whose length is not relevant, for example, $\tilde{x}, \tilde{E}$ denotes sequences of variables and expressions, respectively. The basic actions of processes include input and output. Action prefixes exploit run-time attributes and predicates over them to determine the internal behaviour of components and the communication partners. Specifically, $(\tilde{E})@\Pi$ is used to send the results of the evaluation of expressions $\tilde{E}$ to those components whose attributes satisfy predicate Π. $\Pi(\tilde{x})$ is used to receive a message from any component whose attributes (and the message itself) satisfy the predicate Π and bind the received message to $\tilde{x}$.

The process $\Sigma_J \alpha^i_j.U_j$ (respectively $\Sigma_J \alpha^o_j.U_j$) represents a guarded choice where j ranges over some index set J. We use inactive process 0 to denote an empty sum, prefix process $\alpha.U$ to represent the sum with only one element, and $\alpha_1.P_1 + \alpha_2.P_2$ to represent a binary choice between a pair of input or output actions. The symbol $[a := E]$ denotes an atomic update that assigns the result of the evaluation of expression E to attribute a.

A process can also be an awareness process $\langle\Pi\rangle Q$, a parallel process $P|P$ or a process call K (with a unique definition $K \triangleq P$). The symbol $\langle\Pi\rangle$ blocks the following process until Π is satisfied.

A predicate Π can be either tt, ff or be built using logical connectives such as $\wedge, \neg, \ldots$ over atomic predicates $p(\tilde{E})$ with $p \in \mathcal{V}^k$ for some k the length of sequence $\tilde{E}$.

An expression E may be a constant value v, a variable x, an attribute a or an attribute of the local environment $this.a$. The latter can be used in communication predicates (sending or receiving) to distinguish local attributes from the attributes of other participants. Expressions can be built via some generic operator f, for example, binary ones such that $+, -, *, \ldots$.

Both predicates and expressions can take complex forms, of which we deliberately omit the precise syntax; we just refer to them as k-ary operators on subexpressions, i.e., $f_k(E_1, \ldots, E_k)$ and $p_k(E_1, \ldots, E_k)$. For each f_k, we assume the existence of a corresponding function from $\mathcal{V}^k$ to $\mathcal{V}$ describing its semantics. Similarly, for each p_k with its domain $\mathcal{V}^k$, we assume p_k is decidable.

We assume valuations from a set of expressions, with respect to a local attribute environment Γ, to a set of values, denoted by $[\![\cdot]\!]_\Gamma$.

AbC input actions act as binders for free variables. Thus, in $\Pi(\tilde{x}).U$ the occurrences of $\tilde{x}$ in U are bound. A process is closed if it does not contain any free variables. In our model, all processes are closed. Substitutions, denoted by $[\tilde{v}/\tilde{x}]$ are presupposed for attribute updates, expressions and predicates. For example, $\tilde{E}[\tilde{v}/\tilde{x}]$ is the procedure of replacing each occurrence of x in the sequence $\tilde{E}$ by a corresponding v. Substitutions can be extended to process terms in a straightforward way. The application of substitutions is an implicit meta-syntactic operation, i.e., $[\tilde{v}/\tilde{x}]$ instantaneously replaces $\tilde{x}$ with $\tilde{v}$

Table 2
Auxiliary definitions.

(Predicate Satisfaction) $\models, \not\models$	
$\Gamma \models \mathrm{tt}, \Gamma \not\models \mathrm{ff}$ for all Γ	$\Gamma \models \Pi_1 \wedge \Pi_2$ 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 $\Gamma \models \Pi$
$\Gamma \models \mathrm{p_k}(\mathrm{E_1}, \dots, \mathrm{E_k})$ iff $p_k([\![E_1]\!]_\Gamma, \dots, [\![E_k]\!]_\Gamma)$ is true	
(Attribute updates) $\{\!\mid\,\mid\!\}$	**(Environment restriction)** $\downarrow$
$\{\!\mid C \mid\!\} = \begin{cases} \{\!\mid \Gamma[a \mapsto [\![E]\!]_\Gamma]:_I U \mid\!\} \text{ if } C = \Gamma:_I [a := E]U \\ \Gamma:_I P \text{ if } C = \Gamma:_I P \end{cases}$	$(\Gamma_1 \downarrow I)(a) = \begin{cases} \Gamma(a) \text{ if } a \in I \\ \bot \text{ otherwise} \end{cases}$
$\Gamma[a \mapsto v](a') = \begin{cases} \Gamma(a') \text{ if } a \neq a' \\ v \text{ otherwise} \end{cases}$	

Table 3
Operational semantics of *AbC* components.

$\dfrac{[\![\tilde{E}]\!]_\Gamma = \tilde{v} \quad \{\Pi_1\}_\Gamma = \Pi}{\Gamma:_I (\tilde{E})@\Pi_1.U \xmapsto{\Gamma\downarrow I \rhd \overline{\Pi}(\tilde{v})} \{\!\mid \Gamma:_I U \mid\!\}}$ Brd	$\dfrac{}{\Gamma:_I (\tilde{E})@\Pi.U \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I (\tilde{E})@\Pi.U}$ FBrd
$\dfrac{\Gamma' \models \{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma \quad \Gamma \downarrow I \models \Pi'}{\Gamma:_I \Pi(\tilde{x}).U \xmapsto{\Gamma' \rhd \Pi'(\tilde{v})} \{\!\mid \Gamma:_I U[\tilde{v}/\tilde{x}] \mid\!\}}$ Rcv	$\dfrac{\Gamma' \not\models \{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma \vee \Gamma \downarrow I \not\models \Pi'}{\Gamma:_I \Pi(\tilde{x}).U \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I \Pi(\tilde{x}).U}$ FRcv
$\dfrac{\Gamma \models \Pi \quad \Gamma:_I P \xmapsto{\ell} \Gamma':_I P'}{\Gamma:_I \langle\Pi\rangle P \xmapsto{\ell} \Gamma':_I P'}$ Aware	$\dfrac{\Gamma \not\models \Pi \vee \Gamma:_I P \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I P}{\Gamma:_I \langle\Pi\rangle P \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I \langle\Pi\rangle P}$ FAware
$\dfrac{\Gamma:_I P_j \xmapsto{\ell} \Gamma':_I P'_j}{\Gamma:_I \Sigma_J P_j \xmapsto{\ell} \Gamma':_I P'_j}$ Sum$_j$ $(j \in J)$	$\dfrac{\forall j \in J \,.\, \Gamma:_I P_j \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I P_j}{\Gamma:_I \Sigma_J P_j \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I \Sigma_J P_j}$ FSum
$\dfrac{\Gamma:_I P_1 \xmapsto{\ell} \Gamma':_I P'}{\Gamma:_I P_1 \mid P_2 \xmapsto{\ell} \Gamma':_I P' \mid P_2}$ Int	$\dfrac{\forall i \in \{1, 2\} \,.\, \Gamma:_I P_i \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I P_i}{\Gamma:_I P_1 \mid P_2 \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I P_1 \mid P_2}$ FInt
$\dfrac{\Gamma:_I P \xmapsto{\ell} \Gamma':_I P' \quad K \triangleq P}{\Gamma:_I K \xmapsto{\ell} \Gamma':_I P'}$ Rec	$\dfrac{\Gamma:_I P \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I P \quad K \triangleq P}{\Gamma:_I K \xmapsto{\widetilde{\Gamma' \rhd \Pi'(\tilde{v})}} \Gamma:_I K}$ FRec

in the target term. When considering processes and substitutions, we assume that the bound variables in a process are distinct.

2.2. Operational semantics

We use the transition relation $\mapsto \subseteq \mathrm{Comp} \times \mathrm{CLAB} \times \mathrm{Comp}$ to define the local behaviour of a component where Comp denotes the set of components and CLAB is the set of transition labels ζ:

$$\zeta ::= \ell \quad | \quad \widetilde{\Gamma \rhd \Pi(\tilde{v})} \qquad\qquad \ell ::= \Gamma \rhd \overline{\Pi}(\tilde{v}) \quad | \quad \Gamma \rhd \Pi(\tilde{v})$$

The ℓ-labels are used to denote *AbC* output and input actions. The ζ-labels include a discard label to model the case where a component loses a message. More specifically, at the sender side, an output label $\Gamma \rhd \overline{\Pi}(\tilde{v})$ exposes the information about the actual message to be sent, which contains (part of) the component's environment Γ, the communicated values $\tilde{v}$, and the sending predicate Π. Dually, at the receiver side, the label $\Gamma \rhd \Pi(\tilde{v})$, resp. $\widetilde{\Gamma \rhd \Pi(\tilde{v})}$ exposes the information about the incoming message to be accepted, resp. lost.

The transition relation $\mapsto$ is defined in Table 3 inductively on the syntax of *AbC* (Table 1). The semantics rules involve some auxiliary definitions for predicate satisfaction, environment restriction and attribute updates which are given in Table 2. Let us now comment on the component rules. 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.

Table 4
Operational semantics of *AbC* systems.

$$\frac{\Gamma :_I P \overset{\ell}{\mapsto} \Gamma' :_I P'}{\Gamma :_I P \xrightarrow{\ell} \Gamma' :_I P'}\ \textsc{Comp} \qquad \frac{\Gamma :_I P \xmapsto{\widetilde{\Gamma' \triangleright \Pi'(\tilde{v})}} \Gamma :_I P}{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \Pi'(\tilde{v})} \Gamma :_I P}\ \text{F}\textsc{Comp}$$

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

Rule Brd specifies the behaviour of a component with an output prefixing process. The sequence of expressions $\tilde{E}$ is evaluated, say to $\tilde{v}$, and the *closure* Π of predicate Π_1 under Γ is computed. The message to be sent is a triple $(\Gamma \downarrow I, \Pi_1, \tilde{v})$, where $\Gamma \downarrow I$ is the portion of Γ restricted to the interface I (see Table 2). Afterwards, possible attribute updates associated with the process U are applied. Rule (FBrd) states that an output component ignores any incoming messages.

Rule Rcv governs the execution of a component with an input prefixing process upon hearing a message $(\Gamma', \Pi', \tilde{v})$. It states that the message is received when the local attribute environment (Γ) restricted to interface I ($\Gamma \downarrow I$) satisfies the predicate used by the sender (Π'); and the sender environment Γ' satisfies the receiving predicate $\{\Pi[\tilde{v}/\tilde{x}]\}_\Gamma$. Afterwards, possible updates associated with the process U are applied, under the substitution $[\tilde{v}/\tilde{x}]$. Rule FRcv states that a message is discarded when one of the two mentioned constraints fails.

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, FAware. Rules Sum and FSum describe behaviour of a component with a choice process. The behaviour of a component with interleaving processes is described by rules Int, and FInt, where the symmetric version of Int is omitted. Finally, rules Rec, FRec are the standard rules for handling process definition.

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 ℓ defined previously. The definition of the transition relation $\longrightarrow$ is provided in Table 4.

Rules Comp and FComp depend 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, the system level label of FComp implies 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 Com governs communication between two parallel components C_1 and C_2. If C_1 sends a message then C_2 can receive it by applying rule Comp.

Remark 1. When defining the translation, we distinguish between three kinds of predicates depending on the action they control, and use Π_a, Π_s, Π_r to denote awareness, sending, and receiving predicates, respectively. According to the semantics of *AbC*, we can distribute an awareness predicate over the branches of a choice process, i.e., $\langle\Pi_a\rangle\Sigma_{j\in J} P_j = \Sigma_{j\in J}\langle\Pi_a\rangle P_j$ for some index set J. Moreover, since the evaluation of an awareness predicate, the execution of a communication action and attribute updates are performed atomically, we do associate an action with the preceding awareness predicate (if any) and with the following attribute updates (if any). This simplifies the design of our APIs.

Remark 2. In process calculi, the application of substitutions is a metalevel operation that instantaneously replaces $\tilde{x}$ with $\tilde{v}$ in the target term. However, for practical purposes, as mentioned in [29,30], substitutions are recorded explicitly while evaluating process term and serve as an "environment" for bound variables.

Example 1. We illustrate *AbC*'s syntax and semantics by means of a simple example that amount to finding the maximum value in a given list of numbers [31]. We can model each number v in the input list by an *AbC* component of the form $C_i = \Gamma_i :_\emptyset P$ where the attribute environment is specified as $\Gamma_i = [s \mapsto \text{true}, n \mapsto v]$ with s a boolean attribute, initially set to true, expressing the intention of the component to send out its value, and n a numeric attribute set to the actual value of the element that C_i models. The component behaviour is the process P.

$$P \triangleq A \mid B \quad A \triangleq \langle s\rangle(n)@(\text{tt}).0 \quad B \triangleq (x \geq this.n)(x).[s := \text{false}]0$$

Each component announces its value at most once, which may be preempted by the second parallel branch if before sending a larger element is received. The last component whose attribute s remains true is the maximum.[1]

Fig. 1 shows a possible derivation for the list of three values 1, 2, 3 (the active process is underlined). First, C_2 sends an announcement which is accepted by C_1 and discarded by C_3. In this new configuration, C_1 cannot announce its value since

[1] One might wonder how to decide when all components have announced their value. This issue is connected to deciding termination in a distributed setting; investigating this issue goes beyond the scope of the paper. In our *ABEL* implementation, we simply let components report, upon termination, their value to a separate, globally registered *Erlang* process.

$$
\begin{array}{l}
\Gamma_1 :_{\emptyset} (A|B) \ \| \ \Gamma_2 :_{\emptyset} (\underline{A}|B) \ \| \ \Gamma_3 :_{\emptyset} (A|B) \\
\{\mathrm{Com}\} \ \xrightarrow{\Gamma_2 \triangleright \overline{\mathrm{tt}}(2)} \\
\Gamma_1[s \mapsto \mathrm{false}] :_{\emptyset} (A|0) \ \| \ \Gamma_2 :_{\emptyset} (0|B) \ \| \ \Gamma_3 :_{\emptyset} (\underline{A}|B) \\
\{\mathrm{Com}\} \ \xrightarrow{\Gamma_3 \triangleright \overline{\mathrm{tt}}(3)} \\
\Gamma_1[s \mapsto \mathrm{false}] :_{\emptyset} (A|0) \ \| \ \Gamma_2[s \mapsto \mathrm{false}] :_{\emptyset} (0|0) \ \| \ \Gamma_3 :_{\emptyset} (0|B)
\end{array}
$$

Fig. 1. A possible derivation when looking for max element.

the corresponding guard $\langle s \rangle$ is rendered as false. Hence, only C_3 can send its announcement, which is then accepted by C_2 (and discarded by C_1). In the last configuration, C_3 remains the only component whose s is true and therefore deemed to be the maximum element.

3. ABEL – a programming framework for *AbC*

ABEL [27] is a faithful implementation of *AbC* in *Erlang* with the support of APIs in close correspondence with *AbC* primitives. Using *ABEL*, one can write *AbC* specifications in *Erlang* at ease and execute the code afterwards. In the following we introduce *ABEL*'s programming interface, and informally describe its underlying implementation. We also present a simple translation from *AbC* syntax to this interface. The next section will provide the operational details of *ABEL*.

3.1. Programming interface

As in *Erlang*, *ABEL* code is organized into modules. Each module contains a sequence of process definitions for a component type, defined in terms of *Erlang* functions. The functions make use of *ABEL*'s APIs to encode process behaviour whose representations conform to specific grammar rules (see below). Another separate module, "main", contains top-level commands for components and systems initialization.

The syntax for defining process behaviour is given in Table 5 where $\overline{elem}$ denotes a finite sequence of elements *elem*, and font is used to highlight the *ABEL*'s exported library functions. Moreover, *Atom* ranges over *Erlang* atoms, $C, V, \_V$ are *Erlang* variables, and [ ], {} are notations for *Erlang* lists and tuples. Other elements will be explained below.

The main building block of a process definition is function definition def. A definition takes two parameters: a component address C and the current bindings list V of variables. V is initially empty and is gradually updated with the messages received after input actions; in other words, V keeps track of the actual substitutions. The body of a definition (start after $\rightarrow$) contains a single command *com* determining the process behaviour.

A behaviour *beh* is a reference to a previously defined function. It is represented as an *Erlang* anonymous function of one parameter $\_V$ that provides the possibility of passing a new bindings list to the wrapped function. A reference can be passed as a parameter to commands so that the executing process can continue with the referred behaviour. This programming style is reminiscent of continuation-passing style.

Table 5
ABEL's programming interface for defining *AbC* process.

$def ::=$	$Atom(C, V) \rightarrow com.$	Definition
$beh ::=$	$fun(\_V) \rightarrow Atom(C, \_V)\ end$	Reference
	$\mid nil$	
$com ::=$	prefix$(C, V, \{act, beh\})$	Prefix
	$\mid$ choice$(C, V, [\overline{\{act, beh\}}])$	Choice
	$\mid$ parallel$(C, V, [\overline{beh}])$	Parallel
$act ::=$	$\{g, \{\bar{m}\}, s, [\bar{u}]\}$	Output
	$\mid \{g, r, \{\bar{x}\}, [\bar{u}]\}$	Input

A command *com* has parameters C and V bounded by those of an outer function, and a third one specifying basic actions possibly paired with references, depending on the type of command. *ABEL* supports the following commands.

- prefix – takes as parameters an action *act* and a continuation *beh*. The action can be either an input or an output, and its description is a 4-tuple (see Table 5) where g, s, and r denote awareness, sending and receiving predicates, respectively. Moreover, m denotes the message, x denotes input-binding variables and u denotes an attribute update. If g or u are omitted, *ABEL* treats them as *true* and as the empty list [ ], respectively. This command executes *act* and then the behaviour encapsulated in *beh*. The execution of an input action (if successful) returns a message; *ABEL* then continues by calling *beh* on an updated list of bindings, calculated by appending the association of input variables – message contents to the current list V. If *act* is an output action, the continuation is determined by applying *beh* to V.
- choice – takes as a parameter a list of pairs, each describing a prefixing action *act* and a continuation *beh*. This command executes one of the actions and continues with the associated behaviour.

- parallel – takes as a parameter a list of behaviour *beh* and creates new processes, executing functions resulting from the application of *beh* to V.

We now elaborate on the representations of *AbC*'s basic terms and top-level commands. We assume the disjoint sets $\mathbb{A}$, $\mathbb{X}$ and $\mathbb{V}$ in *Erlang* that represent the attribute set $\mathcal{A}$, variable set $\mathcal{X}$ and value set $\mathcal{V}$ in *AbC*. In practice, we use *Erlang* atoms for $\mathbb{A}$, $\mathbb{X}$, and any ground *Erlang* terms, i.e., its elements need no evaluation for $\mathbb{V}$. An attribute environment is then an *Erlang* map, and an interface is an *Erlang* tuple. Their representations, together with other *AbC*'s terms are shown in Table 6.

Table 6
ABEL's representation of *AbC* basic terms.

$a, x, v ::= \mathbb{A}, \mathbb{X}, \mathbb{V}$	Attribute & Variable & Value
$env, l ::= \#\{\overline{a \Rightarrow v}\}, \{\tilde{a}\}$	Environment & Interface
$g, m ::= fun(L) \rightarrow e\ end$	Awareness predicate & Message
$s ::= fun(L, R) \rightarrow e\ end$	Sending predicate
$r ::= fun(L, M, R) \rightarrow e\ end$	Receiving predicate
$u ::= \{a,\ fun(L) \rightarrow e\ end\} \mid \{a,\ fun(L, M) \rightarrow e\ end\}$	Attribute Update
$e ::= \mathsf{var}(x, V) \mid \mathsf{att}(a, L) \mid \mathsf{att}(a, R) \mid \mathsf{msg}(k, M)$	Expression
$\mid v \mid op(e_1, \ldots, e_k) \mid \ldots$	

Awareness predicates g and message elements m are functions parameterized with the local environment (L). A sending predicate s is a function parameterized with the local environment (L) and the environment of other components (R), while receiving predicate r is parameterized also with an incoming message (M). Attribute update u is a pair of an attribute name and the second is a function parameterized with the local environment, and additionally a message in case the update is associated with an input operation.

The body of a function is an expression e. Several library functions are available. $\mathsf{att}(a, L)$ refers to the value of attribute a in an environment L; $\mathsf{msg}(k, M)$ refers to the kth element in a message (tuple) M, and $\mathsf{var}(x, V)$ refers to the value of x in a list V of variables bindings. In addition, values v and generic functions op, either user-defined or built-in can also be used.

While the programming interface provides a means for writing *AbC* specifications in *Erlang* syntax, top-level commands deal with the creations of a messaging infrastructure and components and start their executions.

The infrastructure is responsible for exchanging messages among components and must be built before component creation. A component is created by invoking new_component while passing the attribute environment env and interface l parameters. The command returns a component address C which can be used by start_component to start the execution of C from an initial behaviour beh.

$$C = \mathsf{new\_component}(env, l)$$

$$\mathsf{start\_component}(C, [\,], beh)$$

Example 2. We can write the code of *AbC*'s process P in Example 1 using the *ABEL* programming interface as below (when creating a component, the code assumes data for component C_1). We will explain how to obtain *ABEL* code from *AbC* systematically in the next section.

```
Env = #{s ⇒ true, n ⇒ 1},

I = {}

C = new_component(Env, I),

start_component(C, [ ], fun(_V) → p(C, _V) end).
```

```
p(C, V) →

  A = fun(_V) →  a(C, _V) end,

  B = fun(_V) →  b(C, _V) end,

  parallel(C, V, [A, B]).
```

```
b(C, V) →

  R = {fun(L, M) → msg(1, M) ≥ att(n, L) end},

  X = {x},

  U = [{s, fun(L, M)  →  false end}],

  Act = {R, X, U},

  prefix(C, V, {Act, nil}).
```

```
a(C, V) →

  G = fun(L)  → att(s, L) end,

  M = {fun(L) → att(n, L) end},

  S = fun(L, R) → true end,

  Act = {G, M, S},

  prefix(C, V, {Act, nil}).
```

Here p models the behaviour of *AbC* process P. The correspondence is realized by a parallel command referring to the other two functions. In turn, functions a, b make use of prefix commands with appropriate parameters representing basic terms such as predicates, messages, etc.

3.2. From AbC to ABEL

As the API offers a close syntax to that of *AbC*, we exploit this fact to translate an input *AbC* specification into an *ABEL* program. Generally, we assume an *AbC* specification contains a set of component specifications, each has the form of $(\Gamma, I, K, \mathcal{D})$ with Γ the attribute environment, I the interface, K the initial process name and $\mathcal{D}$ the set of process definitions. Here the definition of K must be included in $\mathcal{D}$.

We define *a family of functions*, generally denoted as **tr** in order to translate different *AbC* terms in an *AbC* specification into *ABEL*.

To translate attribute environments and interfaces, as mentioned, we assume for each $a, x, v \in \mathcal{A}, \mathcal{X}, \mathcal{V}$ in *AbC*, the translation can always find corresponding elements $\mathtt{a}, \mathtt{x}, \mathtt{v} \in \mathbb{A}, \mathbb{X}, \mathbb{V}$ in *Erlang*. Then the translation over attributes, variables, and values is trivial, i.e., $\mathbf{tr}_a(\mathtt{a}) = \mathtt{a}$, $\mathbf{tr}_x(\mathtt{x}) = \mathtt{x}$, $\mathbf{tr}_v(v) = \mathtt{v}$. Translations for attribute environments Γ and interface I are also straightforward, i.e., for some $\Gamma = [a_1 \mapsto v_1, \ldots, a_k \mapsto v_k]$ and $I = \{a_1, \ldots, a_l\}$, we have

$$\mathbf{tr}_\Gamma([a_1 \mapsto v_1, \ldots, a_k \mapsto v_k]) = \#\{\mathbf{tr}_a(a_1) \Rightarrow \mathbf{tr}_v(v_1), \ldots, \mathbf{tr}_a(a_k) \Rightarrow \mathbf{tr}_v(v_k)\}$$

$$\mathbf{tr}_I(\{a_1, \ldots, a_l\}) = \{\mathbf{tr}_a(a_1), \ldots, \mathbf{tr}_a(a_l)\}$$

We next present the translation **tr** for process definitions in $\mathcal{D}$. It is defined as $\mathbf{tr}_\mathcal{D} = \mathbf{tr}_\mathcal{L} \circ \mathbf{tr}_\mathcal{N}$ that consists of two steps: a *normalization step* that refactors process definitions in $\mathcal{D}$ to match the structure provided by the programming interface, and a *generation step* that produces the actual *Erlang* code.

Normalization. Let X be either a process name K or process code P. We define a function $\mathbf{tr}_\mathcal{N}$ that rewrites the definitions, and while doing so may produce auxiliary definitions. A fresh definition is introduced if any of the following conditions hold: i) the continuation of a prefixing process is not a process name; ii) any branch of a parallel process is not a process name.

Table 7 presents rules for normalization procedure. A generic action denoted by α may be paired with awareness and attributes updates (see Remark 1). Please notice that the rules in Table 7 are applied repeatedly until all definitions in $\mathcal{D}$ are considered.

Table 7
Normalizing process definitions.

(prefix)	$\mathbf{tr}_\mathcal{N}(K \triangleq \alpha X)$	=	$K \triangleq \alpha \cdot \mathcal{R}(X)$
(choice)	$\mathbf{tr}_\mathcal{N}(K \triangleq \Sigma_j \alpha_j X_j)$	=	$K \triangleq \Sigma_j \alpha_j \cdot \mathcal{R}(X_j)$
(parallel)	$\mathbf{tr}_\mathcal{N}(K \triangleq \prod_j X_j)$	=	$K \triangleq \prod_j \mathcal{R}(X_j)$
(new def.)	$\mathcal{R}(K)$	=	K
	$\mathcal{R}(P)$	=	K for K fresh and new $\{K \triangleq P\}$

In a prefixing definition, the procedure generates another definition with the same structure, except that the continuation X needs to be processed by a helper function $\mathcal{R}$: if X is a name, $\mathcal{R}$ returns that name, otherwise, if X is a process code P, $\mathcal{R}$ creates a fresh name K, adds a new definition $\{K \triangleq P\}$ and returns K.

In a choice definition, the procedure recursively processes all the branches of the choice. In a parallel definition, the procedure recursively normalizes all the branches.

Code generation. This step produces *Erlang* functions corresponding to a set of normalized *AbC* process definitions. The rules, formalized via function ($\mathbf{tr}_\mathcal{L}$) are reported in Table 8. The first three rules capture all possible forms of a definition and generate the corresponding *def* definitions in *ABEL* style (see Table 5). The next two rules generate behaviour *beh*. The other next two rules deal with *AbC* actions.

The remaining nine rules are responsible for the actual translation of the basic terms of such actions: Namely, $\mathbf{tr}_{\Pi_a}, \mathbf{tr}_{\Pi_s}, \mathbf{tr}_{\Pi_r}$ consider awareness, sending and receiving predicates Π_a, Π_s, Π_r, respectively. $\mathbf{tr}_u$ considers attribute updates $[\tilde{a} := \tilde{E}]$ and $\mathbf{tr}_e$ considers the expressions $\tilde{E}$. The translation is parameterized with input-binding variables $\tilde{x}$ because the expressions contained in receiving predicates or in attribute updates may need them.

The translation functions in the last five rules are very similar to each other but we separate them out for clarity. The translation $\lfloor\!\lfloor \cdot \rfloor\!\rfloor$ deals with local expressions, awareness predicates and is defined as below.

$$\lfloor\!\lfloor a \rfloor\!\rfloor = \mathtt{att(a, L)} \quad \lfloor\!\lfloor \mathit{this.a} \rfloor\!\rfloor = \mathtt{att(a, L)}$$

$$\lfloor\!\lfloor x \rfloor\!\rfloor = \mathtt{var(x, V)} \quad \lceil\!\lceil v \rceil\!\rceil = \mathtt{v}$$

$$\lfloor\!\lfloor f(\tilde{E}) \rfloor\!\rfloor = \mathtt{op}^f(\lfloor\!\lfloor E_1 \rfloor\!\rfloor, \ldots, \lfloor\!\lfloor E_k \rfloor\!\rfloor)$$

Table 8
Code generation.

$\mathbf{tr}_{\mathcal{L}}(K \triangleq \alpha \cdot K')$	$= \mathtt{k(C, V)} \rightarrow \mathtt{prefix(C, V}, \{\mathbf{tr}_{\alpha}(\alpha), \mathbf{tr}_{\mathcal{L}}(\mathtt{K'})\}).$
$\mathbf{tr}_{\mathcal{L}}(K \triangleq \Sigma_{j=1}^{n}\alpha_j \cdot K_j)$	$= \mathtt{k(C, V)} \rightarrow \mathtt{choice(C, V}, [\{\mathbf{tr}_{\alpha}(\alpha_1), \mathbf{tr}_{\mathcal{L}}(\mathtt{K_1})\}, \ldots]).$
$\mathbf{tr}_{\mathcal{L}}(K \triangleq \prod_{j=1}^{m} K_j)$	$= \mathtt{k(C, V)} \rightarrow \mathtt{parallel(C, V}, [\mathbf{tr}_{\mathcal{L}}(\mathtt{K_1}), \ldots, \mathbf{tr}_{\mathcal{L}}(\mathtt{K_m})]).$
$\mathbf{tr}_{\mathcal{L}}(K)$	$= \mathtt{fun(\_V)} \rightarrow \mathtt{k(C, \_V)\ end}$
$\mathbf{tr}_{\mathcal{L}}(0)$	$= \mathtt{nil}$
$\mathbf{tr}_{\alpha}(\langle\Pi_a\rangle(\tilde{E})@(\Pi_s).[\tilde{a} := \tilde{E}])$	$= \{\mathbf{tr}_{\Pi_a}(\Pi_a), \mathbf{tr}_e(\tilde{E}), \mathbf{tr}_{\Pi_s}(\Pi_s), \mathbf{tr}_u([\tilde{a} := \tilde{E}])\}$
$\mathbf{tr}_{\alpha}(\langle\Pi_a\rangle(\Pi_r)(\tilde{x}).[\tilde{a} := \tilde{E}])$	$= \{\mathbf{tr}_{\Pi_a}(\Pi_a), \mathbf{tr}_{\Pi_r}(\Pi_r)^{\tilde{x}}, \mathbf{tr}_x(\tilde{x}), \mathbf{tr}_u([\tilde{a} := \tilde{E}])^{\tilde{x}}\}$
$\mathbf{tr}_e(\tilde{E})$	$= \{\mathbf{tr}_e(E_1)]\!], \ldots, \mathbf{tr}_e(E_k)\}$
$\mathbf{tr}_x(\tilde{x})$	$= \{\mathbf{tr}_x(x_1), \ldots, \mathbf{tr}_x(x_l)\}$
$\mathbf{tr}_u([\tilde{a} := \tilde{E}])$	$= [\{\mathbf{tr}_a(a_1), \mathbf{tr}_e(E_1)\}, \ldots, \{\mathbf{tr}_a(a_i), \mathbf{tr}_e(E_i)\}]$
$\mathbf{tr}_u^{\tilde{x}}([\tilde{a} := \tilde{E}])$	$= [\{\mathbf{tr}_a(a_1), \mathbf{tr}_e^{\tilde{x}}(E_1)\}, \ldots, \{\mathbf{tr}_a(a_i), \mathbf{tr}_e^{\tilde{x}}(E_i)\}]$
$\mathbf{tr}_{\Pi_a}(\Pi_a)$	$= \mathtt{fun(L)} \rightarrow \lfloor\!\lfloor \Pi_a \rfloor\!\rfloor\ \mathtt{end}$
$\mathbf{tr}_{\Pi_s}(\Pi_s]\!]$	$= \mathtt{fun(L, R)} \rightarrow \lceil\!\lceil \Pi_s \rceil\!\rceil\ \mathtt{end}$
$\mathbf{tr}_{\Pi_r}^{\tilde{x}}(\Pi_r)$	$= \mathtt{fun(L, M, R)} \rightarrow \lceil\!\lceil \Pi_r \rceil\!\rceil^{\tilde{x}}\ \mathtt{end}$
$\mathbf{tr}_e(E)$	$= \mathtt{fun(L)} \rightarrow \lfloor\!\lfloor E \rfloor\!\rfloor\ \mathtt{end}$
$\mathbf{tr}_e^{\tilde{x}}(E)$	$= \mathtt{fun(L, M)} \rightarrow \lfloor\!\lfloor E \rfloor\!\rfloor^{\tilde{x}}\ \mathtt{end}$

For complex expressions (or predicates) $f(\tilde{E})$ that do not have a closed form in *AbC* syntax, the translation generates a function call as a place holder for f, and users need to provide a definition in *Erlang* for this function afterward.

Please notice that $\lfloor\!\lfloor\cdot\rfloor\!\rfloor$ treats a and $this.a$ the same way. The translation $\lceil\!\lceil\cdot\rceil\!\rceil$ that deals with sending and receiving predicates instead differentiates between them. In fact we have:

$$\lceil\!\lceil a \rceil\!\rceil \ = \mathtt{att(a, R)} \quad \lceil\!\lceil this.a \rceil\!\rceil = \mathtt{att(a, L)}$$

For all the other cases, the definitions are exactly the same, one needs only to replace $\lfloor\!\lfloor . \rfloor\!\rfloor$ with $\lceil\!\lceil . \rceil\!\rceil$.

The corresponding parameterized versions $\lfloor\!\lfloor\cdot\rfloor\!\rfloor^{\tilde{x}}$ and $\lceil\!\lceil\cdot\rceil\!\rceil^{\tilde{x}}$ of $\lfloor\!\lfloor . \rfloor\!\rfloor$ and $\lceil\!\lceil . \rceil\!\rceil$ differ only when translating some variable y:

$$\lfloor\!\lfloor y \rfloor\!\rfloor^{\tilde{x}} = \lceil\!\lceil y \rceil\!\rceil^{\tilde{x}} \ = \begin{cases} \mathtt{msg(k, M)} \text{ if } y \in \tilde{x} \text{ and } y = x_k \\ \mathtt{var(y, V)} \text{ otherwise} \end{cases}$$

where k is the index of y in the sequence $\tilde{x}$.

Finally, the translation of some initial behaviour K is obvious, i.e., $\mathbf{tr}_{\mathcal{L}}(K)$. The initial behaviour, together with attribute environment and interface are then provided to top-level commands for actual execution.

Example 3. We illustrate the translation **tr** for the process P in Example 1. For brevity, let α_1 stands for $\langle s\rangle(n)@(\mathrm{tt})$ and α_2 for $(x \geq this.n)(x).[s := \mathrm{false}]$. Note that the *AbC* processes in the example are already normalized.

Attribute environment, interface.

$$\begin{aligned}
&\mathbf{tr}_{\Gamma}([s \mapsto true, n \mapsto 1]) && = \mathtt{\#\{s \Rightarrow true, n \Rightarrow 1\}} \\
&\mathbf{tr}_{I}(\emptyset) && = \{\}
\end{aligned}$$

Process definitions, actions.

$$\begin{aligned}
&\mathbf{tr}_{\mathcal{L}}(P \triangleq A|B) && = \mathtt{p(C, V)} \rightarrow \mathtt{parallel(C, V}, [\mathbf{tr}_{\mathcal{L}}(\mathtt{A}), \mathbf{tr}_{\mathcal{L}}(\mathtt{B})]). \\
&\mathbf{tr}_{\mathcal{L}}(A) && = \mathtt{fun(\_V)} \rightarrow \ \mathtt{a(C, \_V)\ end.} \\
&\mathbf{tr}_{\mathcal{L}}(B) && = \mathtt{fun(\_V)} \rightarrow \ \mathtt{b(C, \_V)\ end.} \\
&\mathbf{tr}_{\mathcal{L}}(A \triangleq \alpha_1 \cdot 0) && = \mathtt{a(C, V)} \rightarrow \mathtt{prefix(C, V}, \{\mathbf{tr}_{\alpha}(\alpha_1), \mathtt{nil}\}). \\
&\mathbf{tr}_{\alpha}(\langle s\rangle(n)@(\mathrm{tt})) && = \{\mathbf{tr}_{\Pi_a}(s), \mathbf{tr}_e(n), \mathbf{tr}_{\Pi_s}(\mathrm{tt})\} \\
&\mathbf{tr}_{\Pi_a}(s) && = \mathtt{fun(L)} \ \rightarrow \mathtt{att(s, L)\ end} \\
&\mathbf{tr}_e(n) && = \{\mathtt{fun(L)} \rightarrow \mathtt{att(n, L)\ end}\} \\
&\mathbf{tr}_{\Pi_s}(tt) && = \mathtt{fun(L, R)} \rightarrow \mathtt{true\ end} \\
&\mathbf{tr}_{\mathcal{L}}(B \triangleq \alpha_2 \cdot 0) && = \mathtt{b(C, V)} \rightarrow \mathtt{prefix(C, V}, \{\mathbf{tr}_{\mathcal{L}}(\alpha_2), \mathtt{nil}\}). \\
&\mathbf{tr}_{\alpha}(x \geq this.n)(x).[s := \mathrm{false}]) && = \{\mathbf{tr}_{\Pi_r}^{\tilde{x}}(x \geq this.n), \mathbf{tr}_x(x), \mathbf{tr}_u^{\tilde{x}}([s := \mathrm{false}])\} \\
&\mathbf{tr}_{\Pi_r}^{\tilde{x}}(x \geq this.n) && = \{\mathtt{fun(L, M)} \rightarrow \mathtt{msg(1, M)} \geq \mathtt{att(n, L)\ end}\} \\
&\mathbf{tr}_x(x) && = \{\mathtt{x}\} \\
&\mathbf{tr}_u^{\tilde{x}}([s := \mathrm{false}]) && = [\{\mathtt{s, fun(L, M)} \ \rightarrow \ \mathtt{false\ end}\}]
\end{aligned}$$

3.3. Coordination strategies

In this section, we describe the implementation with a focus on the way *ABEL* coordinates processes and components. The left part of Fig. 2 shows the internal structure of an *ABEL* component. Conceptually, the implementation maps an *AbC* component into a set of *Erlang* processes P and a special process C which coordinates the activities of the different processes. The component coordinator is also connected to the infrastructure for communicating with other components.

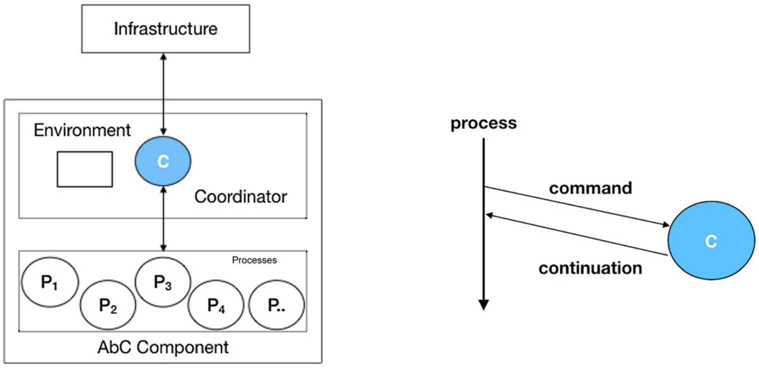

Fig. 2. ABEL component.

The API command new_component (see Section 3.1) automates the creation of C together with the necessary setup, whereas start_component instructs C to create the initial process (that executes the supplied behaviour). After that, new processes may be created by the coordinator, depending on the current behaviour it is handling.

Inside a component, *Erlang* processes execute functions given in form of the programming interface presented in Section 3.1. Such functions rely on APIs which in turn send their actual parameters to the coordinator whose address is C. Each process sends one command (i.e., its parameters) at a time and continues only after receiving an acknowledgement message, as illustrated on the right of Fig. 2. The coordinator decides, on behalf of processes the actual command to be executed. Taking such a decision requires considering different input conditions from the processes and the infrastructure, which the coordinator dynamically records. More specifically, the conditions include the status of component processes, their submitted commands, the current attribute environment, and the messages forwarded from the infrastructure. The coordinator is implemented as a reactive process[2] that combines each received event with the set of previously collected events, in order to select the appropriate (i.e., permitted by *AbC*'s component semantics) action of a component.

Apart from controlling processes inside components, another building block of *ABEL* is the communication infrastructure for preserving a total order of message delivery, obeying *AbC*'s system semantics. This consists of a set of nodes that collaborate on mediating message exchanges. Components join the system via a globally named registration node which assigns them to a node of the infrastructure. A node only communicates with those connected to it; likewise, a component only communicates with the node it is assigned to. The *AbC* semantics was actually formulated on top of broadcast wherein only one output action can take place at a time while input actions do wait concurrently for messages availability. This calls for a restriction on the ordering of message delivery according to a total order [19] which in turn requires appropriate coordination of message exchange in order to guarantee a correct execution semantics.

To guarantee a total order of message delivery for *AbC* components, we exploit an idea proposed in [18]. Whenever a component is willing to send a message, it requests a unique timestamp *id* (i.e., a sequence number) from the infrastructure and labels the message with this *id*. A component delivers a message labelled with an *id* only if it has delivered all messages with $id' < id$. Therefore messages are delivered according to the consecutive messages timestamps.

The infrastructure has been implemented as a set of *Erlang* processes[3] organized logically in a tree-based topology. In particular, each process acts as an inner node while *ABEL* components (coordinators) are connected to the tree as leaves. Following [18], the root node plays the role of a sequencer that allocates fresh timestamps on demand. Whenever a non-root node receives an id request, it forwards the request to its parent. The root issues a counter value and increments the counter. This fresh id is forwarded along the same path of the original request but in reverse order. Eventually, the node which initiated the request receives the fresh id and sends it to the requesting component.

A component sends its message attached with the allocated *id* to its connected tree node. Upon receiving such a message, a tree node forwards the message to its siblings and other connected *AbC* components, except the sender. In this way, any sent message will be eventually forwarded to all components.

4. Formalizing ABEL

The programming interface presented in Section 3.1 allows us to *program* a process as a chain of function calls (to API commands). The execution of each of these functions realises a sequence of interactions with the hosting *ABEL* component that in turn is responsible to dispatch messages through the underlying communication infrastructure. Each component is also responsible for coordinating process executions. This is done by scheduling the activation of pending function calls.

[2] https://erlang.org/doc/man/gen_statem.html.

[3] https://erlang.org/doc/man/gen_server.html.

Table 9
ABEL runtime entities at component level.

(Environment)	$\gamma ::= \epsilon \mid a\ v \mid \gamma \cdot \gamma$
(Substitution)	$\sigma ::= \epsilon \mid x\ v \mid \sigma \cdot \sigma$
(Actions)	$\alpha^i ::= \{g, r, \bar{x}, u\}$
	$\alpha^o ::= \{g, \bar{m}, s, u\}$
(Behaviour)	$b ::= i(\alpha^i, b) \mid o(\alpha^o, b) \mid co(\overline{\alpha^o, b}) \mid ci(\overline{\alpha^i, b}) \mid pa(\bar{b}) \mid nil$
(Process)	$p ::= \{\sigma, b\} \mid [\sigma, b] \mid \langle\rangle_\emptyset$
	$ps ::= \epsilon \mid p \mid p\ ps$

Table 10
ABEL runtime entities at system level.

(Coordinator)	$ctr ::= con(\lambda, c, ms, q) \mid con(\bot, c, ms, q)$
(Component)	$com ::= \langle \gamma; ps; ctr \rangle_\iota$
(Comp. Config.)	$cn ::= com \mid com\ \mathsf{bool} \mid cn\ cn$
(Tree Node)	$t ::= \langle c, ms, q \rangle_\iota \mid \langle \lambda, c, ms, q \rangle_\iota$
(Infrastructure)	$sn ::= t \mid t\ sn$
(System)	$sys ::= \langle\!\langle \mathcal{T} \vdash cn; sn \rangle\!\rangle$

In this section we formalise the behaviour of *ABEL* components. This is defined in terms of their interactions with the infrastructure.

Following previous works on giving operational semantics for systems built upon asynchronous point-to-point communication, e.g., [32–34], we introduce runtime configurations of *ABEL* and specify their semantics in the SOS style [35]. Tables 9 10 present the *ABEL* runtime entities and configurations used in the *ABEL* formal semantics. There, with some abuse, the same notation as for *AbC* is used to represent attributes, values, and input-binding variables.

Attribute environment γ is a list of pairs *attribute-value*. Substitution σ is a list of pairs *variable-value*. Accordingly, $\gamma(a)$ and $\sigma(x)$ denote the values associated with a and x in γ and σ, respectively. The basic actions are α^o for output and α^i for input and the basic terms have the same representation as in the API. Their evaluations however depend also on the substitution σ.

A behaviour b is similar to API commands. However, a distinction between input and output commands is made. In particular, $i(\alpha^i, b)$ ($o(\alpha^o, b)$) indicates an input (output) prefixes, with some action α followed by a behaviour b. $ci(\overline{\alpha^i, b})$ ($co(\overline{\alpha^o, b})$) stands for a choice behaviour among input (output) prefixes. $pa(\bar{b})$ represents a parallel behaviour. Finally, *nil* denotes an inactive behaviour that can be garbage collected.

A process p consists of a behaviour b and a substitution σ. A process can be in three states:

- $\{\sigma, b\}$, an active process with a behaviour b;
- $[\sigma, b]$, a process waiting for an acknowledgement from its coordinator in order to proceed;
- $\langle\rangle_\emptyset$, a terminated process.

Finally, ps denotes a set of processes.

4.1. Component

A (runtime) component *com* contains an attribute environment γ, a set of processes ps, a control ctr and a process identifier ι (in fact, this is the address of the component's coordinator) denoted as $\langle \gamma; ps; con(\lambda, c, ms, q) \rangle_\iota$. Identifiers are used mainly in communication rules and will be omitted when unnecessary. A control has a field λ to hold a timestamp; when the timestamp is used, the field becomes $\bot$. c is a counter representing the number of messages the component has processed. ms is a priority queue while q is a process queue (mailbox). A component configuration cn can be a component or a component followed by a boolean value. The latter is used to enforce explicitly which rules can be applied during component evolution.

In a component, $p\ ps$ denotes the process p is in focus (i.e., active), composed with the rest ps while $[ps]$ is used to mean that all processes in ps are waiting. Moreover, we use _ for any term which is not relevant in the semantics. Similarly, ellipsis ... is used in the control structure *con* to denote the remaining terms, hence the order of terms inside *con* is not relevant. Subscript letters specified next to a given action α (i.e., α_g, α_s, ...) mean the corresponding elements of the action. For queues, $e : q$ denotes a queue with e in its head and the rest is q, whereas $m :: ms$ denotes the insertion of an element in the priority queue ms. $\overset{?}{=}$ is used for checking the form of terms (i.e., a sort of pattern matching), $=$ for constructing a new term and $==$ for checking equality.

The operational semantics of *ABEL* component is defined through the relation $\rightsquigarrow \subseteq AComp \times ALAB \times AComp$ in Tables 11 and 12. The set of labels is the following.

$$ALAB = \{\tau, snd(\rho, (\kappa, \iota)), snd(\rho, (\lambda, m)), snd(\rho, (\lambda, \overline{\mathrm{ff}})), recv(\lambda, m), \widetilde{recv(\lambda, m)}\}$$

Table 11
ABEL component rules (Part 1) where $c' = c + 1$ and $\lambda' = \lambda + 1$.

$$\text{C-out}\ \frac{b \stackrel{?}{=} o(\alpha, \_) \ \ \mathbf{sat}(\alpha_g, \sigma, \gamma) \ \ \lambda == \bot}{\langle \gamma; \{\sigma, b\}\ ps; con(\lambda, q, \ldots)\rangle_\iota \overset{snd(\rho,(\kappa,\iota))}{\rightsquigarrow} \langle \gamma; [\sigma, b]\ ps; con(\kappa, q, \ldots)\rangle_\iota}$$

$$\text{C-in}\ \frac{b \stackrel{?}{=} i(\_, \_)}{\langle \gamma; \{\sigma, b\}\ ps; con\rangle_\iota \overset{\tau}{\rightsquigarrow} \langle \gamma; [\sigma, b]\ ps; con\rangle_\iota} \qquad \text{C-fout}\ \frac{b \stackrel{?}{=} o(\alpha, \_) \ \ (\neg\mathbf{sat}(\alpha_g, \sigma, \gamma) \vee \lambda \neq c)}{\begin{array}{c}\langle \gamma; \{\sigma, b\}\ ps; con(\lambda, c, \ldots)\rangle_\iota \overset{\tau}{\rightsquigarrow} \\ \langle \gamma; [\sigma, b]\ ps; con(\lambda, c, \ldots)\rangle_\iota\end{array}}$$

$$\text{C-snd}\ \frac{\begin{array}{c} b \stackrel{?}{=} o(\alpha, b') \ \ \lambda == c \ \ \mathbf{sat}(\alpha_g, \sigma, \gamma) \\ v = \mathbf{eval}(\alpha_e, \sigma, \gamma) \ \ \pi = \mathbf{eval}(\alpha_s, \sigma, \gamma) \ \ m = (\gamma, \pi, v) \\ \gamma' = \mathbf{upd}(\alpha_u, \sigma, \gamma) \end{array}}{\langle \gamma; \{\sigma, b\}\ ps; con(\lambda, c, \ldots)\rangle_\iota \overset{snd(\rho,(\lambda,m))}{\rightsquigarrow} \langle \gamma'; \{\sigma, b'\}\ ps; con(\bot, c', \ldots)\rangle_\iota \ \mathbf{stch}(\gamma, \gamma')}$$

$$\text{C-fsnd}\ \frac{b \stackrel{?}{=} o(\alpha, b') \ \ \lambda == c \ \ \neg\mathbf{sat}(\alpha_g, \sigma, \gamma)}{\langle \gamma; \{\sigma, b\}\ [ps]; con(\lambda, c, \ldots)\rangle_\iota \overset{snd(\rho,(\lambda,\overline{\mathrm{ff}}))}{\rightsquigarrow} \langle \gamma;\ [\sigma, b]\ [ps]; con(\bot, c', \ldots)\rangle_\iota}$$

$$\text{C-recv}\ \frac{\begin{array}{c} ms \stackrel{?}{=} (\lambda, m) : ms' \ \ \lambda == c \ \ m \stackrel{?}{=} (\gamma_1, \pi_1, v) \\ p \stackrel{?}{=} \{\sigma, i(\alpha, b)\}\ .\ \mathbf{sat}(\alpha_g, \sigma, \gamma) \wedge \mathbf{match}(\alpha_r, \sigma, \gamma, m) \\ \sigma' = \sigma \cdot \alpha_x\, v \ \ \gamma' = \mathbf{upd}(\alpha_u, \sigma', \gamma) \end{array}}{\begin{array}{c}\langle \gamma; [p\ ps]; con(c, ms, \ldots)\rangle_\iota \overset{recv(\lambda,m)}{\rightsquigarrow} \\ \langle \gamma'; \{\sigma', b\}\ [ps]; con(c', ms', \ldots)\rangle_\iota \ (\mathbf{stch}(\gamma, \gamma') \vee \mathbf{stch}(c', \lambda))\end{array}}$$

$$\text{C-frecv}\ \frac{\begin{array}{c} ms \stackrel{?}{=} (\lambda, m) : ms' \ \ \lambda == c \ \ m \stackrel{?}{=} (\gamma_1, \pi_1, v) \\ \forall p \stackrel{?}{=} \{\sigma, i(\alpha, \_)\} \in ps\ .\ \neg\mathbf{sat}(\alpha_g, \sigma, \gamma) \vee \neg\mathbf{match}(\alpha_r, \sigma, \gamma, m) \end{array}}{\langle \gamma; [ps]; con(\lambda, c, ms, \ldots)\rangle_\iota \overset{\widetilde{recv(\lambda,m)}}{\rightsquigarrow} \langle \gamma; [ps]; con(\lambda, c', ms', \ldots)\rangle_\iota \ \mathbf{stch}(c', \lambda)}$$

$$\text{C-res}\ \frac{q \stackrel{?}{=} (\lambda, \epsilon) : q'}{\begin{array}{c}\langle \gamma; ps; con(\kappa, c, q, \ldots)\rangle_\iota \overset{\tau}{\rightsquigarrow} \\ \langle \gamma; ps; con(\lambda, c, q', \ldots)\rangle_\iota \ \mathbf{stch}(c, \lambda)\end{array}} \qquad \text{C-msg}\ \frac{q \stackrel{?}{=} (\lambda, m) : q' \ \ ms' = (\lambda, m) :: ms}{\begin{array}{c}\langle \gamma; ps; con(q, ms, \ldots)\rangle_\iota \overset{\tau}{\rightsquigarrow} \\ \langle \gamma; ps; con(q', ms', \ldots)\rangle_\iota\end{array}}$$

$$\text{C-stch}\ \frac{ps' = \mathbf{unblk}(ps)}{\langle \gamma; ps; con\rangle_\iota \ \text{true} \overset{\tau}{\rightsquigarrow} \ \langle \gamma; ps'; con\rangle_\iota} \qquad \text{C-fstch}\ \frac{}{\langle \gamma; ps; con\rangle_\iota \ \text{false} \overset{\tau}{\rightsquigarrow} \ \langle \gamma; ps; con\rangle_\iota}$$

The first label represents tau actions, the labels start with *snd* denote sending actions to some connected process whose identifier is ρ, whose value will become known at the system level. In order, $snd(\rho, (\kappa, \iota))$ means that the component ι sends a request for a timestamp, $snd(\rho, (\lambda, m))$ means sending a data message. Similarly, the labels starting with *recv* denote receiving actions. We distinguish between $recv(\lambda, m)$, denoting that a message has been successfully received, and $\widetilde{recv(\lambda, m)}$, denoting that a message has been discharged. We will also use special messages, denoted as $(\lambda, \overline{\mathrm{ff}})$ for some timestamp λ, to which all components discard it. The corresponding label for this message is $snd(\rho, (\lambda, \overline{\mathrm{ff}}))$.

In the semantics, we use the functions **sat, eval, upd, match** to implement the auxiliary functions presented in Table 2 and those used by *AbC* component semantics. Notice that since we keep track of variable bindings explicitly, σ is used for retrieving values of input-variables x when needed, i.e., $\sigma(x)$.

- $\mathbf{eval}(g, \sigma, \gamma)$ evaluates the awareness predicate g with attribute environment γ, using σ for determining the values of the bound variables used by g.
- $\mathbf{eval}(s, \sigma, \gamma)$ partially evaluates the sending predicate s with attribute environment γ, using σ for determining the values of the bound variables used by s. By partial evaluation we mean that the result of this evaluation is not completely defined because the predicate s is parameterized also with the environment of a receiving component. We write π_s to denote the result of partial evaluation of s.
- $\mathbf{upd}(\gamma, \sigma, u)$ updates the attribute environment γ according to update u using σ for determining the values of the bound variables used in u.
- $\mathbf{match}(r, \sigma, \gamma, m)$ verifies the communication constraints for accepting the message $m = (\gamma_s, \pi_s, v)$ at a receiver with receiving predicate r and attribute environment γ. The function checks if the sender predicate π_s is satisfied in the receiver's environment γ, and if the receiving predicate is satisfied in the sender environment γ_s and communicated values v.

We now provide some comments about the rules in Table 11. For any output action α with an associated awareness predicate α_g that is satisfied in the current runtime environment, rule C-out requests for a new timestamp if this field λ is not available ($\bot$). The configuration evolves to one with κ in place of the waiting timestamp, and the executing process is suspended. κ is a number different from $\bot$ which essentially prevents the same rule being triggered by other sending processes.

Any input command makes the executing process blocked (Rule C-in). Similarly, any output process that does not meet the condition of the associated awareness predicate or in that sending attempt, the component is not in a sending state ($\lambda \neq c$) goes to a blocking state (Rule C-fout).

When the obtained timestamp λ is equal to the local counter c, a component must send out a message, either an actual data message (handled by Rule C-snd) or the mentioned special message (handled by Rule C-fsnd).

Table 12
ABEL component rules (Part 2) where $c' = c + 1$ and $\lambda' = \lambda + 1$.

$$\text{C-par}\;\frac{b \overset{?}{=} pa(\overline{b'})}{\begin{array}{c}\langle \gamma; \{\sigma, b\}\ ps; con\rangle_\iota \overset{\tau}{\rightsquigarrow}\\ \langle \gamma; \{\sigma, b_1\}\ \dots\ \{\sigma, b_k\}\ ps; con\rangle_\iota\end{array}} \qquad \text{C-cout}\;\frac{b \overset{?}{=} co(\overline{\alpha, \_})\ \ (\exists \alpha' \in \bar{\alpha}\ .\ \mathbf{sat}(\alpha'_g, \sigma, \gamma))\ \ \lambda == \bot}{\begin{array}{c}\langle \gamma; \{\sigma, b\}\ ps; con(\lambda, q, \dots)\rangle_\iota \overset{snd(\rho,(\kappa,\iota))}{\rightsquigarrow}\\ \langle \gamma; [\sigma, b]\ ps; con(\kappa, q, \dots)\rangle_\iota\end{array}}$$

$$\text{C-cin}\;\frac{b \overset{?}{=} ci(\_, \_)}{\begin{array}{c}\langle \gamma; \{\sigma, b\}\ ps; con\rangle_\iota \overset{\tau}{\rightsquigarrow}\\ \langle \gamma; [\sigma, b]\ ps; con\rangle_\iota\end{array}} \qquad \text{C-fcout}\;\frac{b \overset{?}{=} co(\overline{\alpha, \_})\ \ (\forall \alpha' \in \bar{\alpha}\ .\ \neg\mathbf{sat}(\alpha'_g, \sigma, \gamma)) \vee \lambda \neq c}{\begin{array}{c}\langle \gamma; \{\sigma, b\}\ ps; con(\lambda, c, \dots)\rangle_\iota \overset{\tau}{\rightsquigarrow}\\ \langle \gamma; [\sigma, b]\ ps; con(\lambda, c, \dots)\rangle_\iota\end{array}}$$

$$\text{C-csnd}_j\;\frac{\begin{array}{c}b_1 \overset{?}{=} co(\overline{\alpha, b})\ \ (\alpha_j, b_j) \in \overline{(\alpha, b)}\ \ \mathbf{sat}(\alpha_{j_g}, \sigma, \gamma)\ \ \lambda == c\\ v = \mathbf{eval}(\alpha_{j_e}, \sigma, \gamma)\ \ \pi = \mathbf{eval}(\alpha_{j_s}, \sigma, \gamma)\ \ m = (\gamma, \pi, v)\\ \gamma' = \mathbf{upd}(\alpha_{j_u}, \sigma, \gamma)\end{array}}{\langle \gamma; \{\sigma, b_1\}\ ps; con(\lambda, c, \dots)\rangle_\iota \overset{snd(\rho,(\lambda,m))}{\rightsquigarrow} \langle \gamma'; \{\sigma, b_j\}\ ps; con(\bot, c', \dots)\rangle_\iota\ \mathbf{stch}(\gamma, \gamma')}$$

$$\text{C-fcsnd}\;\frac{b \overset{?}{=} co(\overline{\alpha, \_})\ \ (\forall \alpha' \in \bar{\alpha}\ .\ \neg\mathbf{sat}(\alpha'_g, \sigma, \gamma))\ \ \lambda == c}{\langle \gamma; \{\sigma, b\}\ [ps]; con(\lambda, c, \dots)\rangle_\iota \overset{snd(\rho,(\lambda,\bar{\mathrm{ff}}))}{\rightsquigarrow} \langle \gamma; [\sigma, b]\ [ps]; con(\bot, c', \dots)\rangle_\iota}$$

$$\text{C-crecv}_j\;\frac{\begin{array}{c}ms \overset{?}{=} (\lambda, m) : ms'\ \ m \overset{?}{=} (\gamma_1, \pi_1, v)\ \ \lambda == c\\ b_1 \overset{?}{=} ci(\overline{\alpha, b})\ (\alpha_j, b_j) \in \overline{(\alpha, b)}\ \ \mathbf{sat}(\alpha_{j_g}, \sigma, \gamma)\ \ \mathbf{match}(\alpha_{j_r}, \sigma, \gamma, m)\\ \sigma' = \sigma \cdot \alpha_{j_x}\ v\ \ \gamma' = \mathbf{upd}(\alpha_{j_u}\sigma', \gamma)\end{array}}{\begin{array}{c}\langle \gamma; [\{\sigma, b_1\}\ ps]; con(c, ms, \dots)\rangle_\iota \overset{recv(\lambda,m)}{\rightsquigarrow}\\ \langle \gamma'; \{\sigma', b_j\}\ [ps]; con(c', ms', \dots)\rangle_\iota\ (\mathbf{stch}(\gamma, \gamma') \vee \mathbf{stch}(\lambda, c'))\end{array}}$$

$$\text{C-fcrecv}\;\frac{\begin{array}{c}ms \overset{?}{=} (\lambda, m) : ms'\ \ m \overset{?}{=} (\gamma_1, \pi_1, v)\ \ \lambda == c\\ (\forall p \overset{?}{=} \{\sigma, ci(\overline{\alpha, \_})\} \in ps\ .\forall \alpha' \in \bar{\alpha}\ .\ \neg\mathbf{sat}(\alpha'_g, \sigma, \gamma) \vee \neg\mathbf{match}(\alpha'_r, \sigma, \gamma, m))\end{array}}{\langle \gamma; [ps]; con(\lambda, c, ms, \dots)\rangle_\iota \overset{\widetilde{recv(\lambda,m)}}{\rightsquigarrow} \langle \gamma; [ps]; con(\lambda, c', ms', \dots)\rangle_\iota\ \mathbf{stch}(c', \lambda)}$$

Rule C-snd sends a message if there exists an output prefixing process whose awareness predicate α_g is satisfied in the environment γ. A message (m) includes attribute environment (γ), the partial evaluation of the sending predicate (π) and the evaluation of the output expression (v). The component configuration evolves into a new one where γ is possibly updated into γ', the sending process evolves with its continuation, the local counter c is increased by 1 and the corresponding field for message timestamp in the coordinator becomes unavailable $\bot$.

Since a change in the attribute environment may activate sending processes that were previously blocked by their awareness predicates, the new configuration is guarded by a *state change* detection, expressed by the function **stch**. This function compares the old attribute environment γ and a new one γ', that returns true if $\gamma \neq \gamma'$ and false otherwise. The future evolution of this new configuration is determined by the rules C-stch and C-fstch. Rule C-stch unblocks all sending processes, expressed by the function **unblock** while rule C-fstch leaves the component unchanged.

Rule C-fsnd is used when a component is in a sending turn, but no output action is possible because their awareness predicates are not satisfied. In this case, an empty message is sent. This message shall be discarded by all other components but it is necessary to avoid deadlock at the system level.

Rule C-res is triggered when a timestamp λ arrives at the input queue q. The received value λ replaces κ. The evolved configuration is also guarded by a check on state change, e.g., whether the obtained λ matches the local counter c. If this is the case, the component enters a sending state where all blocked sending processes are retried. The evolution after C-res is passed to rules C-stch, C-fstch as described above.

Rule C-msg moves a data message from the top of the input queue to the priority queue ms.

A component evaluates a message m on the top of the priority queue ms for receiving when all processes are blocked, and the associated timestamp of m matches the local counter c. There are two cases, namely successfully receive m (rule C-recv) or discard m (rule C-frecv).

Rule C-recv handles reception of an external message m at the top of the priority queue ms when its attached timestamp is equal to the local counter c and when all processes are blocked. The rule checks if there exists an input prefixing process that can receive m. This includes checking the satisfaction of awareness predicate of the input action α_g in the environment γ and the satisfaction of the communication constrains induced by m and the receiving predicate, expressed by the function **match**. If such an inputting process is found, it is unblocked to evolve according to its continuation behaviour b with an updated substitution σ'. The configuration evolves with local counter c increased by 1 and with a possibly updated attribute environment γ'. A change in the attribute environment may unblock sending processes (as in rule C-snd), thus we guard the configuration with the function $\mathbf{stch}(\gamma, \gamma')$. Moreover, if the new value of the counter c' is equal to the component timestamp λ then the component must send a message. Therefore, we guard the configuration also with the function $\mathbf{stch}(c', \lambda)$, which returns true if λ is equal to c' and false otherwise.

Rule C-frecv, on the other hand, discards a message m when there is no receiving process that can accept it. In the evolved configuration, the local counter is increased, the message is removed from the input queue, and the possibility for the component to send the message is checked by appending **stch** to the configuration.

Table 13
Tree node semantics.

$$\text{T-req}\ \frac{q \overset{?}{=} (\kappa, r) : q' \;\; m = (\kappa, r \cdot \iota)}{\langle c, ms, q\rangle_\iota \xrightarrow{snd(\rho,m)} \langle c, ms, q'\rangle_\iota} \qquad \text{T-new}\ \frac{q \overset{?}{=} (\kappa, r) : q' \;\; r \overset{?}{=} \iota' \cdot r' \;\; m = (\lambda, r')}{\langle \lambda, c, ms, q\rangle_\iota \xrightarrow{snd(\iota',m)} \langle \lambda', c, ms, q'\rangle_\iota}$$

$$\text{T-res}\ \frac{q \overset{?}{=} (\lambda, r) : q' \;\; r \overset{?}{=} \iota' \cdot r' \;\; m = (\lambda, r')}{\langle c, ms, q\rangle_\iota \xrightarrow{snd(\iota',m)} \langle c, ms, q'\rangle_\iota} \qquad \text{T-msg}\ \frac{q \overset{?}{=} (\lambda, m) : q' \;\; ms' = (\lambda, m) :: ms}{\langle q, ms, \ldots\rangle_\iota \xrightarrow{\tau} \langle q', ms', \ldots\rangle_\iota}$$

$$\text{T-fwd}\ \frac{ms \overset{?}{=} (\lambda, m) : ms' \;\; \lambda == c}{\langle c, ms, \ldots\rangle_\iota \xrightarrow{snd(\mu,(\lambda,m))} \langle c+1, ms', \ldots\rangle_\iota}$$

Other rules in Table 12 deal with other types of process behaviour. In particular, rule C-par creates new processes from a list of behaviour $\overline{b'}$ with the substitutions σ inherited from the parent process as their initial bindings lists. Rules C-cin, C-cout, C-csend, C-crecv and their negative versions handle choice operators among inputs and outputs. Among these, C-csend and C-crecv are respectively responsible for the actual sending and receiving data messages. Each rule has several instances, subscripted by an index j to deal with the specific branches of the choice process under consideration.

4.2. Tree nodes

As shown in Table 10, a tree node t is a tuple $\langle c, ms, q\rangle_\iota$ containing a local counter c, a message queue q, a priority queue ms that sorts messages according to their attached timestamps, and a unique identifier (i.e., address) ι. The root of the tree additionally has a sequence number λ that keeps track of the number of timestamps it has allocated, i.e., $\langle \lambda, c, ms, q\rangle_\iota$.

At any point in the system execution, there are two types of messages: timestamp messages and data messages. Both are originated from the sending components and are routed by the nodes of the tree to the appropriate destinations. Data messages are routed in a flooding fashion, effectively modelling broadcast.

The transition labels are of the form $\{snd(\rho, m), snd(\mu, m), snd(\iota, m), \tau\}$ where the first two labels denote the actions of forwarding a given message to the parent and the set of connected nodes, respectively. The third label denotes sending a message to another tree node whose address is ι, the last label denotes an internal action.

The semantics of a tree node is specified in Table 13. T-Req handles requests for a new timestamp by adding the current node identifier to the address list r contained in the message, and then forward the pair to a parent node, indicated by ρ. Note that ρ will be resolved at the system level where the current node is structured into a tree topology. The root node deals with a timestamp request by forwarding its sequence number to an immediate node whose identifier extracted from the address list r, as stated in rule T-new. Other non-root nodes forward the sequence number back to the original sender in the same manner by applying the rule T-res.

The last two rules in Table 13 deal with data messages. Any data message forwarded is first buffered into the priority queue ms (Rule T-msg). A top message in ms is delivered to a set of connected nodes, denoted by μ only if its attached timestamp matches the counter c (Rule T-fwd). Similar to ρ, μ will be resolved at the system level.

4.3. System

Let $\mathcal{C}$ and $\mathcal{S}$ be sets of components and tree nodes, respectively. Moreover, let us denote by $\mathcal{C}^{\mathcal{A}}$ and $\mathcal{S}^{\mathcal{A}}$ the sets of addresses of the components and tree nodes in $\mathcal{C}$ and $\mathcal{S}$, respectively. A tree topology $\mathcal{T}$ is a mapping from child to parent, except that the root is also its own parent. The set of children of a given node $s \in dom(\mathcal{T})$ (i.e., the domain of $\mathcal{T}$) can be defined as $childs(\mathcal{T}, s) = \{s' \neq s \mid \mathcal{T}(s') = s\ \forall s' \in dom(\mathcal{T})\}$. The set of connected nodes of a node s is thus $\textbf{connected}(\mathcal{T}, s) = childs(\mathcal{T}, s) \cup \{\mathcal{T}(s)\}$. An *ABEL* system, denoted by $\langle\!\langle \mathcal{T} \vdash \mathcal{C}; \mathcal{S} \rangle\!\rangle$, is a tuple containing $\mathcal{C}$ and $\mathcal{S}$, structured into a topology $\mathcal{T}$ satisfying the following constraints.

- (all nodes are connected) $dom(\mathcal{T}) = \mathcal{C}^{\mathcal{A}} \cup \mathcal{S}^{\mathcal{A}} \wedge \forall \iota \in \mathcal{C}^{\mathcal{A}} \cup \mathcal{S}^{\mathcal{A}}.\mathcal{T}(\iota) \in \mathcal{S}^{\mathcal{A}}$
- (all components are leaves) $\forall \iota \in \mathcal{C}^{\mathcal{A}}, \exists! \iota' \in \mathcal{S}^{\mathcal{A}}.\mathcal{T}(\iota) = \iota' \ \wedge\ \forall \iota' \in \mathcal{S}^{\mathcal{A}}.childs(\iota') \neq \emptyset$
- (there is only one root) $\exists! \iota \in \mathcal{S}^{\mathcal{A}}.\mathcal{T}(\iota) = \iota$
- (there is no cycle) $\forall \iota \in \mathcal{C}^{\mathcal{A}} \cup \mathcal{S}^{\mathcal{A}}.\iota \notin des(\mathcal{T}, \iota)$ where $des(\mathcal{T}, s) = \{s'' \mid \exists s' \in childs(\mathcal{T}, s).s'' \in des(\mathcal{T}, s')\} \cup childs(\mathcal{T}, s)$

The operational semantics of *ABEL* system is defined through the relation $\rightsquigarrow \subseteq ASYS \times TLAB \times ASYS$ with the set of labels contains the following.

$$TLAB = \{\tau, snd(\lambda, m), recv(\lambda, m), \widetilde{recv(\lambda, m)}\}$$

The system rules are presented in Table 14. Rules C-Tau and T-Tau lift τ moves at components and tree nodes to the system level. Furthermore, the facts that a component sending and discarding special messages are also hidden at the system level, as stated by rules S-Tau, F-Tau.

Rule Multicast stipulates that a data message is forwarded from a tree node to all connected nodes and components. There, we enforce in the rule that the sent message is put immediately into mailboxes of appropriate destinations. We could, alternatively, add an "ether" (or global mailbox) to represent messages in transit and another rule for non-deterministic

Table 14
ABEL system semantics.

$$\text{Send1}\ \frac{n_\iota \overset{snd(\rho,m_1)}{\rightsquigarrow} n'_\iota \quad \iota' = \mathcal{T}(\iota) \quad m_1 \overset{?}{=} (\lambda, m)}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \langle q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle \overset{snd(\lambda,m)}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \langle m_1 : q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle}
\qquad
\text{Send2}\ \frac{n_\iota \overset{snd(\rho,m)}{\rightsquigarrow} n'_\iota \quad \iota' = \mathcal{T}(\iota) \quad m \overset{?}{=} (\kappa, \_)}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \langle q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \langle m : q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle}$$

$$\text{Send3}\ \frac{t_\iota \overset{snd(\iota',m)}{\rightsquigarrow} t'_\iota \quad \iota' \in dom(\mathcal{T})}{\langle\!\langle \mathcal{T} \vdash \mathcal{C}; t_\iota\ \langle q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash \mathcal{C}; t'_\iota\ \langle m : q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle}
\qquad
\text{Send4}\ \frac{t_\iota \overset{snd(\iota',m)}{\rightsquigarrow} t'_\iota \quad \iota' \in dom(\mathcal{T})}{\langle\!\langle \mathcal{T} \vdash \langle q, \ldots\rangle_{\iota'}\ \mathcal{C}; t_\iota\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash \langle m : q, \ldots\rangle_{\iota'}\ \mathcal{C}; t'_\iota\ \mathcal{S}\rangle\!\rangle}$$

$$\text{Recv}\ \frac{n_\iota \overset{recv(\lambda,m)}{\rightsquigarrow} n'_\iota \quad m \overset{?}{=} (\gamma, \pi, \nu)}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle \overset{recv(\lambda,m)}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle}
\qquad
\text{FRecv}\ \frac{n_\iota \overset{\widetilde{recv(\lambda,m)}}{\rightsquigarrow} n'_\iota \quad m \overset{?}{=} (\gamma, \pi, \nu)}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle \overset{\widetilde{recv(\lambda,m)}}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle}$$

$$\text{Multicast}\ \frac{t_\iota \xrightarrow{snd(\mu,(\lambda,m))} t'_\iota \quad \bar{\iota} = \mathbf{connected}(\mathcal{T}, \iota)\backslash\{\iota\}}{\langle\!\langle \mathcal{T} \vdash \bigcup_{\iota_i \in \bar{\iota}} \langle q, \ldots\rangle_{\iota_i}\ \mathcal{C}; t_\iota \bigcup_{\iota_j \in \bar{\iota}} \langle q, \ldots\rangle_{\iota_j}\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash \bigcup_{\iota_i \in \bar{\iota}} \langle (\lambda, m) : q, \ldots\rangle_{\iota_i}\ \mathcal{C}; t'_\iota \bigcup_{\iota_j \in \bar{\iota}} \langle (\lambda, m) : q, \ldots\rangle_{\iota_j}\ \mathcal{S}\rangle\!\rangle}$$

$$\text{C-Tau}\ \frac{n_\iota \overset{\tau}{\rightsquigarrow} n'_\iota}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle}
\qquad
\text{T-Tau}\ \frac{t_\iota \xrightarrow{\tau} t'_\iota}{\langle\!\langle \mathcal{T} \vdash \mathcal{C}; t_\iota\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash \mathcal{C}; t'_\iota\ \mathcal{S}\rangle\!\rangle}$$

$$\text{S-Tau}\ \frac{n_\iota \overset{snd(\rho,m)}{\rightsquigarrow} n'_\iota \quad \iota' = \mathcal{T}(\iota) \quad m \overset{?}{=} (\lambda, (\_, \text{ff}, ()))}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \langle q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \langle m : q, \ldots\rangle_{\iota'}\ \mathcal{S}\rangle\!\rangle}
\qquad
\text{F-Tau}\ \frac{n_\iota \overset{\widetilde{recv(\lambda,m)}}{\rightsquigarrow} n'_\iota \quad m \overset{?}{=} (\_, \text{ff}, ())}{\langle\!\langle \mathcal{T} \vdash n_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle \overset{\tau}{\leadsto\!\!\!\!\Rightarrow} \langle\!\langle \mathcal{T} \vdash n'_\iota\ \mathcal{C};\ \mathcal{S}\rangle\!\rangle}$$

message forwarding [36]. This choice does not affect our proof given the fairness assumption and the fact that data messages are always inspected at components and nodes according to their timestamps.

Apart from Multicast, we have four more system rules concerned with sending messages. The rules first identify the source and target of a message and then append the message to the mailbox of the target. In particular, Send1 (resp. Send2) forwards a data message (resp. a timestamp request) from a component to its parent. Send3 forwards a timestamp message (either request or reply) from an inner tree node to another connected one. Send4 forwards a timestamp from an inner node to its connected component. Here, we require a property of the message forwarding rules, commonly known as fairness, is that the sent message must eventually arrive at (the mailbox of) the target. This further implies that the message will eventually be processed.

Of all the system rules presented, the three main rules Send1, Recv, and FRecv expose the capability of components to the system level, namely sending a message, receiving a message, and losing a message.

4.4. Correctness of the tree structure

Thanks to our construction of the semantics, some properties concern with the communication among tree nodes and components in an *ABEL* system can be stated and established. The goal of the infrastructure, as mentioned before, is to forward the messages exchanged among components in a total order. In this section, we present arguments showing that this is the case.

First, we show that the allocation of message timestamps works as expected.

Proposition 4.1. *(Reliable Message Timestamp). If a component c requests a timestamp then it will eventually receive one. Furthermore, the timestamp is unique.*

Proof. Consider the path (i.e., a sequence of tree nodes) $(a_1, \ldots a_n)$ $(n \geq 2)$ from c to the root of the tree. The existence of such a path is guaranteed by definition of the tree topology. When c, i.e., a_1 sends a request for timestamp, either the rule C-out or C-cout is applied. In both cases, the system forwards the request to a connected node $\mathcal{T}(c) = a_2$ using the rule *Send*2. For each inner node a_k $(k > 1)$ along the path, by the fairness assumption we have that a_k eventually receives the request forwarded from its predecessor a_{k-1} (rule T-req). Thus, the request eventually arrives at the root a_n. Similarly, the timestamp issued (rule T-new) by a_n that propagates along the path in the reverse direction is also guaranteed to arrive at c. Moreover, since the root produces ever-increasing values of λ in reply to timestamp requests, the value received by the component is unique. □

Next, we provide several claims that characterize the properties of individual components and tree nodes. A few terminologies will be helpful. A λ-message is a message that is attached with the timestamp λ, i.e., it is of the form $(\lambda, \_)$. For a data message m, we write $\lambda(m)$ to denote its attached timestamp. A n-state component (or inner tree node) is one whose local counter is equal to n. For components, we use the term message delivery to refer to the events: a component sends or receives or discards a message. For inner nodes, message delivery is the event of forwarding a message when it has an expected timestamp.

Proposition 4.2. *A n-state component can only deliver n-message.*

Proof. By inspecting the relevant rules of component semantics, we have that a) in order to send a message, either the rules C-snd, C-csnd, C-fsend, or C-fcsend must be applicable; b) in order to evaluate a message for acceptance (i.e., either accept or discard the message), the component must use one of the rules C-frecv, C-recv, C-fcrecv, C-crecv. Since all the rules require the local counter to be equal to the message timestamp, a component in n-state can only deliver a n-message. □

Proposition 4.3. *A n-state inner node can only deliver n-message.*

Proof. By inspecting the relevant rules of tree node semantics. □

Proposition 4.4. *If a component delivers message $(l, \_)$ (for $l \geq 0$) then it has delivered all messages $(l', \_)$ with $l' < l$.*

Proof. Since the component is able to deliver a l-message, it must be in l-state ($l \geq 0$) by Proposition 4.2. However, the only way for the component to reach a l-state from its initial 0-state is by delivering l messages. By Proposition 4.2 and the fact that the local counter gets increased each time a message is delivered, the timestamps of these messages are smaller than l. □

Proposition 4.5. *If an inner node delivers a message $(l, \_)$ (for $l \geq 0$) then it has delivered all messages $(l', \_)$ with $l' < l$.*

Proof. Similar to the Proposition 4.4. □

Next, we show that the tree infrastructure disseminates messages in a broadcast manner: messages are only created by components and propagated through the inner nodes to other leaves. For this, we introduce the following lemmas.

Lemma 4.1. *(Leaf-node Agreement). For any component c, its connected tree node t and all $l \geq 0$, if c delivers message $m = (l, \_)$ then either t has already delivered m or t will deliver eventually it.*

Proof. See Appendix. □

Lemma 4.2. *(Node-node Agreement). For any two inner nodes t_1 and t_2, and all $l \geq 0$, if t_1 delivers message $m = (l, \_)$ then either t_2 has already delivered m or t_2 will eventually deliver it.*

Proof. We can assume that the message originates from t_1 (otherwise we swap t_1 and t_2). The proof is by double induction on message timestamp l and the length k of the path in the tree structure that connects t_1 and t_2. Let us consider the base case when $l = 0, k = 1$. Since t_1 delivers the message $(0, \_)$, by the rule Multicast, t_2 is forwarded the message due to the fact that they are neighbours. Further, the local counter of t_2 is initially 0, thus it eventually delivers $(0, \_)$.

Next, assume that the lemma holds for some timestamp $l \geq 0$ and $k = 1$. Since t_1 and t_2 are neighbours, we immediately have that if t_1 delivers $(l+1, \_)$ then t_2 eventually delivers $(l+1, \_)$.

Finally, we assume that the lemma holds for some $l \geq 0$ and some $k \geq 1$, we show that if t_1 delivers the message $(l, \_)$ then t_2 eventually delivers $(l, \_)$ where the distance between t_1 and t_2 is $k+1$. Let t_3 be the inner node right before t_2 in the path from t_1. We have that t_1 delivers the message and so does t_3 by induction hypothesis. Since t_3 delivers $(l, \_)$, it must have delivered all messages $(l', \_)$ for all $l' < l$ by Proposition 4.5. Note that t_3 and t_2 are neighbours, thus each message $(l', \_)$ delivered by t_3 must be forwarded to or from t_2. In other words, t_2 also has these l messages in its queues. By the design of semantic rules, these messages will eventually be sorted by t_2's priority queue ms. So t_2 eventually delivers all the messages including $(l, \_)$. □

To prove the correctness of the tree-based structure, we further need to show that every component in an *ABEL* system sees the same sequence of message delivery events. The proof is the direct consequences of the following Lemmas which assert that the components and tree nodes can not deliver messages out of order in the sense of Definition 4.1.

Definition 4.1. (Ordering) For any component c (resp. inner node t) and any two messages m, m' that c (resp. t) deliver, we say that c (resp. t) delivers m before m' if $\lambda(m) < \lambda(m')$.

Lemma 4.3. *For any component c, its connected tree node t, and any two messages m, m' that c, t deliver, if c delivers m before m' then t delivers m before m'.*

Proof. See Appendix. □

Lemma 4.4. *For any two inner nodes t_1, t_2 and any two messages m, m' that t_1, t_2 deliver, if t_1 delivers m before m' then t_2 delivers m before m'.*

Proof. See Appendix. □

We arrive at Theorem 4.1 which states that the order of messages delivered at components is total.

Theorem 4.1. *(Total order message delivery). For any two components c_1, c_2 and any two data messages m and m', if c_1 delivers m before m' then c_2 delivers m before m'.*

Proof. We have two cases to consider: a) both components share the same parent, b) c_1 and c_2 are connected to different inner nodes. The proof for case a is immediate from Lemma 4.3. For case b, the proof is straightforward from Lemmas 4.3, 4.4. □

Example 4. We show how the system rules work by deriving a possible execution of the *ABEL* program in Example 3. For brevity, we here illustrate with three components and one tree node, which is also the root. The system configuration is thus represented in the beginning as $\langle\!\langle \mathcal{T} \vdash \mathcal{C}s; \mathcal{S} \rangle\!\rangle$ where:

Components. $\mathcal{C}s = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ and for a component $\mathcal{C}_i$, its configuration is the tuple $\langle \gamma_i; ps; ctr\rangle_{\iota_i}$ with the following initializations

- attribute environment $\gamma_i = \mathrm{n}\, i \cdot \mathrm{s}\ \mathrm{tt}$;
- process $ps = \{\epsilon, pa(a, b)\}$ with subprocesses behaviour a, b;
- local coordinator $ctr = con(\bot, 0, \epsilon, \epsilon)$.

Tree Node. $\mathcal{S} = \{\langle 0, 0, \epsilon, \epsilon\rangle_\iota\}$.

System. The system consists of *ABEL* components plugged into topology $\mathcal{T} = [\iota_1 \to \iota, \iota_2 \to \iota, \iota_3 \to \iota, \iota \to \iota]$, rooted at the tree node given above.

Fig. 3 presents an execution, i.e., a sequence of rules applications from the initial configuration that corresponds to the execution of the *AbC* system in Example 1.

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; \underline{pa(a,b)}; con(\bot, \ldots)\rangle\ \langle \gamma_2; \underline{pa(a,b)}; con(\bot, \ldots)\rangle\ \langle \gamma_3; \underline{pa(a,b)}; con(\bot, \ldots)\rangle; \langle 0, 0, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{C-Tau, C-Tau, C-Tau}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; \underline{o(\alpha, \_)}\, b; con(\bot, \ldots)\rangle\ \langle \gamma_2; \underline{o(\alpha, \_)}\, b; con(\bot, \ldots)\rangle\ \langle \gamma_3; \underline{o(\alpha, \_)}\, b; con(\bot, \ldots)\rangle; \langle 0, 0, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{Send2, Send2, Send2}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; [a]\ \underline{i(\_, \_)}; con(\kappa, \ldots)\rangle\ \langle \gamma_2; [a]\ \underline{i(\_, \_)}; con(\kappa, \ldots)\rangle\ \langle \gamma_3; [a]\ \underline{i(\_, \_)}; con(\kappa, \ldots)\rangle; \langle 0, 0, \epsilon, (\kappa, \iota_2) : (\kappa, \iota_3) : (\kappa, \iota_1) : \epsilon\rangle\rangle\!\rangle$$

$$\{\text{C-Tau, C-Tau, C-Tau}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; [a\, b]; con(\kappa, \ldots)\rangle\ \langle \gamma_2; [a\, b]; con(\kappa, \ldots)\rangle\ \langle \gamma_3; [a\, b]; con(\kappa, \ldots)\rangle; \underline{\langle 0, 0, \epsilon, (\kappa, \iota_2) : (\kappa, \iota_3) : (\kappa, \iota_1) : \epsilon\rangle}\rangle\!\rangle$$

$$\{\text{Send4, Send4, Send4}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; \underline{[a\, b]}; con(2, \ldots)\rangle\ \langle \gamma_2; \underline{[a\, b]}; con(0, \ldots)\rangle\ \langle \gamma_3; \underline{[a\, b]}; con(1, \ldots)\rangle; \langle 3, 0, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{C-Tau, C-Tau, C-Tau}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; \underline{o(\alpha, nil)}\ [b]; con(2, 0, \ldots)\rangle\ \langle \gamma_2; \underline{o(\alpha, nil)}\ [b]; con(0, 0, \ldots)\rangle\ \langle \gamma_3; \underline{o(\alpha, nil)}\ [b]; con(1, 0, \ldots)\rangle; \langle 3, 0, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{C-Tau, Send1, C-Tau}\}\ \overset{snd(0,m_2)}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; [a\, b]; con(2, 0, \ldots)\rangle\ \langle \gamma_2; \langle\rangle_\emptyset\ [b]; con(\bot, 1, \ldots)\rangle\ \langle \gamma_3; [a\, b]; con(1, 0, \ldots)\rangle; \underline{\langle 3, 0, \epsilon, (0, m_2) : \epsilon\rangle}\rangle\!\rangle$$

$$\{\text{T-Tau,Multicast,C-Tau}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1; [a\, b]; con(2, 0, (0, m_2) : \epsilon, \epsilon)\rangle\ \langle \gamma_2; [b]; con(\bot, 1, \ldots)\rangle\ \langle \gamma_3; [a\, b]; con(3, 0, (0, m_2) : \epsilon, \epsilon)\rangle; \langle 3, 1, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{Recv,Frecv,C-Tau}\}\ \overset{recv(0,m_2)}{\leadsto}{}^*\ \overset{\widetilde{recv(0,m_2)}}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1'; [a]\ \langle\rangle_\emptyset; con(2, 1, \ldots)\rangle\ \langle \gamma_2; [b]; con(\bot, 1, \ldots)\rangle\ \langle \gamma_3; \underline{o(\alpha, nil)}\ [b]; con(1, 1, \ldots)\rangle; \langle 3, 1, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{Send1}\}\ \overset{snd(1,m_3)}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1'; [a]; con(2, 1, \ldots)\rangle\ \langle \gamma_2; \langle\rangle_\emptyset\ [b]; con(\bot, 1, \ldots)\rangle\ \langle \gamma_3; \langle\rangle_\emptyset\ [b]; con(1, 1, \ldots)\rangle; \underline{\langle 3, 1, \epsilon, (1, m_3) : \epsilon\rangle}\rangle\!\rangle$$

$$\{\text{T-Tau,Multicast}\}\ \overset{\tau}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1'; [a]; con(2, 1, (1, m_3) : \epsilon, \epsilon)\rangle\ \langle \gamma_2; [b]; con(\bot, 1, (1, m_3) : \epsilon, \epsilon)\rangle\ \langle \gamma_3; [b]; con(\bot, 2, \ldots)\rangle; \langle 3, 2, \epsilon, \epsilon\rangle\rangle\!\rangle$$

$$\{\text{FRecv,Recv,C-Tau}\}\ \overset{\widetilde{recv(1,m_3)}}{\leadsto}{}^*\ \overset{recv(1,m_3)}{\leadsto}{}^*$$

$$\langle\!\langle \mathcal{T} \vdash \langle \gamma_1'; [a]; con(2, 2, \epsilon, \epsilon)\rangle\ \langle \gamma_2'; \langle\rangle_\emptyset; con(\bot, 2, \epsilon, \epsilon)\rangle\ \langle \gamma_3; \langle\rangle_\emptyset; con(\bot, 2, \ldots)\rangle; \langle 3, 2, \epsilon, \epsilon\rangle\rangle\!\rangle$$

Fig. 3. A possible derivation of the running example in *ABEL*.

It is worth commenting that, due to nondeterminism, the system may evolve in a way that the message timestamps allocated for the three components are $(\mathcal{C}_1, 1)$, $(\mathcal{C}_2, 0)$ and $(\mathcal{C}_3, 2)$, respectively. In this scenario, $\mathcal{C}_2$ is the first to send its message to which $\mathcal{C}_1$ accepts and $\mathcal{C}_3$ discards it. By accepting message $(0, m_2)$, component $\mathcal{C}_1$ itself is not able to send the promised message (but still holds the timestamp 1). If this were the case, other components that obtained timestamps higher than $\mathcal{C}_1$, such as $\mathcal{C}_3$ in this example could have been blocked. However, by the design of component semantics, rule C-fsend enforces $\mathcal{C}_1$ to send out an empty message that contributes to compensating the value of the local counter at $\mathcal{C}_3$, hence enabling it to proceed.

5. Operational correspondence

In this section, we prove the operational correspondence between *AbC* and *ABEL*. In particular, we show that *ABEL* enjoys two crucial properties. The first one guarantees *soundness*, i.e., the execution of *ABEL* is in agreement with *AbC* semantics. The second one is concerned with *liveness*, i.e., *ABEL* does not get stuck if *AbC* does not.

The translation **tr** introduced in Section 3.2 is static, in the sense that it transforms *AbC* specification into runnable *ABEL* program, and is not suitable to relate running processes. Because of this, in order to establish the above-mentioned properties, in this section we introduce the appropriate notations to relate *AbC* and *ABEL* at *runtime*. The mapping tr (Table 15) transforms an *AbC* component $\Gamma : P$ into a pair consisting of the translations of the attribute environment and the process. tr is defined inductively over the structure of *AbC* process. For other *AbC* terms, tr coincides with **tr**. By relying on tr and the additional definitions given below, we can obtain *ABEL* components and systems from *AbC* ones.

Table 15
Relating *AbC* terms to *ABEL* runtime entities (tr).

$tr(C_1 \parallel \ldots \parallel C_n)$	$= (tr(C_1), \ldots, tr(C_n))$
$tr(\Gamma : P)$	$= (tr(\Gamma), tr(P))$
$tr\llbracket \alpha^o.P \rrbracket$	$= o(tr\llbracket \alpha^o \rrbracket, tr\llbracket P \rrbracket)$
$tr\llbracket \alpha^i.P \rrbracket$	$= i(tr\llbracket \alpha^i \rrbracket, tr\llbracket P \rrbracket)$
$tr\llbracket \Sigma_{j=1}^{n} \alpha_j^i.P_j \rrbracket$	$= ci(\{tr\llbracket \alpha_1^i \rrbracket, tr\llbracket P_1 \rrbracket\}, \ldots, \{tr\llbracket \alpha_n^i \rrbracket, tr\llbracket P_n \rrbracket\}])$
$tr\llbracket \Sigma_{j=1}^{n} \alpha_j^o.P_j \rrbracket$	$= co(\{tr\llbracket \alpha_1^o \rrbracket, tr\llbracket P_1 \rrbracket\}, \ldots, \{tr\llbracket \alpha_n^o \rrbracket, tr\llbracket P_n \rrbracket\}])$
$tr\llbracket P_1 \vert \ldots \vert P_m \rrbracket$	$= pa([tr\llbracket P_1 \rrbracket, \ldots, tr\llbracket P_m \rrbracket])$
$tr\llbracket 0 \rrbracket$	$= nil$

Definition 5.1 *(Context).* A *component context* $\xi = \langle \bullet; con \rangle_\iota$ is a component configuration without attribute environment and process. Given an *AbC* component $C = \Gamma : P$, we write $\xi[tr(C)]$ to represent the running component $\mathcal{C} = \langle \gamma; \{\epsilon, b\}; con \rangle_\iota$ where $tr(\Gamma) = \gamma, tr(P) = b$ and ϵ the empty binding list.

Let $\mathcal{E}$ be a set of component contexts, a *system context* $\mathcal{K} = \langle\langle \mathcal{T} \vdash \mathcal{E}; \mathcal{S} \rangle\rangle$ is a system configuration with the set of components replaced by $\mathcal{E}$. Given an *AbC* system S, we write $\mathcal{K}[tr(S)]$ to represent the running system $\mathcal{G} = \langle\langle \mathcal{T} \vdash \xi[tr(C_1)] \ \ldots \ \xi[tr(C_n)]; \mathcal{S} \rangle\rangle$ for each component C_i of S.

Each component context has an identifier (namely the address of the coordinator con) and a system context conforms to the constraints listed in Section 4.3. However, a system context is not executable if it is not equipped with appropriate code, say $tr(S)$ corresponding to *AbC* system S.

Moreover, we say a system context $\mathcal{K}$ is consistent, denoted as $\mathbb{K}$ if all the counters of component contexts in $\mathcal{K}$ have the same value. Note that, since we do not consider the states of tree nodes in the above definition, there are many contexts that are consistent for a given value of the components' local counters.

In *ABEL*, substitutions are not applied instantaneously: after successfully executing an input action, the binding list σ is accumulated with the received message (see Section 4.1). In this way, the value of any variable if required is looked up in σ at each execution step. Instead in *AbC*, the application of a substitution happens at once. Thus, the mapping tr above and the introduction of *ABEL* code into contexts may lose track of substitutions. To address this gap, we need some notations to equate *ABEL* components and systems. For our purpose, we introduce the notion of *equivalent configurations*.

Definition 5.2. (Equivalent Configurations) Two *ABEL* component configurations $\mathcal{C}_1 = \langle \gamma; \{\sigma_1, b_1\}; con \rangle$ and $\mathcal{C}_2 = \langle \gamma; \{\sigma_2, b_2\}; con \rangle$ are equivalent, written as $\mathcal{C}_1 \simeq_c \mathcal{C}_2$ if there exist *AbC* processes P_1, P_2 such that $tr(P_1) = b_1$ and $tr(P_2) = b_2$ and $P_1\sigma_1 \equiv P_2\sigma_2$. Two *ABEL* system configurations $\mathcal{G}_1 = \langle\langle \mathcal{T} \vdash \mathcal{C}s_1; \mathcal{S} \rangle\rangle$ and $\mathcal{G}_2 = \langle\langle \mathcal{T} \vdash \mathcal{C}s_2; \mathcal{S} \rangle\rangle$ are equivalent, written as $\mathcal{G}_1 \simeq_s \mathcal{G}_2$ if the components in $\mathcal{C}s_1$ and $\mathcal{C}s_2$ are pair-wise equivalent.

Based on the prepared notations, we now define what it means by correspondence between *AbC* and *ABEL*.

Definition 5.3. (System Correspondence $\simeq$) An *ABEL* system $\mathcal{G}$ corresponds to an *AbC* system S, written as $\mathcal{G} \simeq S$ if $\exists \mathbb{K}$ s.t. $\mathbb{K}[tr(S)] \overset{\tau}{\rightsquigarrow}{}^* = \mathcal{G}$ or $\mathbb{K}[tr(S)] \simeq_s \mathcal{G}$.

In fact, not all system configurations are of interest in the proofs, we only concern with systems that are consistent and hence their correspondence with *AbC* systems can be established. Let $\mathcal{G}(n)$ denote a consistent (*ABEL*) configuration with $n \geq 0$ the value of the components' local counters.

Proposition 5.1 claims that there exists interleaved executions of components in which, starting from a consistent configuration $\mathcal{G}(n)$, leads to another consistent configuration $\mathcal{G}(n+1)$. We assume there are sending components, otherwise there would be no interactions.

Proposition 5.1. *Given a consistent configuration $\mathcal{G}(n)$, if there is at least one sending component in $\mathcal{G}(n)$ then there exists interleavings of component execution that lead to $\mathcal{G}(n+1)$.*

Proof. See Appendix. □

Practically speaking, consistent configurations are not likely to occur during the system evolution. For example, consider a component connected to the root of the tree that keeps sending out its internal values (e.g., plays the role of a clock). The counter of this component may increase faster than that of the components located further away from the root. Consistent components states may be enforced by using more messages for coordination, but that would introduce unnecessary overhead.

Let us denote by $\Rightarrow$ any sequence of *ABEL*'s components transitions that satisfy the property in Proposition 5.1.

Let β' ranges over the set of *ABEL* components labels, i.e., $\{snd(\rho, (\lambda, m)),\ recv(\lambda, m),\ \widetilde{recv}(\lambda, m)\}$. Let β ranges over the set of public *ABEL* systems labels, i.e., $\{snd(\lambda, m),\ recv(\lambda, m), \widetilde{recv}(\lambda, m)\}$. For convenience, we prefix a label variable with ! to mean it is a sending label, ? for receiving and : for discarding. The following function f relates the transitions labels between *ABEL* and *AbC* for components and systems, respectively.

Definition 5.4 *(Labels Correspondence).* Let f be a function that maps *ABEL* labels to *AbC* labels as follows. $f(snd(\rho, (\lambda, m)) = f(snd((\lambda, m))) = \Gamma \triangleright \overline{\Pi}(v)$, $f(recv(\lambda, m)) = \Gamma \triangleright \Pi(v)$, $f(\widetilde{recv}(\lambda, m)) = \widetilde{\Gamma \triangleright \Pi}(v)$, where $m = (tr(\Gamma), tr(\Pi), tr(v))$.

The simulation relation ($\preceq$) between an *ABEL* system and an *AbC* system can now be defined as.

Definition 5.5. $\preceq$ is a simulation relation between an *ABEL* system $\mathcal{G}$ and an *AbC* system S, written as $\mathcal{G} \preceq S$ if $\mathcal{G} \simeq S$ and $\mathcal{G} \Rightarrow \mathcal{G}'$, then $S \to S'$ and $\mathcal{G}' \preceq S'$, where $\to$ is a single *AbC* transition.

That is, whenever an *ABEL* system makes a sequence of moves (i.e., rule applications) from a consistent configuration to another, the corresponding *AbC* system can make a transition to another such that the latter two are still correspondent.

The *soundness* property is established in two steps, first at the component level (Lemma 5.1), where we consider components in isolation and then at the system level (Theorem 5.1) where we take into account the interactions of the components. The proof amounts to showing that an *AbC* system can match all the moves of its translation in *ABEL* in the sense of Definition 5.5.

Lemma 5.1. *Given C a single* AbC *component and any context ξ such that $\xi[tr(C)]$ is capable of doing a β' transition. We have that if $\xi[A] \overset{\beta'}{\rightsquigarrow}{}^* \mathcal{C}'$ then there exists C' and ξ' such that $C \xmapsto{f(\beta')} C'$ and $\mathcal{C}' \overset{\tau}{\rightsquigarrow}{}^* = \xi'[tr(C')]$, or $\mathcal{C}' \overset{\tau}{\rightsquigarrow}{}^* \simeq_c \xi'[tr(C')]$.*

Proof. See Appendix. □

Theorem 5.1. *(Soundness) Let S be an* AbC *system and $\mathbb{K}$ a consistent context, if $\mathbb{K}[tr(S)] \Rightarrow \mathcal{G}'$, then there exist S' and $\mathbb{K}'$ such that $S \to S'$ and $\mathcal{G}' \overset{\tau}{\rightsquigarrow\!\!\!\Rightarrow}{}^* = \mathbb{K}'[tr(S')]$, or $\mathcal{G}' \overset{\tau}{\rightsquigarrow\!\!\!\Rightarrow}{}^* \simeq_s \mathbb{K}'[tr(S')]$.*

Proof. Consider an *AbC* system $S = \Gamma_1 : P_1 \ \| \ldots \| \ \Gamma_n : P_n$, then

$$\mathcal{G}(c) = \mathbb{K}[tr(S)] = \langle\!\langle \mathcal{T} \vdash \langle \gamma_1; \{\epsilon, b_1\}; con_1 \rangle \ldots \langle \gamma_n; \{\epsilon, b_n\}; con_n \rangle; \mathcal{S} \rangle\!\rangle$$

is the *ABEL* system resulting from filling the set of translated components into $\mathbb{K}$, where $\forall 1 \le i \le n. \gamma_i = tr(\Gamma_i)$, $b_i = tr(P_i)$ and c the value of components counters.

If no components can output in $\mathcal{G}$, the theorem trivially holds. Therefore, we consider cases where there is at least one component that can send. By Lemma 4.1, we know that there must exist one component, say i that obtains the smallest timestamp, say λ such that $\lambda = c$.

We now consider the evolution of $\mathcal{G}$. The only possibility for $\mathcal{G}$ to evolve into another consistent state is by

$$\begin{aligned} \mathcal{G} = \ & \langle\!\langle \mathcal{T} \vdash \langle \gamma_i; \{\epsilon, b_i\}; con_i \rangle_{\iota_i}\ \mathcal{C}; \mathcal{S} \rangle\!\rangle \\ & \overset{!\beta}{\rightsquigarrow\!\!\!\Rightarrow}{}^* \overset{\bar{l}}{\Rightarrow} \\ = \ & \langle\!\langle \mathcal{T} \vdash \langle \gamma_i'; \{\sigma_i, b_i'\}; con_i' \rangle_{\iota_i}\ \mathcal{C}'; \mathcal{S}' \rangle\!\rangle = \mathcal{G}' \end{aligned}$$

where $!\beta = snd(\lambda, (\gamma_i, \pi, v)) = snd(\lambda, m_i)$ and $\overset{\bar{l}}{\Rightarrow}$ denotes any of the following sequences

- permutations of labelled transitions of the form $\overset{?\beta}{\rightsquigarrow\!\!\!\Rightarrow}{}^*$ and $\overset{:\beta}{\rightsquigarrow\!\!\!\Rightarrow}{}^*$ such that $|\bar{l}| + 1 = n$,
- sequences of $\overset{?\beta}{\rightsquigarrow\!\!\!\Rightarrow}{}^*$ or $\overset{:\beta}{\rightsquigarrow\!\!\!\Rightarrow}{}^*$ such that $|\bar{l}| + 1 = n$.

Intuitively, these sequences mean that component $\mathcal{C}_i$ sends out its message while the rest components deliver this message. Let I_e, I_s be the index sets of components in $\mathcal{C}'$ that have accepted and discarded the message (λ, m_i), respectively. Let $\rho_i = \mathcal{T}(\iota_i)$ be the connected tree node of component $\mathcal{C}_i$, we construct the labels for components by defining $!\beta' = snd(\rho_i, (\lambda, m_i))$, $?\beta' = ?\beta$ and $:\beta' = :\beta$. Then, the transitions at the individual components can be further elaborated as follows. $\forall j \in I_e \,.\, \mathcal{C}_j \overset{?\beta'}{\rightsquigarrow}{}^* \mathcal{C}'_j$ and $\forall k \in I_s \,.\, \mathcal{C}_k \overset{:\beta'}{\rightsquigarrow}{}^* \mathcal{C}'_k$.

By Lemma 5.1, we can derive transitions for each *AbC* component C_* in S from its corresponding *ABEL* component $\mathcal{C}_*$ in $\mathcal{G}$, in particular: $C_i \xmapsto{f(!\beta')} C'_i$ and $\forall j \in I_e \,.\, C_j \xmapsto{f(?\beta')} C'_j$, $\forall k \in I_s \,.\, C_k \xmapsto{f(:\beta')} C'_k$.

By *AbC*'s system semantics, the following transition can be derived for S:

$$S \equiv C_i \parallel \prod_{j \in I_e} C_j \parallel \prod_{k \in I_s} C_k \xrightarrow{\Gamma_i \triangleright \overline{\Pi}_i(v) = f(!\beta')} C'_i \parallel \prod_{j \in I_e} C'_j \parallel \prod_{k \in I_k} C'_k \equiv S'$$

Consider the context $\mathcal{K} = \langle\langle \mathcal{T} \vdash \langle \bullet; con'_1 \rangle \; \ldots \; \langle \bullet; con'_m \rangle; \mathcal{S}' \rangle\rangle$, the components coordinators con_i' in it have their counter values increased by 1. Thus $\mathcal{K}$ is consistent and we take $\mathbb{K}' = \mathcal{K}$. In $\mathbb{K}'[tr(S')]$, all components are pair-wise equivalent to components in $\mathcal{G}'$, hence $\mathbb{K}'[tr(S')] \simeq_s \mathcal{G}'$. □

The above theorem is not sufficient to guarantee the correctness because an empty *ABEL* system that does nothing can still satisfy it. The *liveness* property justifies that the generated implementation does some work.

Theorem 5.2. *(Liveness). Given an* AbC *system S, if $S \to S'$, then $\mathcal{G} = \mathbb{K}[tr(S)] \Rightarrow \mathcal{G}'$. Moreover, given $S = C \parallel S_1$, if $C \xmapsto{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C'$ and $S_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S'_1$ and $\mathbb{K}[tr(S)] \overset{!\beta}{\rightsquigarrow}{}^* \Rightarrow \mathcal{G}'$, then $\mathcal{G}' \simeq C' \parallel S'_1$, where $f(!\beta) = \Gamma \triangleright \overline{\Pi}(\tilde{v})$.*

Proof. See Appendix. □

Intuitively, Theorem 5.2 says that if an *AbC* system S can evolve to S' by performing a single transition $\to$ then the corresponding *ABEL* one $\mathcal{G}$ can also evolve to $\mathcal{G}'$ by performing a sequence of components transitions $\Rightarrow$ (Proposition 5.1). However, because the component that outputs in $\mathcal{G}$ may be different from the one in S, we cannot conclude a correspondence between S' and $\mathcal{G}'$. The second part of the theorem strengthens the hypothesis in order to recover this correspondence. It states that if a specific *AbC* component C actually sent a message and it is the corresponding *ABEL* component the sending one in $\mathcal{G}$ (implied by the first label), then we have the correspondence between the two derivatives.

Generally, an *ABEL* system obtained by the translation can not simulate the original *AbC* system for two reasons. First, *ABEL* components must ask for fresh timestamps before sending which makes their sending order is somewhat fixed in the next steps of the system evolution. Second, practically the implementation has resolved the non-deterministic choice by considering the choice branches from leftmost to rightmost. Both amount to reducing nondeterminism of the original *AbC* specification. However, since any execution of an implementation generated by the translation is allowed by the related *AbC* specification, the transition system of the *AbC* specification subsumes the transition system of the corresponding *ABEL* implementation. For this reason, we conclude that any state-based properties, i.e., safety that hold in an *AbC* model are guaranteed also in the implementation.

6. Related works

Several implementations of the attribute-based interaction have been proposed [7,8,16]. Two of them, namely AbaCus [7], a Java implementation, and Goat [8], a Go implementation, rely on a message broker to distribute messages in a broadcast manner. It is the receiving component that decides whether to use or discard a forwarded message, by checking the sending and the receiving predicates. In both implementations, non-deterministic choice is modelled as an if-then-else construct and process recursion as an infinite loop. We argue that *ABEL* APIs are much closer to the *AbC*'s syntax than those of GoAT and AbaCus. This facilitates automatic translation from *AbC* specifications into *ABEL* programs. Another implementation, named AErlang [16], extends *Erlang* processes with the key ingredients of *AbC* programming idioms, but it cannot be considered a faithful implementation of *AbC* because components can only have a single thread of control and broadcast messages are not ordered. Moreover, none of the three *AbC* implementations considers the issue of correctness.

Despite the different host languages, *ABEL* and GoAT share some similarities in terms of implementation as they rely on the tree-shaped distributed infrastructure proposed in [18]. GoAt provides also a ring and a cluster shape of the infrastructure. Performance of GoAT is evaluated through simulation in [18,8], and considering the same study we have shown that *ABEL* can handle a larger number of components [27]. In [18], the proofs of correctness focused solely on the convergence of counters values at every component and infrastructure nodes. Here, we precisely identify the sequence of *ABEL* transitions that correspond to the *AbC* lock-step evolution. Besides, [18] and [8] did not consider the possibility of system-level deadlock as illustrated in Example 4.

The approach to *ABEL* semantics was inspired by [24], aiming at achieving modularity. This is done specifically by decorating the local transition labels with variables (e.g., ρ, μ in Section 4), making it possible to describe the semantics of components and tree nodes in isolation. Such transitions are made available at system-level so that the relevant rules can exploit labels variables to fulfil the designed purpose. Our work on formalizing *ABEL* is also related to the work in [34] where a similar exercise has been done for ContextErlang, a programming framework for a Context-Oriented approach to multi-agent agents programming.

We would also like to mention two other efforts that are concerned with correctness proof of implementations. In [37], Java code is generated from spi-calculus specifications and a simulation relation is established between the sequential part of the spi-calculus and its translation in Java. Since only the sequential part of the calculus was considered they used a subset of an existing API wrapper. The set of APIs used is formalized and the proofs are fully carried out, but the underlying library implementation is abstracted and assumed correct. In [38] a parallel simulator of a multi-cores memory system is implemented in ABS [33] and it is proved that the implementation simulates the proposed operational model of the memory system. However, the proof is only sketched as the ABS implementation is only described informally.

Among the many works concerned with proving the correspondence between different formalisms based on their operational semantics, our work has also been influenced by [39,40]. The former presents two actor languages and a semantics preserving translation between them, while the latter considers variants of the KLAIM language and proves their bisimilarity under weak fairness assumption. In our case, source and target languages are very different in nature and we are concerned with an actual implementation which requires dealing with the nondeterminism in the source language, which enabled us to prove only similarity but not bisimilarity.

Obviously, *AbC* is not the only formalism for building software-intensive systems. In fact, several frameworks and runtime environments have been proposed [41–44] for similar purposes. Many of them are based on the notions of components and component ensembles that can be traced back to the SCEL language [45,1]. The language allows the specification of components that during system evolution, can dynamically organize themselves into ensembles based on the components knowledge. Helena [42] relies on the notion of role to represents the capability of components. Roles are helpful to group different components for specific collaborative tasks. The approach is somewhat static in the sense that roles are assigned at design time and each component must indicate the ensembles it is part of. DEECo [44] supports specifying components and ensembles as first-class entities. Components exchange knowledge with each other in the same ensemble via a hidden ensemble's coordinator. Ensembles need to be specified explicitly and components memberships have to be periodically checked. TCOEL [43] is a more recent proposal that extends the DEECo to incorporate more expressive membership conditions and dynamic groups formation, achieved by leveraging advanced features of Scala. By contrast to the above, communication groups (ensembles) in *AbC* and *ABEL* are more abstract as they only arise on the basis of predicates satisfaction.

7. Concluding remarks

In this paper, we have presented *ABEL*, an implementation of the *AbC* calculus in *Erlang* and proved its correctness. The correctness of *ABEL* is the result of a series of developments. First, the syntax gap between *AbC* and *ABEL* is bridged by a translation from the former to the latter. Second, the operational details of *ABEL* APIs and its implementation are distilled into a formal semantics that serves as a basis for reasoning. Third, *ABEL*'s coordination infrastructure is proved to agree with the intended meaning of the parallel composition of *AbC*. Finally, we have shown that there is an operational correspondence between the labelled transitions systems of *ABEL* and *AbC*. Specifically, we have proved that to any *AbC* transition there correspond an *ABEL* transition. Thanks to the careful design of the semantics and of the translation, the relevant correctness proofs are relatively straightforward.

Our main technical contribution is therefore the validation that the *ABEL* system obtained from the proposed translation can be simulated by the original *AbC* specification while guaranteeing liveness. Our constructions pave the way to the systematic development of systems via code generation from *AbC* specifications. And this can be done after analysing the latter with formal tools [14,9] that permits removing early design errors (e.g., fixing concurrency bugs), as well as checking (state-based) properties of interest.

Our proofs do not directly involve the *Erlang* language but an intermediate language proposed for *ABEL*. This is justified by the fact that the implementations of the tree and component coordinators (Section 3.3) are not part of the language. On the other hand, the methodology is general enough and may be used to prove implementation correctness in similar settings. We also think that the semantics proposed in this paper can be used to guide *AbC* implementations in another programming languages, in particular in actor-based ones, like Scala.

Our experience with *AbC* suggests that the globally synchronous semantics appears to be too demanding when large distributed systems are considered. In many applications, any arbitrary order of the exchanged messages across distributed components does not change the outcome (e.g., the toy system in Example 1 and others in [27]). Also, global broadcast is not always necessary, for example when communicating between different groups or within the same group. In the future, we would like to investigate on these practical aspects by considering to what extent we can reduce the unnecessary checks on broadcast messages, as well as to increase the asynchrony between components execution. In addition, a type system would be helpful to ensure predicates and attributes are used in an appropriate way.

CRediT authorship contribution statement

All authors contributed equally to this work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. The proofs

Proof of Lemma 4.1. *For any component c, its connected tree node t and all $l \geq 0$, if c delivers message $m = (l, \_)$ then either t has already delivered m or t will deliver eventually it.*

Proof. The proof is by induction on message timestamp l. Let us consider the base case when $l = 0$. Since the message timestamp is unique (Proposition 4.1), there can be either a) c sends m to t or b) t sends m to c.

In case a) c delivers the message m by applying either C-send, C-csend, C-fsend or C-fcsend. In all scenarios, the system rule Send1 will eventually be applied. Since t is connected to c, Send1 appends m to the message queue q of t. By tree node semantics, t will eventually process the message m by applying the rule T-msg that places m inside t's priority queue ms. Since the local counter of t is initially 0, we know that the rule T-fwd will eventually be applied, i.e., t eventually delivers m.

In case b) we have that t delivers the message by applying the rule T-fwd, which in turn triggers the application of the system rule Multicast. Since t and c are connected, the message is appended to the message queue q of c. By the component semantics, c will eventually apply the rule C-msg to place m inside the priority queue ms. Since the local counter of c is initially 0, we know that one of the relevant component rules concerning the evaluation of m for acceptance will eventually be applied, i.e., either C-recv, C-frecv, C-crecv or C-fcrecv. In all scenarios, we have that c eventually delivers m.

Assume that the lemma holds for some $l \geq 0$ we show that it also holds for $l + 1$. Since c and t deliver the message $(l, \_)$ by the hypothesis, we have that their local counters are equal to $l + 1$ by the fact that delivering a message increases the local counters by 1. We can now proceed by case analysis on the possible origins of the message $m = (l + 1, \_)$ and use similar arguments as in the base case. □

Proof of Lemma 4.3. *For any component c, its connected tree node t, and any two messages m, m' that c, t deliver, if c delivers m before m' then t delivers m before m'.*

Proof. Since c delivers m before m', by definition we have that $\lambda(m) < \lambda(m')$. By Lemma 4.1, t also delivers m and m'. Therefore, t delivers m before m'. □

Proof of Lemma 4.4. *For any two inner nodes t_1, t_2 and any two messages m, m' that t_1, t_2 deliver, if t_1 delivers m before m' then t_2 delivers m before m'.*

Proof. Since t_1 delivers m before m', by definition we have that $\lambda(m) < \lambda(m')$. By Lemma 4.2, t_2 also delivers m and m'. Therefore, t_2 delivers m before m'. □

Proof of Proposition 5.1. *Given $\mathcal{G}(n)$, if there is at least one sending component in $\mathcal{G}(n)$ then there exists interleavings of components execution that lead to $\mathcal{G}(n + 1)$.*

Proof. Consider the evolution of $\mathcal{G}(n)$, we have that a message attached with timestamp $\lambda = n$ will be sent by one of the components. Let us denote by λ-$\mathcal{C}_i$, λ-$\mathcal{C}_j, \ldots$ the events of delivering a λ-message at components $\mathcal{C}_i, \mathcal{C}_j \ldots$. By Theorem 4.1, any sequence of events produced by a scheduler must satisfy the condition: $\forall \mathcal{C}_*$, $l \geq 1$, λ-$\mathcal{C}_*$ precedes $(\lambda + l)$-$\mathcal{C}_*$. Moreover, $\forall i, j \,.\, \mathcal{C}_i \neq \mathcal{C}_j$, any two events $(\lambda + l)$-$\mathcal{C}_i$,λ-$\mathcal{C}_j$ are independent. Thus, on a particular sequence of events, whenever there is a $(\lambda + 1)$-event appears before a λ-event (at two different components), we can swap such a pair to produce the desired order. By induction on the total number of out-of-order pairs, we will eventually obtain a schedule where all λ-events precede any $\lambda + 1$-event. Since the λ-events cause the local counters of the components to increase by 1, the schedule results in the consistent configuration $\mathcal{G}(n + 1)$. □

Proof of Lemma 5.1. *Given C a single* AbC *component and any context ξ such that $\xi[tr(C)]$ is capable of doing a β' transition. We have that if $\xi[A] \overset{\beta'}{\rightsquigarrow}{}^{*}\, C'$ then there exists C' and ξ' such that $C \xmapsto{f(\beta')} C'$ and $\mathcal{C}' \overset{\tau}{\rightsquigarrow}{}^{*} = \xi'[tr(C')]$, or $\mathcal{C}' \overset{\tau}{\rightsquigarrow}{}^{*} \simeq_c \xi'[tr(C')]$.*

Proof. By case analysis on the structure of the process P of the *AbC* component $\Gamma : P$ and by transition induction [46] on the derivation of the corresponding *ABEL* component.

We assume any component context $\xi = \langle \bullet; con(\lambda, c, ms, q) \rangle$ that satisfies the hypothesis of the lemma, for some appropriate values of message timestamp λ, local counter c and message queues ms, q.

a) **Case of** $P = (\tilde{E})@(\Pi_s).[\tilde{a} := \tilde{E}']P'$. We consider the corresponding *ABEL* runtime component $\mathcal{C} = \xi[tr(\Gamma : P)] = \langle \gamma, \{\epsilon, o(\alpha, b)\}; con\rangle$ where $\gamma = tr(\Gamma), b = tr(P')$.
There are two cases where $\mathcal{C}$ can perform a labelled transition: one via the rule C-snd and the other via C-frecv.
   - Rule C-snd is applied. We have that $\langle \gamma, \{\epsilon, o(\alpha, b)\}; con(c, \ldots)\rangle \overset{!\beta'}{\rightsquigarrow}{}^{*} \langle \gamma', \{\epsilon, b\}; con'(c+1, \ldots)\rangle = \mathcal{C}'$, where $\gamma' = \mathbf{upd}(\gamma, \epsilon, \alpha_u)$ and $!\beta' = snd(\rho, (c, m))$.
   At the same time, the *AbC* component can evolve with the transition label $f(!\beta')$ $\Gamma : (\tilde{E})@(\Pi_s).[\tilde{a} := \tilde{E}']P' \xrightarrow{\Gamma \triangleright \overline{\Pi}_s(\tilde{v}) = f(!\beta')} \Gamma[\tilde{a} := [\![\tilde{E}']\!]_\Gamma] : P' = \Gamma_1 : P'$ by the application of rule BRD.
   Consider the translation $tr(\Gamma_1)$ that produces an attribute environment γ_1. By assuming that the **upd** function used in the *ABEL* semantics correctly implements the update operation of *AbC*, we have $\gamma_1 = \gamma'$. Therefore, if we take $\xi' = \langle \bullet; con'(c+1, \ldots)\rangle$, then $\xi'[tr(\Gamma_1 : P)] = \langle \gamma_1; \{\epsilon, b\}; con'(c+1, \ldots)\rangle = \mathcal{C}'$.
   - Rule C-frecv is applied. We have that $\langle \gamma, o(\alpha, b); con(c, \ldots)\rangle \overset{:\beta'}{\rightsquigarrow}{}^{*} \langle \gamma, b'; con'(c+1, \ldots)\rangle$ where $:\beta' = \widetilde{recv(c, m)}$.
   At the same time, the original *AbC* component can perform $\Gamma : P \xrightarrow{\widetilde{\Gamma' \triangleright \Pi'(v)} = f(:\beta')} \Gamma : P$ by the application of rule FBRD.
   Therefore, we take $C' = C$ and choose $\xi' = \langle \bullet; con'(c+1, \ldots)\rangle$, the pair C', ξ' satisfy the claim.

b) **Case of** P = $(\Pi_r)(\tilde{x}).[\tilde{a} := \tilde{E}']P'$. Consider the *ABEL* component $\mathcal{C} = \xi[tr(\Gamma : P)] = \langle \gamma; \{\epsilon, i(\alpha, b)\}; con\rangle$ where $b = tr(P')$.
This is a receiving component that can either accept or reject a message.
If $\mathcal{C}$ discards a message by the application of rule C-frecv, we resolve it similar to the previous sub case of a). Let us consider the other case when rule C-recv is applied, i.e., $\mathcal{C}$ actually consumes a message. Accordingly, $\langle \gamma; \{\epsilon, i(\alpha, b\}; con(c, \ldots)\rangle \overset{?\beta'}{\rightsquigarrow}{}^{*} \langle \gamma'; \{\sigma, b\}; con'(c+1, \ldots)\rangle = \mathcal{C}'$ where $?\beta' = recv(c, m)$ for some message $m = (\gamma_s, \pi_s, v)$ that satisfies the receiving condition, $\sigma = \alpha_x\ v$ and $\gamma' = \mathbf{upd}(\gamma, \sigma, \alpha_u)$.
At the same time, we have that $\Gamma : (\Pi_r)(\tilde{x}).[\tilde{a} := \tilde{E}']P' \xrightarrow{\Gamma' \triangleright \Pi'(\tilde{v}) = f(?\beta')} \Gamma[\tilde{a} := [\![\tilde{E}'[\tilde{v}/\tilde{x}]]\!]_\Gamma] : P'[\tilde{v}/\tilde{x}] = \Gamma_1 : P_1$ by the application of rule RECV. The basis for this assertion is that since m satisfies the receiving predicate α_r, a corresponding message $(\Gamma', \Pi', \tilde{v})$ shown in the label of the above transition also satisfies the receiving predicate Π_r. In addition, by using the same argument in the previous subcase of a), we have that $tr(\Gamma_1) = \gamma_1 = \gamma$.
Consider the process $P_1 = P[\tilde{v}/\tilde{x}]$, by Definition 5.2 $\langle \gamma_1; \{\epsilon, b_1\}; con'\rangle \simeq_e \langle \gamma'; \{\sigma, b\}; con'\rangle$. Thus, if we take $\xi' = \langle \bullet; con'\rangle$, then $\xi'[tr(\Gamma_1 : P_1)] = \langle \gamma'; \{\sigma, b\}; con'\rangle \simeq_e \mathcal{C}'$.
Next, we consider two possibilities of choice, among input and output prefixes.

c) **Case of** $P = \Sigma_{j \in J}(\Pi_{r_j})(\tilde{x_j}).[\tilde{a}_j := \tilde{E}_j]P_j$. Encapsulating the translation $tr(\Gamma : P)$ in ξ yields the component $\mathcal{C} = \langle \gamma; \{\epsilon, ci(\overline{\{\alpha_j, b_j\}}); con(c, \ldots)\rangle$, where $\gamma = tr(\Gamma)$ and $b_j = tr(P'_j)\ \forall j \in J$.
This component can either accept or discard a message. We consider the case where it actually consumes a message since the other case is trivial. If $\mathcal{C}$ accepts some message $m = (\gamma_s, \pi_s, v)$ via the application of rule C-crecv$_j$, we have the following transition $\mathcal{C} \overset{?\beta'}{\rightsquigarrow}{}^{*} \langle \gamma'; \{\sigma, b_k\}; con(c+1, \ldots)\rangle$ where $\sigma = \alpha_{k_x}\ v$, $\gamma' = \mathbf{upd}(\gamma, \sigma, \alpha_{k_u})$, $?\beta' = recv(c, m)$ with (α_k, b_k) the k^{th} branch of $ci(\overline{\alpha_j, b_j})$ that consumes m.
Then the *AbC* component $\Gamma : P$ can also perform an input action on some branch P'_k with the same index k by applying the rule SUM$_j$. From here we can proceed similarly as in case b)

d) **Case of** $P = \Sigma_{j \in J}(\Pi_r)@(\tilde{E}_j).[\tilde{a}_j := \tilde{E}'_j]P_j$. This case is resolved similar to case c) and when it comes to picking one branch to consider, we proceed similarly as in case a).

e) **Case of** P = $P_1|P_2$. We have that $\mathcal{C} = \mathcal{E}[tr(\Gamma : P)] = \langle \gamma; \{\epsilon, pa(b_1, b_2)\}; con\rangle \overset{\tau}{\rightsquigarrow} \langle \gamma; \{\epsilon, b_1\}\ \{\epsilon, b_2\}; con\rangle = \mathcal{C}'$ via rule C-par, where $b_i = tr(P_i)$ with $i \in \{1, 2\}$.
We know that the β' transition that $\mathcal{C}$ exposes is performed by one of the two processes in $\mathcal{C}'$. Hence, there are two cases to consider. For each case, we apply induction hypothesis on a shorter derivation of the considered process $\{\epsilon, b_i\}$ to “match” it with the transition of the *AbC* process P_i. Because the other two corresponding parallel processes stay unchanged in this derivation, it is straightforward to obtain the claim.
Case of P = K or P = 0. For the first case, we proceed with the process structure in the definition K. For the second case, the proof is straightforward since the only possible derivation corresponds to rules for discarding message, i.e., ones expose the label $:\beta'$.
Case of $P = \langle \Pi_a \rangle P'$. The proof of this case depends on the proof for P'. Assume P' is not an awareness process (otherwise we connect its awareness predicate with Π_a to obtain the desired form), we thus proceed with the structure of P' and distribute the awareness predicate to each of the subcases. There are two scenarios. If Π_a evaluates to true, all the previous considerations hold. If Π_a evaluates to false, the only possible derivation corresponds to rules that discarding messages and hence is trivial to prove. □

Proof of Lemma 5.2. *Given an* AbC *system S, if $S \rightarrow S'$, then $\mathcal{G} = \mathbb{K}[tr(S)] \Rightarrow \mathcal{G}'$. Moreover, given $S = C \parallel S_1$, if $C \xmapsto{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C'$ and $S_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S'_1$ and $\mathbb{K}[tr(S)] \overset{!\beta}{\leadsto\!\!\Rightarrow}{}^{*} \Rightarrow \mathcal{G}'$, then $\mathcal{G}' \simeq C' \parallel S'_1$, where $f(!\beta) = \Gamma \triangleright \overline{\Pi}(\tilde{v})$.*

Proof. Consider an *AbC* system S, since S can perform a transition we know that there at least one component in S that can send a message. Therefore, assume that there are $\#s \geq 1$ sending components, then there exists one component with some index i is actually responsible for initiating the synchronization. That is, $S \xrightarrow{\Gamma_i \triangleright \overline{\Pi_i}(\tilde{v}_i)} C_i' \parallel S''$, where $\Gamma_i, \Pi_i, \tilde{v}_i$ are C_i's attribute environment, sending predicate and its communicated values, respectively.

Consider the corresponding *ABEL* system obtained from S, i.e., $\mathcal{G} = \mathbb{K}[tr(S)]$.

Since originally there are $\#s$ components that can output in S, there are also $\#s$ corresponding components that can output in $\mathcal{G}$. These components will ask for message timestamps by using either rule C-out or C-cout according to the *ABEL* component semantics. By Lemma 4.1, any such components will obtain a message timestamp. Since message timestamps are unique, there exists one component, say k that obtains the smallest timestamp λ among them.

Therefore, in the sequence $\Rightarrow$, $\mathcal{G}$ must evolve with the first transition whose label corresponds to $\mathcal{C}_k$'s sending label, i.e., some $!\beta$ while the rest of $\Rightarrow$ contains accepting or discarding labels based on β, performed by other different components in $\mathcal{G}$. More specifically, the evolution has the shape $\mathcal{G} \overset{!\beta}{\rightsquigarrow}{}^* \overset{\bar{l}}{\Rightarrow} \mathcal{G}'$ with $\bar{l}$ the sequence capturing the accepting or discarding labels mentioned above. Obviously, $\mathcal{G}'$ is a consistent system because all the counter values of the components get increased by 1.

For the second part of the theorem, note that when $f(!\beta) = \Pi_i \triangleright \overline{\Pi_i}(\tilde{v}_i)$ then $k = i$. We can now use the same argument as in the proof of Theorem 5.1 to show that the components of *ABEL* and *AbC* systems are pair-wise correspondent, thereby concluding that $\mathcal{G}' \simeq S'$. □

References

[1] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: the SCEL language, ACM Trans. Auton. Adapt. Syst. 9 (2) (2014) 7:1–7:29.

[2] Y.A. Alrahman, R. De Nicola, M. Loreti, F. Tiezzi, R. Vigo, A calculus for attribute-based communication, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, ACM, 2015, pp. 1840–1845.

[3] S. Anderson, N. Bredeche, A. Eiben, G. Kampis, M. van Steen, Adaptive Collective Systems: Herding Black Sheep, BookSprints for ICT Research, 2013.

[4] K.V. Prasad, A calculus of broadcasting systems, Sci. Comput. Program. 25 (2–3) (1995) 285–327.

[5] Y.A. Alrahman, R.D. Nicola, M. Loreti, On the power of attribute-based communication, in: FORTE, in: Lecture Notes in Computer Science, vol. 9688, Springer, 2016, pp. 1–18.

[6] Y.A. Alrahman, R. De Nicola, M. Loreti, A calculus for collective-adaptive systems and its behavioural theory, Inf. Comput. 268 (2019).

[7] Y. Abd Alrahman, R. De Nicola, M. Loreti, Programming of CAS systems by relying on attribute-based communication, in: ISoLA, in: Lecture Notes in Computer Science, vol. 9952, 2016, pp. 539–553.

[8] Y. Abd Alrahman, R. De Nicola, G. Garbi, GoAt: attribute-based interaction in Google Go, in: ISoLA, in: Lecture Notes in Computer Science, vol. 11246, Springer, 2018, pp. 288–303.

[9] R. De Nicola, T. Duong, O. Inverso, Verifying AbC specifications via emulation, in: ISoLA, in: Lecture Notes in Computer Science, vol. 12477, Springer, 2020, pp. 261–279.

[10] Y. Abd Alrahman, R. De Nicola, M. Loreti, Programming interactions in collective adaptive systems by relying on attribute-based communication, Sci. Comput. Program. 192 (2020) 102428.

[11] M. Loreti, J. Hillston, Modelling and analysis of collective adaptive systems with CARMA and its tools, in: SFM, in: Lecture Notes in Computer Science, vol. 9700, Springer, 2016, pp. 83–119.

[12] V. Ciancia, D. Latella, M. Massink, On-the-fly mean-field model-checking for attribute-based coordination, in: COORDINATION, in: Lecture Notes in Computer Science, vol. 9686, Springer, 2016, pp. 67–83.

[13] R. De Nicola, L. Di Stefano, O. Inverso, Multi-agent systems with virtual stigmergy, Sci. Comput. Program. 187 (2020) 102345.

[14] R. De Nicola, T. Duong, O. Inverso, F. Mazzanti, Verifying properties of systems relying on attribute-based communication, in: ModelEd, TestEd, TrustEd, in: LNCS, vol. 10500, Springer, 2017, pp. 169–190.

[15] Y.A. Alrahman, R. De Nicola, M. Loreti, Programming of CAS systems by relying on attribute-based communication, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2016, pp. 539–553.

[16] R.D. Nicola, T. Duong, O. Inverso, C. Trubiani, AErlang: empowering Erlang with attribute-based communication, in: COORDINATION, in: Lecture Notes in Computer Science, vol. 10319, Springer, 2017, pp. 21–39.

[17] C. Ene, T. Muntean, Expressiveness of point-to-point versus broadcast communications, in: FCT, in: Lecture Notes in Computer Science, vol. 1684, Springer, 1999, pp. 258–268.

[18] Y.A. Alrahman, R.D. Nicola, G. Garbi, M. Loreti, A distributed coordination infrastructure for attribute-based interaction, in: FORTE, in: Lecture Notes in Computer Science, vol. 10854, Springer, 2018, pp. 1–20.

[19] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, ACM Comput. Surv. 36 (4) (2004) 372–421.

[20] J. Armstrong, Making reliable distributed systems in the presence of software errors, Ph.D. dissertation, The Royal Institute of Technology, Stockholm, 2003.

[21] L.-Å. Fredlund, H. Svensson, McErlang: a model checker for a distributed functional programming language, ACM SIGPLAN Not. 42 (9) (2007) 125–136.

[22] H. Svensson, L.-A. Fredlund, A more accurate semantics for distributed Erlang, in: Erlang Workshop, 2007, pp. 43–54.

[23] H. Svensson, L.-Å. Fredlund, C. Benac Earle, A unified semantics for future Erlang, in: Proceedings of the 9th ACM SIGPLAN workshop on Erlang, ACM, 2010, pp. 23–32.

[24] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, A core Erlang semantics for declarative debugging, J. Log. Algebraic Methods Program. 107 (2019) 1–37.

[25] I. Lanese, N. Nishida, A. Palacios, G. Vidal, A theory of reversibility for Erlang, J. Log. Algebraic Methods Program. 100 (2018) 71–97.

[26] M. Logan, E. Merritt, R. Carlsson, Erlang and OTP in Action, Manning Publications Co., 2010.

[27] R.D. Nicola, T. Duong, M. Loreti, ABEL - a domain specific framework for programming with attribute-based communication, in: COORDINATION, in: Lecture Notes in Computer Science, vol. 11533, Springer, 2019, pp. 111–128.

[28] R. De Nicola, T. Duong, O. Inverso, F. Mazzanti, A systematic approach to programming and verifying attribute-based communication systems, in: From Software Engineering to Formal Methods and Tools, and Back, in: Lecture Notes in Computer Science, vol. 11865, Springer, 2019, pp. 377–396.

[29] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitutions, J. Funct. Program. 1 (4) (1991) 375–416.

[30] G.L. Ferrari, U. Montanari, P. Quaglia, A pi-calculus with explicit substitutions, Theor. Comput. Sci. 168 (1) (1996) 53–103.

[31] K.V.S. Prasad, Programming with broadcasts, in: CONCUR, in: Lecture Notes in Computer Science, vol. 715, Springer, 1993, pp. 173–187.

[32] G.A. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, J. Funct. Program. 7 (1) (1997) 1–72.

[33] E.B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: a core language for abstract behavioral specification, in: FMCO, in: Lecture Notes in Computer Science, vol. 6957, Springer, 2010, pp. 142–164.
[34] G. Salvaneschi, C. Ghezzi, M. Pradella, ContextErlang: a language for distributed context-aware self-adaptive applications, Sci. Comput. Program. 102 (2015) 20–43.
[35] G.D. Plotkin, A structural approach to operational semantics, Tech. Rep. Computer Science Department, Aarhus University, 1981.
[36] N. Nishida, A. Palacios, G. Vidal, A reversible semantics for Erlang, in: International Symposium on Logic-Based Program Synthesis and Transformation, Springer, 2016, pp. 259–274.
[37] A. Pironti, R. Sisto, Provably correct Java implementations of Spi Calculus security protocols specifications, Comput. Secur. 29 (3) (2010) 302–314.
[38] N. Bezirgiannis, F.S. de Boer, E.B. Johnsen, K.I. Pun, S.L.T. Tarifa, Implementing SOS with active objects: a case study of a multicore memory system, in: FASE, in: Lecture Notes in Computer Science, vol. 11424, Springer, 2019, pp. 332–350.
[39] I.A. Mason, C.L. Talcott, Actor languages their syntax, semantics, translation, and equivalence, Theor. Comput. Sci. 220 (2) (1999) 409–467.
[40] J. Eckhardt, T. Mühlbauer, J. Meseguer, M. Wirsing, Semantics, distributed implementation, and formal analysis of KLAIM models in Maude, Sci. Comput. Program. 99 (2015) 24–74.
[41] jRESP: Java Runtime Environment for SCEL Programs, http://jresp.sourceforge.net/.
[42] R. Hennicker, A. Klarl, Foundations for ensemble modeling – the Helena approach, in: Specification, Algebra, and Software, Springer, 2014, pp. 359–381.
[43] T. Bures, I. Gerostathopoulos, P. Hnetynka, F. Plasil, F. Krijt, J. Vinárek, J. Kofron, A language and framework for dynamic component ensembles in smart systems, Int. J. Softw. Tools Technol. Transf. 22 (4) (2020) 497–509.
[44] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECO: an ensemble-based component system, in: CBSE, ACM, 2013, pp. 81–90.
[45] R. De Nicola, G. Ferrari, M. Loreti, R. Pugliese, A language-based approach to autonomic computing, in: International Symposium on Formal Methods for Components and Objects, Springer, 2011, pp. 25–48.
[46] R. Milner, Communication and Concurrency, PHI Series in Computer Science, Prentice Hall, 1989.