ABEL - A Domain Specific Framework for Programming with Attribute-Based Communication

Rocco De Nicola, Tan Duong, Michele Loreti

ABEL - A domain specific framework for programming with attribute-based communication

Rocco De Nicola[1], Tan Duong[2], and Michele Loreti[3]

[1] IMT - School for Advanced Studies, Lucca, Italy
rocco.denicola@imtlucca.it
[2] Gran Sasso Science Institute, L'Aquila, Italy
tan.duong@gssi.it
[3] University of Camerino, Camerino, Italy
michele.loreti@unicam.it

Abstract. Attribute-based communication is a promising paradigm for modelling and programming complex interactions in open distributed systems such as collective adaptive systems (CAS). This new paradigm has been formalized in *AbC*, a kernel calculus with a minimal set of primitives that can be used to model formally verifiable CAS. The calculus assumes an underlying coordination infrastructure that has to guarantee the wanted communication and leaves open the actual implementation of the way communication partners are selected. The proposed implementations of messages exchange for *AbC* are either not in full agreement with the original semantics or do miss detailed performance evaluations. In this paper, we continue the search for efficient implementations of *AbC* and present *ABEL* - a domain specific framework that offers programming constructs with a direct correspondence to those of *AbC*. We use *Erlang* to implement *ABEL* inter- and intra-components interaction that together faithfully model *AbC* semantics and enable us to verify properties of *ABEL* program. We also consider a number of case studies and, by experimenting with them, show that it is possible to preserve *AbC* semantics while guaranteeing good performance. We also argue that even better performances can be achieved if the "strong" *AbC* requirement on the total order of message delivery is relaxed.

Keywords: attribute-based communication, process calculi, distributed programming, Erlang

1 Introduction

Attribute-based communication, originally proposed in [12] is a novel paradigm that allows the dynamic selection of communication groups while taking into account run-time properties and status of interacting entities. At its core, the paradigm relies on a pair of communication primitives. The command $send(v)@\pi$ is used to send a tuple of values v to all components satisfying the predicate π.

The command $receive(x)@\pi'$ is used to receive a tuple of values on x with contents satisfying the predicate π'. The interaction predicates are also parametrised with local attributes and when their values change, the interaction groups do implicitly change, allowing opportunistic interactions.

This paradigm was formalized in the *AbC* kernel calculus [3, 5] to study the impact of attribute-based communication in the realm of CAS [7]. In *AbC*, components are equipped with a set of attributes describing their features, which can change at runtime. Component interactions is driven by conditions over their states to enable anonymity, adaptivity and open-endedness. The expressive power of *AbC* in terms of representing different communication paradigms, such as point-to-point, group based, channel-based, and broadcast-based models has been demonstrated in [5]. *AbC* communication model follows broadcast in the style of [16]; output action can take place even without the presence of any receivers while input action instead waits to synchronize with available messages.

The original semantics of *AbC* has been formulated in a way that when a component sends a message, that message is delivered to all components in the system in a single move and each individual receiver decides whether to use the message or to discard it. This semantics implies a restriction on the ordering of message delivery because only one component can send its message at a time. That is, message delivery in *AbC* is performed according to a *total order* [8].

Some proposals have already been put forward to efficiently implement message exchange for *AbC*. It has been implemented in Java [4], Google Go [1] and Erlang [11]. However, these implementations are either not in full agreement with the original semantics or do miss detailed performance evaluations. This may give rise to doubts about efficiency and correctness and thus prevent the adoption of attribute-based communication.

Indeed, there are a number of challenges to face when providing an implementation fully respecting the exact semantics of *AbC*. The first is posed by the fact that its message passing model requires guaranteeing a total order on message delivery. While various protocols for total order broadcast have been proposed in the literature, see, e.g., [9, 15], the anonymity and open-endedness features of *AbC* makes then unsuitable in this context because components cannot contribute to establishing an order. A closely related work in building total order for *AbC* has recently been presented in [2] where a sequencer-based protocol is formalized and proved correct on different topologies of networks. However, there is still the possibility for the proposed protocol to give rise to unexpected behaviours due to the complex behaviour of *AbC* components. Moreover, since *AbC* components contain parallel processes operating independently on a shared, dynamic changing attribute environment, to simulate the right interleaving semantics among processes, any implementation should carefully coordinate processes, otherwise situations may arise where processes interfere without exhibiting the wanted behaviour.

In this paper, we continue the search for efficient implementations of *AbC* and present *ABEL* - a programming framework for systems whose elements interact according to the *AbC* style. Our framework provides an execution environment

for *AbC* specifications, that supports *AbC* programming abstractions for writing and running programs, and, at the same time, fully preserves the original semantics of *AbC*. More concretely:

1. We provide a set of attribute-based programming constructs, implemented as an *Erlang* library that allows easily defining component programs.
2. We implement coordination mechanisms for intra- and inter-components interactions in *Erlang*, each dealing with one of the parallel operators of *AbC* and together guaranteeing the original semantics.
3. We demonstrate with experiments that better performances can be achieved if the "strong" *AbC* requirement on the total order of message delivery is relaxed.

We show that by starting from a given *AbC* specification, it is straightforward to derive a corresponding *Erlang* program containing API calls. We also show that the close correspondence between *AbC* and *ABEL* enables us to reason about the execution code by using verification tools developed for *AbC* [10]. Moreover, by experimenting with a number of case studies, we show that it is possible to preserve *AbC* semantics while guaranteeing good performance and that by, slightly relaxing the ordering requirements, better ones can be obtained.

The rest of the paper is organized as follows. In Section 2 we briefly review the *AbC* calculus, and provide examples illustrating its programming paradigm. In Section 3, we present the API support for *AbC*. In Section 4, we describe the coordination mechanisms used to handle intra- and inter-components interactions. Section 5 reports our experiments on case studies, taking into account different message ordering strategies. Section 6 concludes the paper by discussing the difference between our work and previous proposals, together with some conclusions and hints to future works.

2 Programming with *AbC*

The *AbC* calculus provides concrete primitives that permit the construction of formally verifiable models of CASs according to the attribute-based communication paradigm. A system is rendered as a collection of interacting components. A component C is either a process P associated with an attribute environment Γ and an interface I, or the parallel composition of two components.

$$\text{(Components)} \qquad C ::= \Gamma :_I P \mid C_1 \parallel C_2$$

The environment Γ is a partial mapping from attribute names to values, representing the component state. The interface $I \subseteq Dom(\Gamma)$ contains a set of names, exposed for interaction purpose. The process P can be either an inactive process 0, a prefixing process $\alpha.P$, an update process U, an awareness process $\langle \Pi \rangle P$, a choice process $P_1 + P_2$, a parallel process $P_1|P_2$, or a process call K (with a unique definition $K \triangleq P$).

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

AbC prefixing actions exploit run-time attributes and predicates over them to determine the internal behaviour of components and the communication partners.

$(\tilde{E})@\Pi$ is an output action that evaluates expressions $\tilde{E}$ under the local environment and sends the result to those components whose attributes satisfy predicate Π;

$\Pi(\tilde{x})$ is an input action that binds to the variables $\tilde{x}$ the message received from any component whose attributes and the communicated values satisfy the receiving predicate Π;

$[a := E]$ is an update operation that assigns to attribute a the evaluation of expression E under the local environment;

$\langle \Pi \rangle$ blocks the following process until Π is satisfied under the local environment.

Attribute updates and awareness predicates are local to components and their executions are atomic with the associated communication action.

An expression E may be a constant value v, a variable x, an attribute name a, or a reference $this.a$ to attribute a in the local environment. Predicate Π can be either tt, or can be built using comparison operators $\bowtie$ between two expressions and logical connectives $\wedge$, $\neg$, ... Both expressions and predicates can take more complex forms, of which we deliberately omit the precise syntax; we just refer to them as n-ary operators on subexpressions, i.e., $f(\tilde{E})$ and $p(\tilde{E})$.

The original semantics of *AbC* has been formulated in a way that when a component sends a message, this is delivered in a single move to all components in the system. Atomically, each individual receiver decides whether to keep the message or to discard it. This semantics imposes a restriction on the ordering of message delivery because only one component at a time can send a message. That is, message delivery in the original *AbC* is performed according to a total order [8].

Even if this approach is useful to describe in an abstract way the *AbC* one-to-many interactions, it may be considered too strong when large scaled distributed systems are considered. To relax the *total ordering* formulation of the original *AbC* semantics, we have extended *AbC* syntax to explicitly model the *infrastructure* responsible of message dispatching already used in [2].

To this goal we introduce the new category of *servers*. *AbC* systems are now built by using *servers* of the form $\{\cdot\}^{\iota,\omega}$ that are responsible for managing a set of *components*. Each server is equipped with an *input queue* ι and an *output queue* ω. The former is the queue of messages, coming from the environment, that the server must deliver to the managed components. The latter is the queue of messages that have been generated locally and that the server must forward

to other components. Each message m is a triple of the form $(\Gamma, \pi, \tilde{v})$ where Γ is the environment of the sending component, π is the target predicate used to select the receivers, and $\tilde{v}$ is the tuple of sent values. In what follows we will use q to denote a queue of messages, and [] to denote the empty queue. Moreover, we will use $m :: q$ (resp. $q :: m$) to represent an extended queue obtained by adding message m at the beginning (resp. at the end) of q.

The syntax of AbC servers is the following:

$$\begin{array}{rr} \text{(Servers)} & S ::= \{M\}^{\iota,\omega} \\ \text{(Managed Elements)} & M ::= C \mid S \parallel S \end{array}$$

A server thus equips its managed element (which in turn can be either single AbC component or recursively other servers) with input and output queues. Servers communicate with one another by adding and withdrawing messages from their queues, thereby relaxing the original synchronous semantics.

The operational semantics of AbC systems is defined via the transition relation $\rightarrow \subseteq \mathsf{Sys} \times \mathsf{Lab} \times \mathsf{Sys}$, where Lab is the set of transition labels λ defined by the following syntax:

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

The transition label $\Gamma \triangleright \overline{\Pi}(\tilde{v})$ represents an output of $\tilde{v}$ executed by a component with environment Γ that is sent to receivers satisfying Π. Input actions are represented by label $\Gamma \triangleright \Pi(\tilde{v})$ that represents the capability of a system to receive message $\tilde{v}$ sent by a component with environment Γ to receivers satisfying Π. Finally, τ represents internal/silent actions.

We have fully formalized the new semantics that relies on the two queues of the servers. Due to lack of space, in Table 1 we only report some the rules of the operational semantics that we consider more relevant.

$$\frac{S_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S_1' \quad S_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S_2'}{S_1 \parallel S_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S_1' \parallel S_2'} \textsc{Sync}$$

$$\frac{S_1 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} S_1' \quad S_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S_2'}{S_1 \parallel S_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} S_1' \parallel S_2'} \textsc{ComL} \qquad \frac{S_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} S_1' \quad S_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} S_2'}{S_1 \parallel S_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} S_1' \parallel S_2'} \textsc{ComR}$$

$$\frac{m = (\Gamma, \tilde{v}, \Pi) \quad M_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} M_2}{\{M_1\}^{m::\iota,\omega} \xrightarrow{\tau} \{M_2\}^{\iota,\omega}} \textsc{SerDIn} \qquad \frac{M_1 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v}) M_2}}{\{M_1\}^{\iota,\omega} \xrightarrow{\tau} \{M_2\}^{\iota,\omega::(\Gamma,\tilde{v},\Pi)}} \textsc{SerDOut}$$

$$\frac{}{\{M_1\}^{\iota,\omega} \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} \{M_2\}^{\iota::(\Gamma,\tilde{v},\Pi),\omega}} \textsc{SerIn} \qquad \frac{}{\{M_1\}^{\iota,(\Gamma,\tilde{v},\Pi)::\omega} \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} \{M_2\}^{\iota,\omega}} \textsc{SerOut}$$

Table 1. Operational Semantics: Relevant Rules

The rules for parallel composition ($\|$) are the expected ones: SYNC states that S_1 and S_2, when in parallel, can both receive the same message; COML and COMR state that messages sent by S_1 (resp. S_2) are received by S_2 (resp. S_1) and the result of the synchronisation is an *output label* to allow other components in parallel to receive the same message.

The rules describing behaviour of a *server* are SERDIN, SERDOUT, SERIN and SEROUT. The first two rules model the interaction of the server with the enclosed element, while the last two describe the interaction of the server with the enclosing environment. Rule SERDIN states that the first message is consumed from the input queue when the managed element M_1 receives it. Rule SERDOUT states that a message is added at the end of the output queue whenever the managed element M_1 executes an output action. A message is added at the end of the input queue whenever a new message is received from the enclosing environment (rule SERIN). Conversely, a server sends a message by removing it from the front of output queue (rule SEROUT).

2.1 *AbC* at work

In order to illustrate *AbC* primitives and assess our implementation we now provide a couple of simple case studies that we will later use also for assessing performances of our implementations.

Stable matching. We consider a variant of the well-known stable matching problem (SMP) [13] that can be naturally expressed in terms of partners' attributes [10]. Solving an SMP problem amounts to finding a stable matching between two equally-sized sets of elements, after each element has specified an ordering of preferences over all members of the opposite set. The original algorithm [13] goes through a sequence of proposals initiated by members of one group, say M, according to their preference lists. Members of the other group, say W, after receiving a proposal, do choose the best candidate between their current partner and the one making advances. The algorithm guarantees the existence of a unique set of pairs, which is stable, i.e. there is no pair of unmatched elements, that prefer each other to their current partners. An *AbC* specification for this algorithm can be found in [4].

Our variant allows participating agents to express their interests in potential partners by relying on partner's attributes rather than on their identities. In this scenario, M components start by proposing to those W components that satisfy their highest requirements, i.e., a predicate that specifies all wanted attributes. If an M agent cannot find any partner for a given demand, it retries after weakening the predicate by dropping one of the wanted attributes. W components are reactive to proposals and perform "select and swap" partners as before. However, since a proposal may target multiple partners, the protocol needs extra acknowledgement messages between agents of two types to select the partner. And, an agent of type M relaxes a predicate Π only if all potential partners, addressed by Π, rejected it.

In AbC the two types of components M and W are modelled as follows:

$$M_i \triangleq \Gamma_{mi} :_{\{id,a_1,a_2,\ldots\}} P_M \quad \text{and} \quad W_j \triangleq \Gamma_{wj} :_{\{id,b_1,b_2,\ldots\}} P_W$$

where the interfaces expose the names of attributes that represent the features of the components and P_M and P_W their actual behaviour.

Below, we specify part of the system concerned components of type M, whose behaviour combines 4 processes

$$P_M \triangleq Q \mid P \mid A \mid R$$

An agent m that is looking for partners satisfying predicate Π[4] has first to learn about the number of potential partners, say `c`, satisfying this predicate. This is taken care by process Q that first broadcasts a query message and then collects interested replies (not detailed here). For each attempt, m keeps track of the set `bl` of partners which have rejected it. As long as there are available partners ($c > |bl|$), process P sends a 'propose' message containing the agent's characteristics (denoted by the sequence $\widetilde{msg}$) to predicate Π, excluding those in `bl`.

$$\begin{aligned} P \triangleq\ & \langle partner = 0 \wedge c > |bl| \wedge send = 1 \rangle \\ & (\text{`}propose\text{'}, this.id, \widetilde{msg})@(\Pi \wedge \text{id} \notin \text{this.bl}).[\text{send} := 0]P + \ldots \end{aligned}$$

Other branches of P have a similar structure and will be used for proposing with other requirements. Process A, reported below, handles multiple 'yes' messages which may arrive in parallel. The continuation H chooses the sender (bound to y) of the first message to match, confirms to y that it has been selected and updates new `partner`. A 'toolate' message is sent to senders of subsequent 'yes' messages:

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

The sent proposal may also be rejected. Process R collects messages of this type. The arrivals of 'no' and 'split' requests cause the addition of the senders to the set `bl` and enabling `send`. A 'split' message which originates from some matched partner resets the current `partner` and opens the possibility for process P to become active and retry.

$$\begin{aligned} R \triangleq\ & (x = \text{`}split\text{'})(x, y).[bl := bl \cup \{y\}, send := 1, partner := 0]R \\ & + (x = \text{`}no\text{'})(x, y).[bl := bl \cup \{y\}, send := 1]R \end{aligned}$$

[4] As an example, consider a user interested in finding a server that has some specific resources. A possible communication predicate would be $\Pi = (cores = this.pcores \wedge mem = this.pmem)$ where $cores$ and mem are two attributes of servers, and $pcores$ and $pmem$ are two attributes of users.

Graph colouring. We now consider a distributed graph colouring problem where vertices exchange messages with their neighbours to collaborate on deciding a colour (in our case a positive number) for each of them in such a way that adjacent vertices do not get the same colour. The following AbC specification is adapted from [6]. A graph is modelled naturally as a set of components $V_i \triangleq \Gamma_i :_{\{id,nbr\}} P_V$, one for each vertex, with attribute `id` representing a unique identifier, and `nbr` representing a set of neighbours ids. Vertices operate in rounds and in each round they use a predicate $(this.id \in nbr)$ to send messages to neighbours. Vertices concurrently execute the four processes *F, T, D, A* until they get assigned a 'definitive' colour.

Every non `assigned` vertex selects the first available colour which has not been used by his neighbours, that is $min\{i \notin \text{this.}\texttt{used}\}$. A try message of the form $(\text{'}try\text{'}, c, r)$ is sent to inform others that the sending vertex wants to attain colour c at round r.

$$\begin{aligned} F \triangleq\ & (\text{'}try\text{'}, \min\{\text{i} \notin \text{this.used}\}, this.r)@(this.id \in nbr). \\ & [\text{colour} := \min\{\text{i} \notin \text{used}\}, \text{counter} := \text{counter} + 1]0 \end{aligned}$$

Each vertex counts the number of 'try' messages (including its own) using the attribute `counter`. Process T collects 'try' messages from neighbours where it records colours proposed from neighbours with greater ids in a set `constraints` to avoid conflict. Other branches of T (not shown here) deal with 'try' messages from neighbours operating in one round ahead, i.e., $(this.r < z)$ for which the relevant information are kept in attributes `counter1`, `constraints1`.

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

After collecting all 'try' messages, i.e., $(counter = |nbr| + 1)$, each vertex checks whether the colour it proposed among neighbours is valid, encoded in process A below. If this is the case[5], the vertex sends a 'done' message of the form $(\text{'}done\text{'}, c, r)$ to indicate that c has been taken at round r, setting `assigned` to true and terminates. If the proposed colour leads to conflict, the vertex starts a new round by sending a new 'try' message. At this point, the vertex has learnt about neighbours and thus tries to take the best decision by selecting a new colour excluding also `constraints`, i.e., $min\{i \notin \text{this.}\texttt{used} \cup \text{this.}\texttt{constraints}\}$. During this new round, $r + 1$, the vertex does not count messages from those vertices who might have sent a 'try' message (collected by process T above via `counter1`) and from 'done' neighbours (collected by process D presented next).

$$\begin{aligned} A \triangleq\ & \langle(\text{counter} = |\text{nbr}| + 1) \wedge \text{colour} \notin \text{constraints} \cup \text{used}\rangle \\ & (\text{'}done\text{'}, \text{this.colour}, \text{this.r})@(\text{this.id} \in \text{nbr}).[\text{assigned} := tt]0 \\ +\ & \langle(\text{counter} = |\text{nbr}| + 1) \wedge \text{colour} \in \text{constraints} \cup \text{used}\rangle \\ & (\text{'}try\text{'}, min\{i \notin this.used \cup this.constraints\}, this.r + 1)@(\text{this.id} \in \text{nbr}). \\ & [\text{r} := \text{r} + 1, \text{counter} := \text{done} + \text{counter1} + 1, \text{constraints} := \text{constraints1}, \ldots]A \end{aligned}$$

[5] It happens for those vertices whose ids are greatest among the unassigned neighbors

Each vertex collects 'done' messages to update the set of `used` colours, and counts 'done' neighbours in `done`. In addition, if the messages come from the previous round, i.e., $(this.r > z)$, the vertex treats them as 'try' messages in the current round, and thus increments the `counter`.

$$\begin{aligned} D \triangleq\ & (x = \text{`}done\text{'} \wedge \text{this.r} = z)(x, y, z). \\ & [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{y\}]D \\ +\ & (x = \text{`}done\text{'} \wedge \text{this.r} > z)(x, y, z). \\ & [\text{done} := \text{done} + 1, \text{used} := \text{used} \cup \{\text{y}\}, \text{counter} := \text{counter} + 1]D \end{aligned}$$

3 Programming support for AbC

Our framework aims at providing a direct mapping from *AbC* specifications to executable programs in *Erlang*, allowing experimentations with attribute-based communication with little of efforts. An *ABEL* program is based on a sequence of behaviour definitions for processes, and top-level commands for starting all components. The running program is a set of concurrently executing components.

Creating components
A component is set up in two steps: it is first created and then assigned an initial behaviour. To create a new component, an attribute environment *Env* and an interface I are provided to **new_component**(*Env*,I), which returns a unique address C. The command **start_beh**(C, $[BRef]$) starts the execution of a component C with an initial behaviour specified as a list of behaviour references, whose actual definitions have been previously declared. The term `[elem]` is used to indicate a list of type `elem`.

$$\begin{aligned} \mathsf{C} = \ & \textbf{new\_component}(Env, I), \\ & \textbf{start\_beh}(\mathsf{C}, [BRef]) \end{aligned}$$

In the above commands, environment *Env* is represented as a map whose keys are atoms denoting attribute names. Interface I is a tuple of atoms denoting public attributes, to which other components may want to use their values. In particular, when sending a message, any component attaches also the portion of the environment *Env'* corresponding to *Env* but limited by the interface.

Behaviour Definition The syntax for behaviour definition is given in Fig. 1. Elements wrapped by $\langle\rangle$ are optional. A definition $BDef$ is a function that has as first parameter a component address C. The body of a definition is a sequence of commands, which also require C as their first parameters.

Action *Act* gives the descriptions for *AbC* input and output actions. There we use m to denote message, u to denote update and g, s, r to denote awareness, sending and receiving predicates respectively. We now briefly explain the different commands.

Prefix - takes as a parameter a pair containing a prefixing action *Act* and a continuation $BRef$. The command executes *Act* and continues with $BRef$. If *Act* is an output description, the command evaluates m into $\tilde{v}$, the sending

$$
\begin{array}{rll}
BDef & ::= & beh\_name(\mathsf{C}, \langle param\ list \rangle) \rightarrow Com. \\
BRef & ::= & \textbf{fun}(\langle param\ list \rangle) \rightarrow Beh(\mathsf{C}, \langle param\ list \rangle)\ \textbf{end} \\
Act & ::= & \{\langle g \rangle, m, s, \langle u \rangle\} & \text{Output} \\
 & & \{\langle g \rangle, r, \langle u \rangle\} & \text{Input} \\
Com & ::= & \textbf{prefix}(\mathsf{C}, \{Act, BRef\}) & \textbf{Prefix} \\
 & & \textbf{choice}(\mathsf{C}, [\{Act, BRef\}]) & \textbf{Choice} \\
 & & \textbf{parallel}(\mathsf{C}, [BRef]) & \textbf{Parallel} \\
 & & Beh(\mathsf{C}, \langle param\ list \rangle) & \textbf{Call}
\end{array}
$$

Fig. 1. ABEL API for process definitions.

predicate s into s' and broadcast the triple (Env', s', $\tilde{v}$) to all components and possibly performs an attribute update u, whenever guard g (if specified) is satisfied. If Act is an input description; the command returns a message and optionally performs an attribute update u, whenever guard g (if specified) is satisfied and the sender's attributes and the communicated message satisfy the receiving predicate r.

Choice - takes as parameter a list of pairs, each providing a description of the prefixing action Act and a continuation in form of $BRef$. This command executes one of the actions and continues with the behaviour associated to that action.

Parallel - This command dynamically creates parallel processes, each of which executes a behaviour indicated by a reference in the parameter list.

Process Call - executes the behaviour Beh.

The basic elements m, g, r, s, u are represented as follows.

Message. A message m to be sent is represented as a tuple. A message element can be a function parameterized with the sender environment, i.e., $fun(S) \rightarrow \ldots end$, for making it possible to refer to attribute values in S.

Predicates. An awareness predicate g is a unary function parameterized with the environment of the executing component, i.e., $fun(E) \rightarrow \ldots end$. A sending predicate s is a binary function parameterized with the sender and receiver environments in that order, i.e., $fun(S, R) \rightarrow \ldots end$. A receiving predicate r is a ternary function parameterized with the environments of the receiver and the sender, and a communicated message in that order, i.e., $fun(R, S, M) \rightarrow \ldots end$.

Attribute Update. An update is represented as a list of pairs; in each pair, the first element is an attribute name and the second is an expression to be used for the update. The expression can be a function parameterized with the local environment (to use also attribute values), and with the communicated message, when in case the update is associated with a receive action.

As an example of using the presented API for deriving execution code from AbC specifications, Fig. 2 presents the definitions in $ABEL$ for processes F and D in the graph coloring scenario (Example 2). Bold-faces are used to highlight

the structural correspondence between the *ABEL* code and the *AbC* one. The rest of the code is used to specify predicates, messages and updates, which can be addressed by automatic translation. Indeed, we have developed the translator from *AbC* to *ABEL* and made it available with the *ABEL* implementation.

```
f(C) →
    M = {'try', fun(S) → min_colour(att(used,S)) end, fun(S) → att(round,S) end},
    P = fun(S,R) → sets:is_element(att(id,S),att(nbr, R)) end,
    U = [{counter, fun(S) → att(counter,S) + 1 end},
         {colour, fun(S) → min_colour(att(used,S)) end}],
    Act = {M,P,U},
    prefix(C,{Act,nil}).

d(C) →
    P1 = fun(R,S,M) → size(M) == 3 andalso element(1,M) == 'done'
                andalso att(round,S) == element(3,M)
         end,
    U1 = [{done, fun(R,M) → att(done,R) + 1 end},
         {used, fun(R,M) → sets:add_element(element(2,M),att(used,R)) end}],
    Act1 = {P1, U1},
    P2 = fun(R,S,M) → size(M) == 3 andalso element(1,M) == 'done'
                andalso att(round,S) > element(3,M)
         end,
    U2 = [{done, fun(R,M) → att(done,R) + 1 end},
         {used, fun(R,M) → sets:add_element(element(2,M),att(used,R)) end},
         {counter,fun(R,M) → att(counter,R) + 1 end}],
    Act2 = {P2, U2},
    DRef = fun() → d(C) end,
    choice(C,[{Act1,DRef},{Act2,DRef}]).
```

Fig. 2. Example code derived from the *AbC* processes F and D in Graph Colouring

4 Coordinating components

In this section, we consider two alternative implementations for coordinating *AbC* components. The first implementation is obtained by using an infrastructure similar to the one proposed in [2]; minor modifications have been introduced to make the total ordering protocol more robust. The second one is obtained from the former by relaxing some requirements on ordering preservation. For both implementations we have a corresponding formal semantics. The first one exactly captures the original total ordering *AbC* semantics first presented in [3] while the second one is that briefly explained in Sect. 2.

An *AbC* system in *ABEL* consists of a set of components and an infrastructure with 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. Fig. 4 shows an example of a tree-structure where each node (black) is responsible for a group of components (white). The model assumes that a node only communicates with those connected to it; likewise a component only communicates with the node they are assigned to. In what follows, we describe the implementations with respect to the behaviour of components and infrastructure nodes.

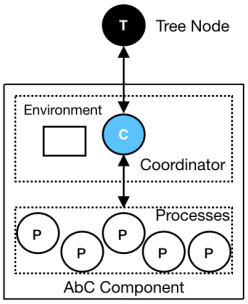

Fig. 3. An *AbC* component in ABEL

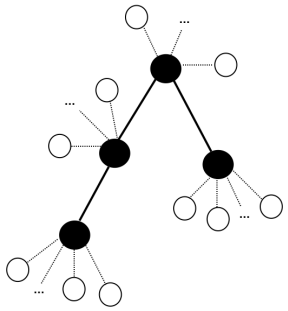

Fig. 4. A tree-based infrastructure

Coordinating processes An *AbC* component is an autonomous entity with multiple processes operating on a shared environment. The behaviour of a component is that of its processes. We have that (i) If more than one process offers an output action, then *only one* of them will be allowed (ii) If more than one process can actually input a message, again *only one* of them will succeed. (iii) A component discards a message only if *all* of its processes discard that message. (iv) Processes and environment influence each other, e.g., a change in the environment caused by one process may enable or disable other processes.

This semantics suggests us considering the attribute environment as a reactive process, rather than a static store. Fig. 3 pictures the internal structure of an *AbC* component. Processes P represent *AbC* processes that communicate with a coordinator C by message passing via the API presented in Sect. 3. Each process submits one action at a time, and continues only after receiving an acknowledgment message. The coordinator keeps track of component environment and decides the actual actions to be executed. The execution of an action may require updating the environment if the action has an associated update request. Actions that cannot be executed because of guards are stored and retried when the environment is updated.

To model interleaving, a coordinator dynamically keeps track of the number of processes, of the set of submitted actions, and uses an input queue for storing messages forwarded from infrastructure. Events that are handled by a coordinator includes commands from processes, messages forwarded from the infrastructure and other implicit information. We briefly explain the operations performed for each event as below.

Parallel events The coordinator creates new processes, and updates the total number of processes.

Sending events When the guard (if specified) is satisfied, the message is forwarded to the infrastructure and the sending process is acknowledged, otherwise, the sending action is added to the set of submitted ones.

Receiving events The action is added to the set of submitted actions.

Choice events The coordinator handles them like normal sending and receiving events, the acknowledgement message to the choice process is a continuation behaviour.

Delivery events These events are internally generated when the input queue is not empty and the number of submitted actions is equal to the number of processes. Predicates of input actions are checked against messages extracted from the queue, one by one until there is a match or the queue is empty. In case of matching, the message is delivered to the receiving process.

Retry events These events are internally triggered when the environment changes. Output actions and choices among output actions in the submitted set are retried.

Coordination Strategies As mentioned above we have implemented two different strategies for message exchange that may lead to different ordering of message delivery, we call them synchronous and asynchronous.

Synchronous. Encapsulated in the infrastructure presented in [2] there is a sequencer-based protocol that guarantees a total order of message delivery for *AbC* components. The infrastructure, apart from broadcasting messages for components, plays the role of a sequencer that allocates unique ids for components messages. When a component is willing to send a message, it requests a fresh id for the message. A component can deliver a message labeled with an id only if it has delivered all messages with $id' < id$. This means that messages are delivered in the order of consecutive messages' ids, which is total.

This protocol is used to coordinate *ABEL* components by relying on a tree-based infrastructure as follows. Every component coordinator keeps a counter c, initially set to 1.

To handle sending events, the coordinator, after checking the guard, requests a fresh id if his previous one has already been used. The output action can take place only if the fresh id matches the counter, otherwise it is postponed. If the action is executed, c is incremented by 1.

To handle delivery events, the coordinator sends a message extracted from the input queue if its id is equals to c, and increases the counter. If the message is not consumed by any input action, the procedure repeats until the queue is empty or there is no message with the expected id.

To handle retry events, the coordinator may send an empty message if there is an unused fresh id which is equal to the local counter and if all components processes can not send a message.

When a non-root tree node receives an id request, it forwards the request to its parent. The root replies its counter value for each request and increments the counter. This fresh id is forwarded along the same path of the original request, but in a reverse order. Eventually, the node which initiated the request receives the fresh id and sends it to the requesting component. When a tree node receives a data message, it forwards the messages to the other connected nodes and to connected *AbC* components, except the sender. In addition, to guarantee that the number of messages exchanged is bound, each node has an input queue for

storing incoming data messages and only forwards a message if its id equals to the node's counter.

Asynchronous. In this implementation, components can send the actual messages simultaneously via the infrastructure without asking and checking for fresh ids. We rely on the tree structure as for the synchronous case but we do not take advantage of it and could have used another structure. A component delivers messages from its input queue in any order to its processes, while trying to filter out as many 'uninteresting' messages as possible. In this case, tree nodes forward messages as soon as they receive them and the root has no special role; which makes the implementation simpler.

5 Experiments

In this section, we report on the performance evaluation of our *Erlang* prototype[6] of *ABEL* by considering the two case studies, stable marriage and graph colouring, whose *AbC* specifications have been sketched in Sect. 2. These *AbC* specifications were model checked by following the approach presented in [10].In particular, we have verified the termination and soundness properties for the graph colouring problem, and the completeness, symmetry of matchings and orchestration properties for stable matching problem. The explicit-state model checker [14] helped us to verify early designs of these case studies and to come up with correct specifications. From the verified specifications we have then derived the *ABEL* programs introduced in Sect. 3.

For our experiments, we used a workstation with a dual Intel Xeon processor E5-2687W (16 cores in total) and 128 GB of memory. The OS version is Linux 4.9.95-gentoo and the Erlang/OTP version is 21.2. In addition, the coordination infrastructure is the tree-based one with a varying number of nodes and the the previously outlined communication strategies. The tree-based infrastructure was chosen as the default topology after considering the result in [2] showing that it guarantees better performance over others.

5.1 Stable Marriage with Attribute

The input to this case study is randomly generated assuming some predefined probabilities of attributes and preferences. First, we define their ranges, then we associate a probability to each value in the range so that the sum of the probabilities is 1. In this way, an attribute (or preference) can take a concrete value v with a probability $p(v)$. In the experiment, we consider 2 attributes and 2 preferences with ranges of 2 values. We select 10 different combinations of probabilities, consider problem sizes of 100 and 200 pairs of elements, and generate 100 instances for each problem size.

We ran the *ABEL* program to solve problem instances and took the average of the execution times. We have also checked the completeness and stability

[6] https://github.com/ArBITRALABEL

conditions on the outcomes of the program. Table 2 presents the numbers under two different ordering strategies. These numbers are the average over 500 runs with a tree of 7 nodes. The result shows that the interaction protocol for this case study performs faster when using asynchronous messaging.

Odering	**Execution Times** (in sec.)	
	Size = 100	Size = 200
Total Order	5.65	54.63
No Order	3.39	35.56

Table 2. Results of stable matching

5.2 Graph Colouring

We conducted some experiments with several DIMACS graphs collected from various public sources: flat300_28_0.col (300 vertices and 21695 edges), dsjc500.1.col (500 vertices and 12458 edges), will199GPIA.col (701 vertices and 7065 edges) and dsjc1000.1.col (1000 vertices and 49629 edges). The datasets chosen provides an increasing number of vertices which are considered as *AbC* components.

The following metrics are considered: the running time in seconds, the total number of messages exchanged between components and the infrastructure and the total size of messages in MB. When measuring, vertices do not wait for each other to report the completion of their colouring: as soon as a vertex decides on a colour, it reports that colour, the number of messages exchanged (and message size) to an external process. Tables 3 and 4 show the results of graph colouring with total ordering and relaxed ordering, respectively. S is the number of nodes used by the tree structure. The other columns show the numbers for colour, round, messages and the total size in that order. The execution times are computed as the average of 50 runs, the other numbers report the average over the results of different number of nodes. Overall, the *ABEL* code can perform

Graph	**Execution Times** (in sec.)				**#C**	**#R**	**#Msg** (in milli.)	**Size** (in MB)
	S = 3	S = 7	S = 15	S = 31				
flat300_28_0	4.57	4.22	4.19	4.4	46	44	1.5	7,141
dsjc500.1	4.22	3.65	3.34	3.39	20	19	2.15	4,008
will199GPIA	7.46	5.95	5.56	5.53	11	25	4.62	4,255
dsjc1000.1.col	32.02	27.19	25.94	24.97	32	30	12.5	42,324

Table 3. Results of graph colouring using total order

colouring for experiment graphs without any conflicts, and resulted in small speedups when increasing the number of nodes. This is however more obvious with the larger graph. On the other hand, its performance varies on different graphs. This might have to do with their specific topologies. In general, in graphs with more edges, components are more likely to face colour conflicts among

Graph	**Execution Times** (in sec.) S = 3	S = 7	S = 15	S = 31	**#C**	**#R**	**#Msg** (in milli.)	**Size** (in MB)
flat300_28_0	3.76	4.69	4.44	4.53	46	45	1.5	7,095
dsjc500.1	3.17	3.32	2.22	2.94	20	19	2.12	3,905
will199GPIA	6.53	4.03	3.02	2.81	11	25	4.5	4,070
dsjc1000.1.col	49.62	33.58	28.69	21.04	32	30	12.4	41,524

Table 4. Results of graph colouring using relaxed order

neighbours and thus to require more interactions. This leads to an increased number of rounds and message exchanges.

It can be seen from the results that both ordering strategies return similar outcomes for the same input graphs. Although, in most cases, relaxed ordering guarantees slightly better performance.

6 Concluding Remarks and Related Works

We have presented *ABEL*, an implementation of *AbC* in *Erlang* that builds on an API that mimics *AbC* constructs. Our purpose is to develop and experiment with systems featuring complex interactions according to the *AbC* paradigm. The API is integrated seamlessly with underneath coordination mechanisms that together simulate the original synchronous semantics of *AbC* and a less demanding one. Because of the direct correspondence between the two formal semantics and our actual implementations, we can perform formal verification of *ABEL* programs by considering their *AbC* abstractions. Indeed, from *AbC* specifications we can obtain verifiable models that can be provided as input to model checkers.

There have been other attempts at providing implementations of *AbC*. *AErlang* [11] extends *Erlang* processes to allow attribute-based communication beside the point-to-point one. However, the programming style is based on the host language and it might not be immediate to derive *AErlang* programs from *AbC* specifications. Furthermore, the lack of a corresponding formal semantics of *AErlang* calls for alternative directions, not based on translations, when it comes to reasoning on programs. Other *AbC* implementations such as [1, 4] exhibit a gap between *AbC* primitives and their programming constructs. More efforts are needed to derive *AbC* code and and automatic translations are not immediate. We refer the readers to [11] for a more detailed account of related works on concurrent languages and communication models.

In near future, we want to establish a formal relationship between *ABEL* and *AbC* and prove the correctness of the API implementation, as well as the correctness of the developed translation. This would require studying the operational semantics of the inter- and intra- components coordinators, and formalizing the translation rules. We also plan to integrate the model checking part within our framework. Moreover, it might be useful to equip *ABEL* with other communication and synchronization abstractions; we have experienced that programming complex distributed systems using only *AbC* send and receive may be difficult.

References

1. Alrahman, Y.A., De Nicola, R., Garbi, G.: Goat: Attribute-based interaction in google go. In: International Symposium on Leveraging Applications of Formal Methods. pp. 288–303. Springer (2018)
2. Alrahman, Y.A., De Nicola, R., Garbi, G., Loreti, M.: A distributed coordination infrastructure for attribute-based interaction. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 1–20. Springer (2018)
3. Alrahman, Y.A., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Proc. of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE). pp. 1–18 (2016)
4. Alrahman, Y.A., De Nicola, R., Loreti, M.: Programming of cas systems by relying on attribute-based communication. In: International Symposium on Leveraging Applications of Formal Methods. pp. 539–553. Springer (2016)
5. Alrahman, Y.A., De Nicola, R., Loreti, M.: A behavioural theory for interactions in collective-adaptive systems. CoRR **abs/1711.09762** (2017), http://arxiv.org/abs/1711.09762
6. Alrahman, Y.A., De Nicola, R., Loreti, M.: Programming the interactions of collective adaptive systems by relying on attribute-based communication. CoRR **abs/1711.06092** (2017), http://arxiv.org/abs/1711.06092
7. Anderson, S., Bredeche, N., Eiben, A., Kampis, G., van Steen, M.: Adaptive collective systems: herding black sheep. BookSprints for ICT Research (2013)
8. Baldoni, R., Cimmino, S., Marchetti, C.: Total order communications: A practical analysis. In: European Dependable Computing Conference. pp. 38–54. Springer (2005)
9. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. **9**(3), 272–314 (Aug 1991). https://doi.org/10.1145/128738.128742, http://doi.acm.org/10.1145/128738.128742
10. De Nicola, R., Duong, T., Inverso, O., Mazzanti, F.: Verifying properties of systems relying on attribute-based communication. In: ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma. pp. 169–190 (2017)
11. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: AErlang: empowering Erlang with attribute-based communication. Science of Computer Programming (2018)
12. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **9**(2), 7:1–7:29 (2014)
13. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly **69**(1), 9–15 (1962)
14. H.ter Beek, M., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming **76**(2), 119–135 (2011)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
16. Prasad, K.V.: A calculus of broadcasting systems. Science of Computer Programming **25**(2-3), 285–327 (1995)