Programming of CAS systems by relying on attribute-based communication*

Yehia Abd Alrahman[1], Rocco De Nicola[1], and Michele Loreti[2]

[1] IMT School for Advanced Studies Lucca, Italy
[2] Università degli Studi di Firenze, Italy

Abstract. In most distributed systems, named connections (i.e., channels) are used as means for programming interaction between communicating partners. These kinds of connections are low level and usually totally independent of the knowledge, the status, the capabilities, ..., in one word, of the attributes of the interacting partners. We have recently introduced a calculus, called AbC, in which interactions among agents are dynamically established by taking into account "connection" as determined by predicates over agent attributes. In this paper, we present $Ab^{a}C$uS, a Java run-time environment that has been developed to support modeling and programming of collective adaptive systems by relying on the communication primitives of the AbC calculus. Systems are described as sets of parallel components, each component is equipped with a set of attributes and communications among components take place in an implicit multicast fashion. By means of a number of examples, we also show how opportunistic behaviors, achieved by run-time attribute updates, can be exploited to express different communication and interaction patterns and to program challenging case studies.

1 Introduction

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

Groups or collectives are dynamically formed at the time of interaction by means of available/interested receiving components that satisfy sender predicates.

* This research has been partially supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and by the Italian project PRIN 2010LHT4KM CINA.

In this way run-time attribute updates introduce opportunistic interactions between components. Indeed, interaction predicates can be parametrized with respect to local attribute values and when these values change, the interaction groups or collectives do implicitly change. This makes modeling and programming adaptation quite natural.

Programming opportunistic behavior in classical communication paradigms like channel-based communication, e.g., $b\pi$-calculus [13], is challenging. Components should agree on specific names or channels to interact. Channels have no connection with the component attributes, characteristics or knowledge. They are specified as addresses where the exchange should happen. These names/channels are quite static and changing them locally at run-time requires explicit communication and intensive use of name restriction which affect program readability and compositionally.

As an example, consider the behavior of a component that inspects its sensor and based on the input message communicates with its peer components. This behavior can be modeled in $b\pi$-calculus as follows:

$$C \triangleq \nu b(\bar{b}a \mid b(c).\bar{c}d) \xrightarrow{\tau} \nu b\bar{a}d$$

where component C communicates with its sensor along a private channel "b" and uses the received channel "a" to communicate with its peers. Clearly, this intensive use of scoping and explicit communication for modeling a simple read operation hinders readability and compositionality of large models. In many cases, one is only interested in how components interact with each other and abstracts from local interactions. In this case we would like to define C as:

$$C \triangleq \overline{\mathcal{E}(c)}d$$

where $\mathcal{E}(c)$ is a function that returns a channel name based on the message received from the sensor. This would permit abstracting from local interactions and concentrating on components interactions while taking into account the environment/space in which they are operating. This intuition is captured by relying on attribute-based communication where we assume that components have local views of their own status and of their surrounding environment and their behaviors are parametrized with respect to these views. Formally a component is defined as $\Gamma{:}P$ where Γ is an abstraction of its local view and of its environment and P is its behavior. In fact, we generalize more and replace function $\mathcal{E}(\bullet)$ with a predicate that considers elements of Γ. Send and receive operations then rely on predicates, i.e., $(d)@\Pi$ rather than on names i.e., $\bar{c}d$.

The attribute-based system is more than just the parallel composition of interacting partners; it is also is parametric with respect to the shared environment or space where system components are executed. The shared environment has a great impact on how components behave. It introduces a new way of indirect communication, where components mutually influence each other unintentionally. For instance, in the ant foraging system [19], when an ant disposes pheromone in the shared space to keep track of her way back home, she influences other ants

behavior as they are programmed to follow traces of pheromone with higher concentration. In this way, the ant unintentionally influences the behavior of the other ants by only modifying the shared space. This type of indirect communication cannot be easily modeled even by relying on asynchronous communication [18] where messages are placed with the intention that other addressed components will receive them at some point of time.

In this paper we present $A^{a}bC$uS, a Java run-time environment for the AbC calculus. AbC is the first calculus that was designed to focus on a minimal set of primitives that permits attribute-based communication. AbC was first proposed in [6] and a new stable and refined version was released in [5]. The latest version is accompanied with a formal labeled semantics and a behavioral theory used to establish results about the relations with other existing approaches. $A^{a}bC$uS was developed by building on the formal semantics of the latest version of AbC. Given the generality and flexibility of the interaction primitives of the AbC calculus, $A^{a}bC$uS was developed for programming modern software systems where adaptation, reconfiguration and collaboration are key issues. We would like to use it to assess the practical impact of this new communication model on challenging case studies like the ones from the realm of collective-adaptive systems (CAS) [15] and to fully understand both its merits and limits.

The rest of the paper is organized as follows. In Section 2 we review the AbC calculus and its expressive power through a running example. In Section 3 we present $A^{a}bC$uS, a run-time environment for the AbC calculus and in Section 4 we present a case study about a smart conference application that we have implemented in $A^{a}bC$uS. In Section 5 we discuss related work and finally in Section 6 we draw some conclusions and sketch a plan for future work.

2 The *AbC* Calculus

In this section we briefly review the AbC calculus and its specific features and we illustrate how well-known interaction patterns can be naturally modeled in AbC. To help the reader appreciate AbC features, we proceed by a simple running example. We consider the classical stable marriage problem (SMP) [16], a problem of finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element.

In our example, we consider n men and n women, where each person has ranked all members of the opposite sex in order of preferences, we have to engage the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable. For convenience we assume there are no ties; thus, if a person is indifferent between two or more possible partners he/she is nevertheless required to rank them in some order. We will use this example to gently introduce a subset of the AbC calculus and its informal semantics throughout this section. The presentation is intended to be intuitive and interested readers are referred to [5] for full details concerning the full AbC syntax and formal semantics.

The top-level entities of the AbC calculus are *components* (C), a component is either a process P associated with an *attribute environment* Γ (denoted by $\Gamma{:}P$) or the parallel composition $C_1 \| C_2$ of components. The *attribute environment* Γ is a partial map from attribute identifiers $a \in \mathcal{A}$ to values $v \in \mathcal{V}$. Values could be numbers, names (string), tuples, etc.

$$C ::= \Gamma{:}P \quad | \quad C_1 \| C_2 \quad | \quad \ldots$$

Example (step 1/3): The marriage scenario can be modeled in AbC as follows:

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

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

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

The behavior of an AbC process can be generated by the following grammar:

$$P ::= \mathsf{0} \quad | \quad \alpha.P \quad | \quad [\tilde{a} := \tilde{E}]P \quad | \quad \langle \Pi \rangle P \quad | \quad P_1 + P_2 \quad | \quad P_1 | P_2 \quad | \quad A(\tilde{x})$$

- $\mathsf{0}$: denotes the inactive process;
- $\alpha.P$: denotes an action prefixed process, a process that executes action α and continues as P;
- $[\tilde{a} := \tilde{E}]P$: denotes an attribute update process, a process that behaves as P given that its attribute environment is updated by setting the value of each attribute in the sequence $\tilde{a}$ to the evaluation of the corresponding expression in the sequence $\tilde{E}$. The attribute updates and the first move of P are atomic;
- $\langle \Pi \rangle P$: denotes an awareness process, a process that tests awareness data about a component status or its environment by inspecting the local attribute environment where the process resides. It blocks the execution of process P until the predicate Π becomes true;
- the processes $P_1 + P_2$, $P_1 | P_2$, and $A(\tilde{x})$ are standard for nondeterminism, parallel composition, and parametrized process definition respectively. It should be noted that the parallel operator "|" does not allow communication between P_1 and P_2, they can only interleave while the parallel operator "‖" at the component level allows communication between components.

Example (step 2/3): The structures of process M, specifying the behavior of a man, and the process W, specifying the behavior of a woman, are defined as follows:

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

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

A man, M, picks his first best from the ranking list "$\texttt{this}.preferences$" and assumes it to be his partner. This element is removed from his preferences. In the same transition he proposes to this possible partner by executing action a (to be specified later) and then continues as M' . The prefix this is a reference to the value assigned to the attribute identifier "$preferences$". Functions $\texttt{Top}(\texttt{arg})$ and $\ominus(\texttt{arg})$ both take a list as an argument. The former returns the first element of the list if the list is not empty and the empty string otherwise, while the latter returns the list resulting from the removal of its first element.

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

The AbC communication actions can be generated by the following grammar:

$$\alpha ::= (\tilde{E})@\Pi \quad \mid \quad \Pi(\tilde{x})$$

- $(\tilde{E})@\Pi$: denotes an attribute-based output action, it evaluates the sequence of expressions $\tilde{E}$ under the local attribute environment Γ and then sends the result to the components whose attributes satisfy the predicate Π. If Π semantically equals to a logic "false", the message is not exposed and the action is used to represent a silent move;
- $\Pi(\tilde{x})$: denotes an attribute-based input action, it binds to sequence $\tilde{x}$ the corresponding received values from components whose *communicated attributes* or values satisfy the predicate Π.

Example (step 3/3): In the previous step, if we further specify the action "a" and the process M' in M, the action "b" and the processes W_1 and W_2 in W, the behavior of a man and a woman becomes:

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

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

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

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

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

Classical interaction patterns in *AbC*. In [5] we have shown how the classical group-based [4,11] and publish/subscribe-based [7,14] interaction patterns can be naturally translated into *AbC*. We have also shown how to translate channel-based communication like in $b\pi$-calculus [13] into *AbC*. The interested readers are referred to the website on [1]; in this website we discuss the direct implementation of these translations into *AbC* linguistic primitives. Below, we briefly introduce the basic idea behind such translations.

To select partners in *channel-based communication*, structured messages are used where the name of the channel is rendered as the first element in the message; receivers only accept messages with attached channels that match their receiving channels. Attributes do not play any role in such interaction so we assume components with empty environments i.e., $\Gamma = \emptyset$. Thus a pair of processes, one willing to receive on channel a and the other willing to send on the same channel, can be modeled as follows:

$$\emptyset : (x = a)(x, y).P \parallel \emptyset : (a, msg)@(\mathsf{tt}).Q$$

Group names are rendered as attributes when translating *group-based interaction* as shown below:

$$\Gamma_1 : (msg)@(group = a).P \parallel \Gamma_2 : (\mathsf{tt})(x).Q \quad \text{where} \quad \Gamma_2(group) = a$$

The operations for joining or leaving a given group are rendered as attribute updates.

The *publish and subscribe* paradigm can be seen as special cases of the attribute-based one; a natural modeling of the topic-based publish/subscribe model [14] into *AbC* can be obtained by allowing publishers to broadcast messages with "tt" predicates (i.e., satisfied by all) and making sure that only subscribers can check the compatibility of the exposed publishers attributes with their subscriptions:

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

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

3 $Ab^{a}C$uS: A Run-time Environment for the *AbC* Calculus

In this section we present $Ab^{a}C$uS [1], a Java run-time environment for supporting the communication primitives of the *AbC* calculus. We also show how one can exploit these flexible primitives to provide a general programming framework that encompasses different communication frameworks and interaction patterns. Having a run-time environment allows us to assess the practical impact of this young communication paradigm in real applications. In fact, we plan to use the new programming framework to program challenging case studies, dealing with collective adaptive systems, from different application domains.

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

AbC's operational semantics abstracts from a specific communication infrastructure. An *AbC* model consists of a set of parallel components that cooperate in a highly dynamic environment where the underlying communication infrastructure can change dynamically. The current implementation $Ab^{a}C$uS is however a centralized one, in the sense that it relies on a message broker that mediates the interactions. In essence, the broker accepts messages from sending components, and delivers them to all registered components with the exception of the sending

ones. This central component plays the role of a forwarder and does not contribute in any way to message filtering. The decision about accepting or ignoring a message is taken when the message is delivered to the receiving components.

We would like to stress that, although the current $Ab^{a}C$uS implementation is centralized, components interact anonymously and combine their behaviors to achieve the required goals. Components are unaware of the existence of each other, they only interact with the message broker. To facilitate interoperability with other tools and programming frameworks, $Ab^{a}C$uS relies on JSON [3], a standard data exchange technology that simplifies the interactions between heterogenous network components and provides the basis for allowing $Ab^{a}C$uS programs to cooperate with external services or devices.

The advantages of this programming framework can be summarized by saying that it provides a small set of programming constructs that naturally supports adaptation and guarantees a high degree of scalability in distributed settings by allowing anonymous interaction. The new programming framework also has a direct correspondence with an existing formal model with clear and understood semantics and with a sound foundational theory which lays the basis for formal reasoning and verification.

In what follows we summarize the main $Ab^{a}C$uS programming constructs, their implementation, and their relations with the *AbC* primitives. We consider the implementation of one of the *man* components introduced in the running example of the previous section.

Components. *AbC* components are implemented via the class `AbCComponent`. Instances of this class are executed in either virtual or physical machines that provide access to input/output devices and network connections. An instance of the class `AbCComponent` contains an attribute environment and a set of processes that represents the behavior of the component. Components interact via ports supporting either local communication, i.e., components run in the same application, or external communication, i.e., components run in different applications or different machines. The following $Ab^{a}C$uS code shows how to create a *man* component m_1, to assign the process `ManAgent()` to it, and finally to start its execution.

```
1   AbCComponent m1 = new AbCComponent("M_1");
2   m1.addProcess(new ManAgent());
3   m1.start();
```

Attribute Environments. *AbC* attribute environments are implemented via the class `AbCEnvironment`. An instance of the class `AbCEnvironment` contains a set of attribute identifiers, implemented via the class `Attribute`, and another set to store their values. The attribute environment maintains the attribute values by providing read and update operations via the methods `getValue(attribute)` and `setValue(attribute,value)` respectively. The class `Attribute` implements an *AbC* attribute and ensures type compatibility of the assigned values. The

following $Ab^{a}C$uS code shows how to create an AbC attribute and to read and update its value.

```
Attribute<Integer> idAttribute = new Attribute<>("ID", Integer.class);
getValue(idAttribute);
setValue(idAttribute, 1);
```

Processes. The generic behavior of an AbC process is implemented via the abstract class `AbCProcess`. The AbC communication actions $(\tilde{E})@\Pi$ and $\Pi(\tilde{x})$ are implemented via the methods `Send(predicate, values)` and `receive(msg->Function(msg))` respectively. The receive operation accepts a message and passes it to a boolean function that checks if it satisfies the receiving predicate. The attribute updates are implemented via the method `setValue(attribute,value)` while the awareness operator and the process definition are implemented via the methods `waitUntil(predicate)` and `call(process)` respectively. The method `exec(p)` spawns a new process `p` and runs it in parallel with the executing process while the recursive call is implemented via a native Java `while` loop. The non-deterministic choice of several input actions possibly preceded by attribute updates and/or awareness operators is implemented in $Ab^{a}C$uS by overloading the receive method `receive(`$in_1, \ldots, in_n$`)`. This method takes a finite number of arguments of type `InputAction`, each of which has the following form $[\tilde{a} := \tilde{E}]\langle\Pi\rangle\Pi(\tilde{x})$. When a message arrives to the component this method only enables the correct branch or blocks the execution in case of unwanted messages. The non-deterministic choice between an input and output actions is not allowed because output actions are non-blocking. It should be noted that the method `addProcess(p)` for a component `c` has the same effect of running the process `p` in parallel with the already existing processes in component `c`. The generic behavior of a process is defined via the abstract method `doRun()` that is invoked when the process is executed. The programmer should implement this method to specify the behavior of the process. The following $Ab^{a}C$uS code implements a *man* behavior; the one-to-one correspondence with the AbC process M in Section 2 is evident.

```
public class ManAgent extends AbCProcess {
  public ManAgent() {
  super("ManAgent");
}
@Override
protected void doRun() throws Exception {
    while ( !Definition.preferences.isEmpty() ) {
      Integer partner = Definition.preferences.poll();
      setValue(Definition.partnerAttribute, partner);
      send( new HasValue("ID", partner) , new Tuple( "PROPOSE" ,
           getValue(Definition.idAttribute)) );
      receive(o -> isAnInvalidationMessage(o) );
    }
  }
...
```

Network Infrastructure. In $Ab^{a}C$uS each component is equipped with a set of ports for interacting with other components as mentioned above. A port is identified

by an address that can be used to manage the interaction with a mediator or a message broker. It should be noted that these ports are not meant to be used by components to identify the addresses of the other components but rather as a way to communicate with the message broker. The components remain anonymous to each other and are not allowed to communicate directly; they only know the message broker. The latter can be seen as an access point which mediates the interaction between components. It serves as a forwarder that shepherds the interaction, but it has nothing to do with message filtering.

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

Currently, two concrete implementations are supported in *AbC*uS: VirtualPort and AbCClient. The VirtualPort implements a port where interactions are performed via a buffer stored in the main memory. A VirtualPort is used to run components in a single application without relying on a specific network infrastructure. AbCClient instead assumes the presence of a server that mediates the interactions between components. The following *AbC*uS code shows how to create an `AbCServer` and start its execution.

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

The following code shows how to create a client port, register it to an existing server, assign the port for the man component m_1, and finally starts its execution.

```
1      AbCClient client = new AbCClient(InetAddress.getLoopbackAddress(), 1234);
2      client.register( InetAddress.getLoopbackAddress() , DEFAULT_SUBSCRIBE_PORT );
3      m1.setPort(client);
4      client.start();
```

It should be noted that the *AbC* server and the client ports usually operate from different machines or networks.

Interested readers are referred to [1] for a detailed description of the implementation, source code, and also a few demos.

4 Case Study: A Smart Conference System

In this section we show how to use the programming constructs of the *AbC* calculus to program a smart conference system in an intuitive and easy way.

The idea is to exploit the mobile devices of the participants to guide them to their locations of interest. Each participant expresses his/her topic of interest and the conference venue is responsible for guiding each participant into the location that matches their interests. The conference venue is composed of a set of rooms where the conference sessions are to be held. We assume that the name of each room identifies its location. The conference program and room

relocation can be dynamically adjusted at anytime to handle specific situations, i.e., a crowded session can be relocated into a larger room and this should be done seamlessly without disruption to the whole conference program. When relocation happens, the new updates should be communicated to the interested participants. A participant only receives updates about his/her topic of interest.

The conference venue is represented as a set of parallel AbC components, each of them representing a room $(Room_1 \| \ldots \| Room_n)$ and each room has the following form $\Gamma_i : R$. Participants instead have the following form $\Gamma_j : P$. We assume that each room has a unique name that identifies its location and each participant has a unique id. The overall system is represented as the parallel composition of the conference venue and the set of participants as shown below:

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

When a participant arrives to the conference venue, he/she selects the topic of the talk of interest and updates his/her attribute $interest$ as shown in process P below:

$$P(x) \triangleq [\texttt{this}.interest := x]Com$$

By doing so, a communication process Com, that is responsible for communicating the participant interests to the conference venue, is activated. Process Com sends a session request $Sreq$ to nearby providers (i.e., with a provider role); in our case, this is a room. The message also contains the participant topic of interest and its id. Once the message is emitted, the process blocks until a reply notification that matches his/her interest arrives. The notification contains the session name, an $interestRply$ label, and the name of the room where the session is to be held. By receiving this notification, the process updates the participant destination and activates process $Update$.

$$\begin{aligned} Com \triangleq\ & (\texttt{this}.interest,\ Sreq,\ \texttt{this}.id)@(role = Provider). \\ & (x = \texttt{this}.interest \wedge y = interestRply)(x,\ y,\ z). \\ & [\texttt{this}.dest := z]()@\texttt{ff}.Update \end{aligned}$$

The process $Update$, defined below, blocks and waits for new updates or changes in schedule for the session of interest. Precisely, it blocks until it receives an $interestUpd$ notification about a session that matches the participant interest. The notification message contains the previous session that was supposed to be held in this room, the current session, an $interestUpd$ label, and the name of the room where the session of interest has been moved. Once a notification message is received, the process updates the destination to the new location and waits for future updates.

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

On the other hand, the behavior of a room in the conference venue is modeled by process R which consists of three parallel processes:

$$R \triangleq Service \quad | \quad ReLoc \quad | \quad Swap$$

The room can provide a normal service, through the process *Service* defined below, by replying to session requests from those participants who are interested in its current session. Once a session request *Sreq* that matches the current room session is received from a participant, the room sends an *interestRply* to the requesting participant identifying them by their *id*. The reply contains the current session, an *interestRply* label, and the name of the room. The parallel composition with *Service* ensures that the room is always ready to handle concurrent requests from different participants.

$$\begin{aligned} \mathit{Service} \triangleq\ & (x = \texttt{this}.\mathit{session} \wedge y = \mathit{Sreq})(x,\ y,\ z). \\ & (\ (\texttt{this}.\mathit{session},\ \mathit{interestRply},\ \texttt{this}.\mathit{name})@(\mathit{id} = z).\mathsf{0}\ \mid\ \mathit{Service}\ ) \end{aligned}$$

On the other hand, any room might experience a change in schedule unexpectedly at run-time. The process *ReLoc*, defined below, is responsible for handling this issue in a way such that interested participants in the new session and also other rooms where a swap of schedule should happen are notified.

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

Relocation is triggered by setting the value of attribute *relocate* to true and assigning the attribute *newSession* with a new session name, the room updates its previous session to the current one and the current session to the new one. The relocation flag *relocate* is turned off by setting its value to false. The process continues by sending the updated previous and current sessions of the room accompanied with an *interestUpd* label and the room name to either participants who are waiting for updates about the updated session or to another room where a swap of schedule should happen.

Furthermore, a room can also receive update notifications from other rooms in case a swap of schedule is required, i.e., a small crowded room can switch its session with a larger one with few attendees. This message is exactly the same message that is received by the participants, the only difference is concerned with the receiving predicate. The room accept an *interestUpd* message only if the second value in the message matches its current session. By doing so, the room is made aware that a swap of sessions is required. So it performs the swap and then sends a message to interested participants to update their destination.

$$\begin{aligned} \mathit{Swap}\ \triangleq\ & (z = \mathit{interestUpd} \wedge y = \mathit{this}.\mathit{session})(x, y, z, l). \\ & [\texttt{this}.\mathit{prevSession} := \texttt{this}.\mathit{session},\ \texttt{this}.\mathit{session} := x] \\ & (\ (\texttt{this}.\mathit{prevSession},\ \texttt{this}.\mathit{session},\ \mathit{interestUpd},\ \texttt{this}.\mathit{name}) \\ & \quad @(\mathit{interest} = \texttt{this}.\mathit{session} \vee \mathit{session} = \texttt{this}.\mathit{session}).\mathsf{0} \\ & \mid\ \mathit{Swap}) \end{aligned}$$

The following $Ab\overset{a}{C}$uS code corresponds to the participant process $P(x)$. The code is self-explanatory and has a one-to-one correspondence with the specifications in process $P(x)$.

```
1  public class ParticipantAgent extends AbCProcess {
2    private String selectedTopic;
3    public ParticipantAgent(String selectedTopic) {
4      super(name);
5      this.selectedTopic = selectedTopic;
6    }
7    @Override
8    protected void doRun() throws Exception {
9      setValue(Definition.interestAttribute, this.selectedTopic);
10     send(new HasValue(Definition.ROLE_ATTRIBUTE_NAME, Definition.PROVIDER),
11         new Tuple(getValue(Definition.interestAttribute),
12             Definition.REQUEST_STRING, getValue(Definition.idAttribute)
13         ));
14     Tuple value = (Tuple) receive(o -> isAnInterestReply(o));
15     setValue(Definition.destinationAttribute, (String) value.get(2));
16     while (true) {
17       value = (Tuple) receive(o -> isAnInterestUpdate(o));
18       setValue(Definition.destinationAttribute, (String) value.get(4));
19     }
20   }
21   ...
```

Due to space limitations we do not show the code for the whole system; interested readers are referred to [1] for a description of the full implementation, GUI demos, and source code.

5 Related Work

Attribute-based communication was used in the context of autonomic computing when the SCEL language [12] was designed. The interesting results of using attribute-based interactions to program in SCEL inspired the distillation of the *AbC* calculus [5]. There was a necessity to understand the full impact of attribute-based communication in distributed programming. SCEL was designed to support programming of autonomic computing systems. Compared with SCEL, the knowledge representation in *AbC* is abstract and is not designed for detailed reasoning during the model evolution. This reflects the different objectives of SCEL and *AbC*. While SCEL focuses on programming issues, *AbC* concentrates on a minimal set of primitives to study effectiveness of attribute-based communication. Further related work can be found in [8], where a specification language was designed based on the *AbC* primitives to support quantitive analysis of large systems.

jRESP [2] is a run-time environment for the SCEL language which provides an API to permit using SCEL linguistic constructs in Java programs. $Ab^{a}C$uS inherits from jRESP its large use of design patterns and integration capabilities. This is done by allowing abstract implementation of the underlying network infrastructure and by taking advantage of JSON date exchange technology [3] to facilitate the interaction with heterogenous network components.

Programming collective and/or adaptive behavior has been studied in different research communities like context-oriented programming and component-based approach. In Context-Oriented Programming (COP) [17], a set of linguistic constructs is used to define context-dependent behavioral variations. These variations are expressed as partial definitions of modules that can be overridden

at run-time to adapt to contextual information. They can be grouped via layers to be activated or deactivated together dynamically. These layers can be also composed according to some scoping constructs. Our approach is different in that components adapt their behavior by considering the run-time changes of the values of their attributes which might be triggered by either contextual conditions or by local interaction. Another approach that considers behavioral variations by building on the Helena framework is considered in [20].

The component-based approach, represented by FRACTAL [10] and its Java implementation, JULIA [9], is an architecture-based approach that achieves adaptation by defining systems that are able to adapt their configurations to the contextual conditions. System components are allowed to manipulate their internal structure by adding, removing, or modifying connectors. However, in this approach interaction is still based on explicit connectors. In $Ab\overset{a}{C}$uS predefined connections simply do not exist, we do not assume a specific architecture or containment relations between components. The connectivity is always subject to change at any time by means of attribute updates. In our view, $Ab\overset{a}{C}$uS is definitely more adequate when highly dynamic environments have to be considered.

6 Concluding Remarks

We informally introduced the AbC calculus by considering a simple running example and we discussed the expressiveness of the calculus and compared it with different communication paradigms. We introduced $Ab\overset{a}{C}$uS, a Java run-time environment for supporting the communication mechanisms of the AbC calculus. We presented a case study about smart conference systems that represents a typical application from the realm of collective adaptive systems. In the considered system, the conference venue collaborates with the mobile devices of the participants to guide them to the talks they are interested in. The scenario shows how, in case of session relocations, a change in the schedule for a room can be managed coherently with the allocation of other rooms and with the interest of the participants. The latter need to be kept updated about any schedule changes that impacts on the locations where the topics of their interest are presented.

To program component interactions $Ab\overset{a}{C}$uS relies on a mediator or a message broker. This represents a single point of failure which puts the communication reliability at risk. Our future efforts will be dedicated to providing an efficient distributed implementation of $Ab\overset{a}{C}$uS and to further investigating the practical impact of this programming framework with more realistic case studies. We are also interested in establishing a formal relationship between $Ab\overset{a}{C}$uS and AbC by providing a formal definition for a distributed abstract machine that is operationally complete with respect to AbC itself.

References

1. $Ab\overset{a}{C}$uS: A run-time environment for the AbC calculus. `http://lazkany.github.io/AbC/`. Established: 2015-12-08.

2. Jresp: Java runtime environment for scel. `http://jresp.sourceforge.net`.
3. Json: Javascript object notation. `http://www.json.org`.
4. Gul Agha and Christian J Callsen. *ActorSpace: an open distributed programming paradigm*, volume 28. ACM, 1993.
5. Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. On the power of attribute-based communication. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP International Conference, FORTE*, pages 1–18. Springer, 2016. Full technical report can be found on `http://arxiv.org/abs/1602.05635`.
6. Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1840–1845. ACM, 2015.
7. Michael A Bass and Frank T Nguyen. Unified publish and subscribe paradigm for local and remote publishing destinations, June 11 2002. US Patent 6,405,266.
8. Luca Bortolussi, Rocco De Nicola, Vashti Galpin, Stephen Gilmore, Jane Hillston, Diego Latella, Michele Loreti, and Mieke Massink. Carma: Collective adaptive resource-sharing markovian agents. *Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2015*, pages 16–31, 2015.
9. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.
10. Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model. *Draft of specification, version*, 2(3), 2004.
11. Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing (CSUR)*, 33(4):427–469, 2001.
12. Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: the scel language. *ACM Transactions on Autonomous and Adaptive Systems*, pages 1–29, 2014.
13. Christian Ene and Traian Muntean. A broadcast-based calculus for communicating systems. In *Parallel and Distributed Processing Symposium, International*, volume 3, pages 30149b–30149b. IEEE Computer Society, 2001.
14. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.
15. Alois Ferscha. Collective adaptive systems. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pages 893–895.
16. David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
17. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.
18. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 133–147. Springer, 1991.
19. Duncan E Jackson and Francis LW Ratnieks. Communication in ants. *Current biology*, 16(15):R570–R574, 2006.
20. Annabelle Klarl. Engineering self-adaptive systems with the role-based architecture of helena. In *Infrastructure for Collaborative Enterprises, WETICE 2015, Larnaca, Cyprus, June 15-17, 2015*, pages 3–8, 2015.