# A systematic approach to programming and verifying attribute-based communication systems

Rocco De Nicola[1], Tan Duong[2], Omar Inverso[2], and Franco Mazzanti[3]

[1] IMT School for Advanced Studies Lucca, Italy
rocco.denicola@imtlucca.it
[2] Gran Sasso Science Institute, Italy
{tan.duong,omar.inverso}@gssi.it
[3] ISTI CNR, Pisa Italy
franco.mazzanti@isti.cnr.it

**Abstract.** A methodology is presented for the systematic development of systems of many components, that interact by relying on predicates over attributes that they themselves mutually expose. The starting point is a novel process calculus *AbC* (for Attribute-based Communication) introduced for modelling collective-adaptive systems. It is shown how to refine the model by introducing a translator from *AbC* into UML-like state machines that can be analyzed by UMC. In order to execute the specification, another translator is introduced that maps *AbC* terms into *ABEL*, a domain-specific framework that offers faithful *AbC*-style programming constructs built on top of *Erlang*. It is also shown how the proposed methodology can be used to assess relevant properties of systems and to automatically obtain an executable program for a non-trivial case study.

## 1 Introduction

Collaboration between Stefania and the first author of this paper dates back to the late eighties, when they were both working at CNR and interested in devising formal methods to provide models and techniques to guarantee correctness of concurrent systems. Together with other colleagues, they developed a model checker [1] for proving logical properties of concurrent systems using ACTL [2], an action-based version of the branching time logic CTL. Stefania has since continued to work on developing variants of ACTL and tailored model checkers for new classes of systems she has been confronted with. Stefania and the last author of this paper, together with other colleagues, developed the FMC model checker [3] adopting as models and logic, respectively, transition systems and a version of ACTL extended with fixed-point operators; they then developed the UMC [4] model checker, where the model was instead directly inspired by UML statecharts and the logic, UCTL, could express properties of both actions and states.

In this paper we build on Stefania's work and show how UMC can be used to prove properties of so-called collective-adaptive systems (CAS). These are

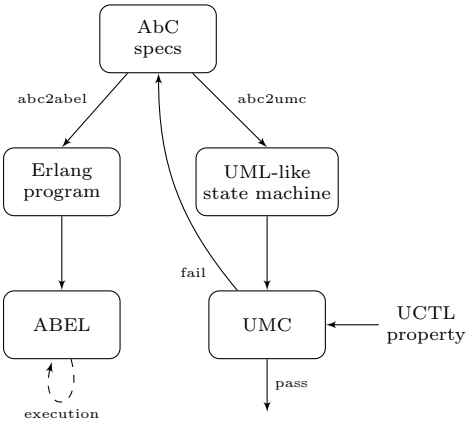
systems formed by anonymous components that can dynamically join and leave, and adapt their behaviour to the environment in pursuit of an individual or collective goal.

To describe CAS, we rely on a specifically conceived process calculus, *AbC* [5,6] whose distinguishing features are the interaction primitives, based on the concept of *attribute-based communication*, which permits groups of partners to communicate according to predicates over attributes that they expose. Communication takes place anonymously in an implicit multicast fashion without a prior agreement among the involved peers. Groups are dynamically formed by considering, among the potential receivers, those that satisfy the predicates of the sender; run-time changes of attribute values allow opportunistic interactions between components. By parameterising the interaction predicates with local attributes, groups can be implicitly changed and adaptation is naturally captured. Sending operations are non-blocking while receiving operations are; this breaks synchronisation dependencies between interacting partners, and permits modelling systems where agents can enter or leave at any time without perturbing the overall behaviour.

When devising new languages for a new class of systems, following [7], we think that it is important to consider three main ingredients:

1. a specification language equipped with a formal semantics, which associates mathematical models to each term of the language to precisely capture the expected behaviour of systems;
2. a set of techniques and tools, built on top of the models, to express and verify the properties of interest;
3. a programming framework together with an associated run-time environment, to actually execute the specified systems.

In this paper, we rely on *AbC* as our specification language, on UMC as the tool for property verification and on *Erlang* [8] to set up the programming framework that we call *ABEL* [9]. The execution and verification flows of our approach are summarised by the following diagram.



Starting from *AbC* specifications we obtain a UMC model whose properties we can express with UCTL and then model check with the tool. We see this as an iterative process that calls for a progressive revision of the specification in case any property of interest is not satisfied. Once satisfactory specifications are obtained, they are translated into *ABEL* and can finally be executed.

The main contributions of this paper, are thus the two translators (i) from *AbC* to the language of UMC and (ii) from *AbC* to the language of *ABEL*. The former is a refinement of a translation already presented in [10] while the latter is presented here for the first time to take advantage of the *ABEL* implementation presented in [9].

The feasibility of the approach is vindicated by considering a case study dealing with distributed graph colouring. We first show how errors in an initial specifications can be detected by exploiting the counterexample facility of UMC, then we show how properties of interest of a new version can be formally verified, finally we provide an executable *Erlang* program obtained from the correct specifications.

## 2 Background

In this section we briefly review the main ingredients of our approach, i.e., the *AbC* process calculus, the UMC verification framework, and the *ABEL* framework. We also illustrate *AbC* programming via a non-trivial example.

### 2.1 *AbC* process calculus

*AbC* (Attribute-based Communication calculus) is a process calculus specifically conceived for collective adaptive systems. In *AbC* [6], a system is a collection of interacting components; each component $C$ is either a process $P$ associated with an attribute environment $\Gamma$ and an interface $I$, or the parallel composition of two components. The environment $\Gamma$ is a partial mapping from attribute names to values, representing the component state. The interface $I$ is a set of exposed attribute names with which other components may use when specifying their predicates.

$$
\begin{array}{rl}
\text{(Components)} & C ::= \Gamma :_I P \mid C_1 \parallel C_2 \\
\text{(Processes)} & P ::= 0 \mid (\tilde{E})@\Pi.U \mid \Pi(\tilde{x}).U \mid \langle \Pi \rangle P \mid P_1 + P_2 \mid P_1 | P_2 \mid K \\
\text{(Update)} & U ::= \ [a := E]U \mid P \\
\text{(Expressions)} & E ::= v \ \mid \ x \ \mid \ a \ \mid \ this.a \ \mid \ f(\tilde{E}) \\
\text{(Predicates)} & \Pi ::= \mathrm{tt} \mid p(\tilde{E}) \mid \Pi_1 \ \wedge \ \Pi_2 \mid \neg \Pi
\end{array}
$$

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 $K \triangleq P$). The derived syntax allows no sequential composition between *AbC* processes, similar to TAPAs [11].

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

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

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

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

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

Attribute updates and awareness predicates are local to components and their execution includes the associated communication action, atomically.

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 $\Pi$ 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 $p(\tilde{E})$.

In $AbC$, the output action is non-blocking while the input action waits for synchronization on available messages. Parallel processes inside a single component do not communicate; they simply interleave their actions and have access to the shared environment. According to the system semantics, message passing is performed in a broadcast fashion. An outbound message is attached with the portion of environment limited by the senders' interface and the sending predicate. Every component which can execute an input action, upon receiving a message checks both the sending and receiving predicates to decide whether to use the message or discard it.

We now illustrate $AbC$ constructs by considering a distributed variant of graph colouring (borrowed from [12]). The problem amounts to labelling the vertices of a graph with a colour (in our case, positive integers) such that adjacent vertices have a different colour. We model the vertices as separate components of the form $\Gamma_i :_{\{id,nbr\}} V$, with public attributes a unique identifier, `id`, and a set, `nbr`, of neighbours. Additionally, the environment $\Gamma_i$ maintains the following private attributes for local computations:

- `colour, assigned`: the proposed colour and the colouring status
- `round`: the current round of computation
- `counter`: the number of un-assigned neighbours operating in the same round
- `constraints`: the set of colours proposed by greater neighbours in the same round
- `done`: the number of neighbours who have finished colour selection
- `used`: the set of colours already used by neighbours
- `send`: a flag controlling when to send colour proposals

The vertices operate in rounds and use predicates of the form $this.\texttt{id} \in \texttt{nbr}$ to communicate with neighbours. Each vertex has the same behaviour, and consists of four parallel processes $V \triangleq (F \mid T \mid D \mid A)$. Initially, `assigned` = false, `round`

$\texttt{= counter = done = colour} = 0$, $\texttt{constraints = used} = \emptyset$ and $\texttt{send} = \text{true}$. Any un$\texttt{assigned}$ vertex repeatedly performs two consecutive actions:

$$\begin{aligned} F \triangleq\ & \langle \texttt{send} \wedge \neg \texttt{assigned} \rangle ()@(\text{ff}).[\texttt{colour} := \min\{\text{i} \notin \texttt{used}\}] \\ & (\text{`}try\text{'}, this.\texttt{colour}, this.\texttt{round})@(this.\texttt{id} \in \texttt{nbr}).[\texttt{send} := \text{ff}]F \end{aligned}$$

to update the current $\texttt{colour}$ (after an empty output action), and to send a 'try' message of the form $(\text{`}try\text{'}, c, r)$ to inform neighbours that it wants to take colour c at round r. Note that $\min\{\text{i} \notin \texttt{used}\}$ denotes the smallest element not in $\texttt{used}$.

Each vertex counts the number of 'try' messages from its neighbours as described by the following expression:

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

where the first two branches deal with messages from neighbours in the same round (i.e., $this.\texttt{round} = z$);to avoid conflicts, the proposed colours from neighbours with greater ids are stored in $\texttt{constraints}$: only the vertex with the greatest id among un-assigned neighbours can take a conflict colour. Messages associated with a greater round are instead handled by the two branches. Any message of this kind will denote the beginning of a new round at the receiving end, by updating $\texttt{round, counter, constraints}$, and enabling $\texttt{send}$ in order to activate process $F$.

Upon succeeding in deciding a colour, a vertex sends a message of the form $(\text{`}done\text{'}, c, r + 1)$ to notify the others that the colour c has been taken at round r, setting $\texttt{assigned}$ to true and terminating:

$$\begin{aligned} A \triangleq\ & \langle (\texttt{counter} + \texttt{done} = |\texttt{nbr}|) \wedge \texttt{colour} > 0 \wedge \texttt{colour} \notin \texttt{constraints} \cup \texttt{used}) \rangle \\ & (\text{`}done\text{'}, this.\texttt{colour}, this.\texttt{round} + 1)@(this.\texttt{id} \in \texttt{nbr}).[\texttt{assigned} := \text{tt}]0 \end{aligned}$$

this process is activated if the vertex has collected all neighbours information (i.e., $\texttt{counter} + \texttt{done} = |\texttt{nbr}|$) and there are no conflicts (i.e., $\texttt{colour} \notin \texttt{constraints} \cup \texttt{used}$). At the other endpoint, a vertex receiving notification messages from neighbours updates $\texttt{done}$ and the set of $\texttt{used}$ colours, and, in case the message is associated with a greater round, triggers the execution of a new round:

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

### 2.2 UMC model checker

UMC [4] is one of the model checkers belonging to the KandISTI [13] formal verification framework used for analyzing functional properties of concurrent systems. In UMC, a system is represented as a set of UML-like communicating state machines, each associated with an active object in the system. UMC adopts doubly-labelled transition systems (L2TS) [14] as semantic model of systems behaviours. A L2TS is essentially a directed graph in which nodes and edges are labelled with sets of predicates and of events, respectively. The model checker allows to interactively explore graphs and to verify behavioural properties specified in the state-event logic UCTL [15]. UCTL allows to express state predicates and event predicates and to combine them with temporal and boolean operators in the style of CTL [16] and ACTL [2].

The main building block of a concurrent system in UMC is a class definition, that defines the structure a UML-like state machine. The active components of a system are represented by class instances, i.e. objects. A class definition has the following structure:

```
class Name is
  Signals
  -- asynchronous events accepted by the class
  Operations
  -- synchronous events accepted by the class
  Vars
  -- local variables of this object
  --
  -- state properties
  Behavior
  -- transitions that determine the behaviour of the class
end Name
```

The `Signals` and `Operations` sections contain the set of events to which an active object may react by triggering some transition of the state machine. The `Vars` section contains the private (non statically-typed) local variables of the class and optionally their initial value. Values can denote object names, boolean values, integer values or, recursively, (dynamically sized) sequences of values. The `Vars` section can be followed by a list of `state` definitions that allow to specify special properties of some of the states, like the list of events deferred inside those states. The `Behaviour` section contains a set of transition rules that describe the behaviour of the class and have the following general form:

```
source -> target {trigger [guard] / actions}
```

Source and target denote internal states of the state machine and may be defined by composite names in presence of composite (sequential or parallel) states. A single evolution step of a state machine has the semantics of a run-to-completion step as defined in [17]. At each step one event is extracted from events queue of the object and dispatched to the set of active states for which it may trigger a transition. A transition is enabled not only if the requested trigger is being currently dispatched, but also when its `guard` expression (if any) is satisfied by the current object state and trigger arguments. If different conflicting transitions are concurrently enabled, one of them is nondeterministically selected for firing, taking into account possible priorities. Concurrent non conflicting transitions are

instead executed in the same run-to-completion step. Transitions not requiring a triggering event (completion transitions) have a higher priority than normally triggered transitions. When a transition is fired the execution of its actions may change the state of the object and send further events to the same or other objects. UMC supports a fairly rich language to specify actions and guards. Actions, in particular, can be composite actions like finite loops or conditionals, and can use local temporary transition variables. The reader is referred to the UMC website [18] and to the documentation therein for additional details.

While the operational semantics in terms of state machine of a UMC specification is directly defined by the system behaviour, the evaluation of logical formulas on the system is carried out by reasoning on an abstract L2TS derived from systems behaviour by a set of Abstraction rules. These rules allow to decorate the states and the edges of the L2TS with relevant state predicates and abstract relevant events. They are defined inside the `Abstractions` section:

```
Abstractions {
  Action: <internal event> -> <edge label>
  ...
  State:  <internal system state> -> <node label>
  ...
}
```

The labels exposed can be visualized, to provide a compact summary of the computation trees, and can be referred when specifying UCTL formulae.

### 2.3 *ABEL*

*ABEL* [9] is a faithful implementation of *AbC* in *Erlang* with the support of APIs closely corresponding to *AbC* primitives. *ABEL* relies on provable-correct coordination strategy [19] for preserving *AbC* execution semantics. This is an advantage over previous proposals; using *ABEL* in our framework makes automatic translation immediate.

An *ABEL* program is essentially an *Erlang* program which uses *ABEL* APIs. It consists of the process definitions of the components, and of top-level statements for initialization. The syntax of components and processes is given in Fig. 1, where [*elem*] is used to denote a list of *elem*s. A component is created by invoking *new_component* which takes as parameters an attribute environment Env (a map) and an interaction interface I (a list). The command returns a component address $C$ which can be used by *start_component* to start the execution of $C$ from an initial behaviour referenced by $BRef$. The main building block of a process definition is function definition $BDef$. A definition takes two parameters: a component address $C$ and the current list $V$ of *bound variables* of the process. $V$ is initially empty and may be gradually updated with the messages received by input actions. The body of a definition contains a single command $Com$ determining the process behaviour.

A reference $BRef$ to a definition is a function of one parameter that may alter the list of bound variables of the wrapped function. A reference can be passed as parameters to commands so that they can continue with the referred behaviour. This way of programming is reminiscent of continuation passing style.

$$
\begin{array}{rll}
C ::= & \textbf{new\_component}(Env, I) & \text{Create} \\
 & \textbf{start\_component}(C, BRef) & \text{Start} \\
BDef ::= & beh\_name(\mathbf{C}, \mathbf{V}) \rightarrow Com. & \text{Definition} \\
BRef ::= & \textbf{fun}(\_\mathbf{V}) \rightarrow beh\_name(\mathbf{C}, \_\mathbf{V})\ \textbf{end} & \text{Reference} \\
 & \mid \textbf{nil} & \\
Com ::= & \textbf{prefix}(\mathbf{C}, \mathbf{V}, \{Act, BRef\}) & \text{Prefix} \\
 & \mid \textbf{choice}(\mathbf{C}, \mathbf{V}, [\{Act, BRef\}]) & \text{Choice} \\
 & \mid \textbf{parallel}(\mathbf{C}, \mathbf{V}, [BRef]) & \text{Parallel} \\
 & \mid \textbf{call}(\mathbf{C}, \mathbf{V}, BRef) & \text{Call} \\
Act ::= & \{\text{`!'}, g, \tilde{m}, s, [u]\} & \text{Output} \\
 & \mid \{\text{`?'}, g, r, \tilde{x}, [u]\} & \text{Input}
\end{array}
$$

**Fig. 1.** ABEL API for process definitions.

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

**prefix** - takes as parameter an action $Act$ and a continuation $BRef$. $Act$ can be either input ('?') or output ('!') action and its description is a tuple where $g$, $s$, and $r$ denote awareness, sending and receiving predicates, respectively, while $\tilde{m}$ denotes the message, $\tilde{x}$ denotes input-binding variables and $u$ denotes an attribute update. If $g$ or $u$ are omitted, *ABEL* treats them as $true$ and empty list [], respectively. This command executes $Act$ and then the behaviour encapsulated in $BRef$. The execution of an input action (if successful) returns a message; *ABEL* then continues by calling $BRef$ on an updated list of bound variables, calculated by appending the message to the current list $V$. If $Act$ is an output action, the continuation is determined by applying $BRef$ to $V$.

**choice** - takes as parameter a list of pairs, each providing a description of the prefixing action $Act$ and a continuation $BRef$. This command executes one of the actions and continues with the associated behaviour. Currently, *ABEL* does not support mixed choices between input and output actions.

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

**call** - executes the behaviour referenced by $BRef$ by applying $BRef$ to $V$.

The representation of *AbC* basic terms is based on *Erlang*. Function $g$ is parameterized with the local environment, $(fun(L) \rightarrow \ldots end)$. The tuple $\tilde{m}$ is composed of functions parameterized with the local environment, $(fun(L) \rightarrow \ldots end)$. $\tilde{x}$ is a tuple of atoms. Function $s$ is parameterized with the local environment and the environment of other components $(fun(L, R) \rightarrow \ldots end)$, while

$r$ is parameterized also with the incoming message ($fun(L, M, R) \rightarrow \ldots end$). Finally, $u$ is a pair whose first element is an attribute name and the second is a function parameterized with the local environment, and a message, in case the update is associated with an input operation ($fun(L, \langle M \rangle) \rightarrow \ldots end$).

In order to properly model the semantics of these elements, several helper functions are available: $att(a, E)$ refers to the value of attribute $a$ in an environment $E$; $msg(x, M)$ refers to the value of variable $x$ in a message $M$, and $var(x, V)$ refers to the value of $x$ in a list $V$ of bound variables.

## 3 From *AbC* to UMC

We now show how to model an $AbC$ system as a unique UML state machine whose state is the union of the states of the components in the system. The behaviour of each component is modelled by a concurrent region of a global parallel system state. This yields several advantages. First, the receivers of a sending predicate can be easily determined, because components can read the attribute environments of the others. Second, message broadcasting can be effectively modelled via signals to the state machine itself, to allow all targeted components to receive in parallel.

Our translation takes as input a collection of $AbC$ components of the form $\Gamma_k : \langle D_k, P_{init_k} \rangle$, where $\Gamma_k$, $D_k$, and $P_{init_k}$ denote the environment, the process definitions, and the initial behaviour, respectively. The output of our translation is a UML class whose structure is depicted in Figure 2.

```
0  [[(Γ0 : ⟨D0, Pinit0⟩, Γ1 : ⟨D1, Pinit1⟩ ..., Γs : ⟨Ds, Pinits⟩)]] =
1      Class System with RANDOMQUEUE is
2      Signals: allowsend(i:int),
3               bcast(tgt,msg,j:int);
4      Vars:
5         receiving:bool := false;
6         pc:int[];
7         bound:obj[];
8         /* Attribute vectors - attr1:int[]; attr2:int[]; ...*/
9      State Top Defers allowsend(i)
10     Transitions:
11        /* Initial movement of the system */
12        init → SYS {-/
13         for i in 0..pc.length-1 {
14             self.allowsend(i);
15        }}
16        /* Transitions of all components */
17        [[⟨D0, Pinit0⟩]]
18        [[⟨D1, Pinit1⟩]]
19        ...
20     end System
```

**Fig. 2.** Structure of the UMC encoding for $AbC$

The `System` class includes fixed code snippets such as the necessary signals and data structures to model $AbC$ input and output actions. Attributes (line 8)

are represented as vectors whose values are accessible via component indexes. Transitions model the components' behaviour (lines 17 - 20). The `Transitions` section contains the sets of transitions, for each component. Specifically, for component $k$, our translation visits all actions (of all processes) starting from $P_{initk}$ and uses $D_k$ for looking up new definitions when needed.

In the following we first explain the structural part of the translation, where we combine $AbC$ actions according to the process structure, and then the behavioural part, where $AbC$ input and output actions are modelled.

To model process interleaving we introduce program counters: `pc[k][p]` is the program counter for process `p` of component `k`. Furthermore, for each action in `p` we calculate two values: an *entry point* $\mathtt{C_{in}}$ and an *exit point* $\mathtt{C_{out}}$ based on the process structure. These values can be worked out by recursively visiting the process structure, following the approach of [10]. Fig. 3 gives an idea of how different transitions can be combined by using program counters and entry and exit points: (a) is the graphical representation of a single transition; (b) an action-prefixing process $\alpha.P$ has the entry point of $\alpha$ as $\mathtt{C_{in}}$, and the entry point of $P$ as the exit point of $\alpha$ ($\mathtt{C_{out}}$); (c) a choice process $P_1 + P_2$ has the same entry point on both sub-processes $P_1$ and $P_2$; (d) the entry points of sub-processes $P_1$ and $P_2$ in a parallel process $P_1|P_2$ contain their entry points, possibly combined with the exit point of a previous action. An action is then encoded as a transition of the following form:

```
SYS.Ck.s0 -> Ck.s0 {Trigger[... & pc[k][p]=Cin]/
   -- transition body
   pc[k][p]:=Cout;
}
```

A guard $\mathtt{pc[k][p]} = \mathtt{C_{in}}$ makes sure that the program counter points to the correct transition to be executed next. Right after a transition, `pc[k][p]` is assigned a new value to correctly enable the next set of feasible transitions.

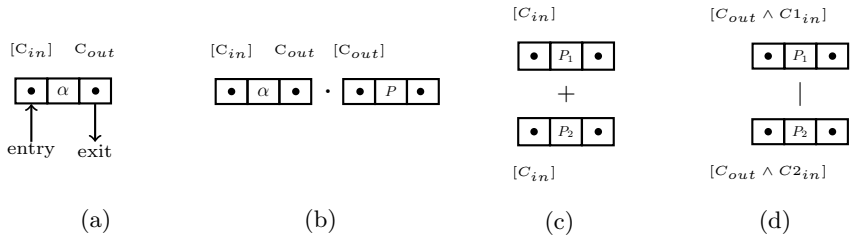



**Fig. 3.** Structural composition of UMC transitions with program counters

In order to model communication, we introduce two unique events to model attribute-based input and output actions. The event $\mathtt{bcast}(\mathtt{tgt}, \mathtt{msg}, \mathtt{j})$ carries out the set `tgt` of component indexes allowed to receive the message `msg`, and the index `j` of the sending component and triggers all the input actions in all

components. The event `allowsend(i)`, where `i` is a component index, schedules the components through interleaving when sending messages. The event queue of the state machine stores a set of `allowsend(i)` signals for each $AbC$ component. These signals are declared in the top state of the system as `Defers`, to prevent them from being removed from the event queue when they do not trigger any transition. The queue is defined as `RANDOM` so that the relative ordering of signals is not considered relevant. In this way, whenever an $AbC$ output action is allowed, a single `allowsend(i)` signal is nondeterministically selected from the queue to enable a single component, with index $i$, to proceed.

**Output and Input Actions.** We encode an output action as two separate transitions that do occur strictly sequentially: a sending operation to self (i.e., the state machine) of the `bcast` event (that will be dispatched to all the parallel components), and a discarding of this very message.

$[\![\langle \Pi_a \rangle (\tilde{E})@\Pi_s.[\tilde{a} := \tilde{E}]]\!]^{k,p,c_{in},c_{out}} =$

```
SYS.Ck.s0 -> Ck.s0 {
    allowsend(i)[i=k & receiving=false & [[Π_a]] & pc[k][p] = c_in]/
    tgt: int[];
    for j in 0..pc.length-1 {
      if ([[Π_s]]) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    receiving:=true;
    self.bcast(tgt,[[Ẽ]],k);
    [[ã := Ẽ]];
    pc[k][p] = c_in + 1;
}
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[pc[k][p] = c_in + 1]/
    receiving:=false;
    self.allowsend(k);
    pc[k][p] = c_out;
}
```

Note that an action may be preceded by an awareness predicate $\Pi_a$ and followed by an attribute update $[\tilde{a} := \tilde{E}]$. The global variable `receiving` of the state machine works as a lock, to guarantee the correct ordering of the two transitions. Here the (main) transition is enabled if the component $k$ is selected by UMC (i.e., `i=k`) and no other component is performing an output action, (i.e., `receiving=false`), and the awareness predicate $\Pi_a$, if any, holds. The transition body includes the computation of the set of potential receivers `tgt`, a sending operation `self.bcast(tgt,msg,k)` where `msg` is the result of translating expressions $\tilde{E}$ and of any attribute updates. This transition also sets the global variable `receving` to prevent other components from performing further transitions of this kind, and to allow only transitions triggered by `bcast(tgt,msg,k)`. The second transition resets `receiving`, updates the program counter of the current process to the correct exit point $\mathtt{c_{out}}$, and pushes the `allowsend(k)` signal on the event queue of the state machine.

An input action is instead translated into a single UMC transition, as below.

$[\![\langle \Pi_a \rangle \Pi_r(\tilde{x}).[\tilde{a} := \tilde{E}]]\!]^{k,p,c_{in},c_{out}} =$

```
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[tgt[k]=1 & [[Π_a]] & [[Π_r]] & pc[k][p]= c_in ]/
```

```
    bound[k][p] = msg;
    [[ã := Ẽ]];
    pc[k][p] = c_out;
}
```

The transition is enabled, for a component $k$ (if $k$ is in the target set `tgt` of receivers), when the receiving predicate $\Pi_{\mathrm{r}}$ and (possibly) the preceding awareness predicate $\Pi_a$ are satisfied. Variable binding is achieved by assigning the received message `msg` to a global vector `bound` indexed by $k$ and $p$. This allows other transitions of the same process to use the message. Like for output actions, the transition body may contain attribute updates.

For brevity, we have omitted the detailed treatment for other syntactic elements such as expressions and predicates. Their translations are quite straightforward since in each transition, we have full information of component and process indexes for accessing components attributes and bound variables.

## 4 From *AbC* to *ABEL*

We now describe the translation from *AbC* into *ABEL*. We make the following assumptions on the input: i) any process involved in a choice is preceded (guarded) by an action, although the choice branches can still initially appear in the form of process call; ii) the guarded actions in a choice are not mixed, i.e., all of them are either input actions or output actions; iii) the names of input-binding variable in the code of a process definition are unique.

Our translation is structured in two phases. A *normalization phase* that consists in refactoring process definitions (of all components) in the input *AbC* specification to match the structure the forms provided by *ABEL*'s programming interface, and a *generation phase* produces the actual *Erlang* code.

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

Fig. 4 presents the rewriting rules for normalization. We note that the rules are applied exhaustively until all definitions in $D$, including the newly created ones, are processed.

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

In a choice definition, the procedure recursively processes all the branches of the choice. For process names, $\mathcal{N}$ looks up the definition of that name and

$$
\begin{array}{lll}
\text{(prefix)} & \mathcal{N}[\![K \triangleq \alpha X]\!] & = \quad K \triangleq \alpha \cdot \mathcal{R}[\![X]\!] \\
\\
\text{(choice)} & \mathcal{N}[\![K \triangleq X_1 + X_2]\!] & = K \triangleq \mathcal{N}[\![X_1]\!] + \mathcal{N}[\![X_2]\!] \\
& \mathcal{N}[\![K]\!] & = \mathcal{N}[\![P]\!] \quad \text{where } P = D(K) \\
& \mathcal{N}[\![\alpha X]\!] & = \alpha \cdot \mathcal{R}[\![X]\!] \\
& \mathcal{N}[\![X_1 + X_2]\!] & = \mathcal{N}[\![X_1]\!] + \mathcal{N}[\![X_2]\!] \\
\\
\text{(parallel)} & \mathcal{N}[\![K \triangleq X_1 \mid X_2]\!] & = K \triangleq \mathcal{R}[\![X_1]\!] \mid \mathcal{R}[\![X_2]\!] \\
& \mathcal{N}[\![X_1 \mid X_2]\!] & = \mathcal{R}[\![X_1]\!] \mid \mathcal{R}[\![X_2]\!] \\
\\
\text{(call)} & \mathcal{N}[\![K \triangleq K']\!] & = \quad K \triangleq K' \\
\\
\text{(new def.)} & \mathcal{R}[\![K]\!] & = K \\
& \mathcal{R}[\![P]\!] & = K \text{ for K fresh and } D = D \cup \{K \triangleq P\}
\end{array}
$$

**Fig. 4.** Normalizing process definitions

continues the normalization for the corresponding code. For prefixing process, $\mathcal{N}$ behaves similarly to the prefixing case. If one of the branches is again a choice process, $\mathcal{N}$ normalizes its sub-processes.

In a parallel definition, the procedure recursively normalizes all the branches. Finally, any call definition remains unchanged. The above procedure is guaranteed to terminate because the input specification contains a finite number of definitions and $\mathcal{N}$ processes them only once.

**Code Generation.** This phase produces *Erlang* code from a set of normalized definitions. The generated code consists of a module for each component type wherein each normalized definition is translated into one function. The rules for code generation ($\mathcal{G}$) are reported in Fig. 5. The first four rules capture all possible forms of a definition and generate the corresponding $BDef$ definitions in *ABEL* style (see Fig. 1). The next two rules do generate references $BRef$. The other next two rules deal with *AbC* actions. The remaining nine rules are responsible for the actual translation of the basic elements of such actions: Namely, they consider awareness, sending and receiving predicates $\Pi_a, \Pi_s, \Pi_r$, attribute updates $[\tilde{a} := \tilde{E}]$ and the message $\tilde{E}$ to be sent. The translation is parameterized with input-binding variables $\tilde{x}$ because the expressions contained in receiving predicates or in attribute updates may need them.

Please, notice that the details of the translation of the last five rules are omitted. The definition of $[\![\cdot]\!]$ is quite standard and is based on the actual syntax of *AbC* predicates and expressions. However, we would like to add some consideration about some special cases. In an awareness predicate $\Pi_a$, the two terms $a$ and $this.a$ have the same meaning, therefore $[\![a]\!] = [\![this.a]\!] = \texttt{att(a,L)}$. The translation for interaction predicates ($\Pi_s$, $\Pi_r$) instead differs between $a$ and $this.a$ being $a$ interpreted as a remote attribute: $[\![a]\!] = \texttt{att(a,R)}$ while $[\![this.a]\!] = \texttt{att(a,L)}$. Next, consider a variable $x$. The special case is when $x$ appears in input-binding

$$
\begin{array}{ll}
\mathcal{G}[\![K \triangleq \alpha \cdot K']\!] & = \mathtt{k(C,V)} \rightarrow \mathtt{prefix(C,V,\{}\mathcal{G}[\![\alpha]\!], \mathcal{G}[\![\mathtt{K'}]\!]\mathtt{\}).} \\
\mathcal{G}[\![K \triangleq \alpha_1 \cdot K_1 + \ldots + \alpha_n \cdot K_n]\!] & = \mathtt{k(C,V)} \rightarrow \mathtt{choice(C,V,[\{}\mathcal{G}[\![\alpha_1]\!], \mathcal{G}[\![\mathtt{K_1}]\!]\mathtt{\}, \ldots, \{}\mathcal{G}[\![\alpha_n]\!], \mathcal{G}[\![\mathtt{K_n}]\!]\mathtt{\}]).} \\
\mathcal{G}[\![K \triangleq K_1 | \ldots | K_m]\!] & = \mathtt{k(C,V)} \rightarrow \mathtt{parallel(C,V,[}\mathcal{G}[\![\mathtt{K_1}]\!], \ldots, \mathcal{G}[\![\mathtt{K_m}]\!]\mathtt{]).} \\
\mathcal{G}[\![K \triangleq K']\!] & = \mathtt{k(C,V)} \rightarrow \mathtt{call(C,V,}\mathcal{G}[\![\mathtt{K'}]\!]\mathtt{).} \\
\mathcal{G}[\![K]\!] & = \mathtt{fun(\_V)} \rightarrow \mathtt{k(C,\_V)\ end} \\
\mathcal{G}[\![nil]\!] & = \mathtt{nil} \\
\mathcal{G}[\![\langle \Pi_a \rangle (\tilde{E}) @ (\Pi_s).[\tilde{a} := \tilde{E}]]\!] & = \{\text{`!'}, \mathcal{G}[\![\Pi_a]\!], \mathcal{G}[\![\tilde{E}]\!], \mathcal{G}[\![\Pi_s]\!], \mathcal{G}[\![[\tilde{a} := \tilde{E}]]\!]\} \\
\mathcal{G}[\![\langle \Pi_a \rangle (\Pi_r)(\tilde{x}).[\tilde{a} := \tilde{E}]]\!] & = \{\text{`?'}, \mathcal{G}[\![\Pi_a]\!], \mathcal{G}[\![\Pi_r]\!]^{\tilde{x}}, \mathcal{G}[\![\tilde{x}]\!], \mathcal{G}[\![[\tilde{a} := \tilde{E}]]\!]^{\tilde{x}}\} \\
\mathcal{G}[\![\tilde{E}]\!] & = \{\mathcal{G}[\![E_1]\!], \ldots, \mathcal{G}[\![E_k]\!]\} \\
\mathcal{G}[\![\tilde{x}]\!] & = \{\mathtt{x_1}, \ldots, \mathtt{x_1}\} \\
\mathcal{G}[\![[\tilde{a} := \tilde{E}]]\!] & = [\{\mathtt{a_1},\mathcal{G}[\![E_1]\!]\}, \ldots, \{\mathtt{a_i},\mathcal{G}[\![E_i]\!]\}] \\
\mathcal{G}[\![[\tilde{a} := \tilde{E}]]\!]^{\tilde{x}} & = [\{\mathtt{a_1},\mathcal{G}[\![E_1]\!]^{\tilde{x}}\}, \ldots, \{\mathtt{a_j},\mathcal{G}[\![E_j]\!]^{\tilde{x}}\}] \\
\mathcal{G}[\![\Pi_a]\!] & = \mathtt{fun(L)} \rightarrow [\![\Pi_a]\!]\ \mathtt{end} \\
\mathcal{G}[\![\Pi_s]\!] & = \mathtt{fun(L,R)} \rightarrow [\![\Pi_s]\!]\ \mathtt{end} \\
\mathcal{G}[\![\Pi_r]\!]^{\tilde{x}} & = \mathtt{fun(L,M,R)} \rightarrow [\![\Pi_r]\!]^{\tilde{x}}\ \mathtt{end} \\
\mathcal{G}[\![E]\!] & = \mathtt{fun(L)} \rightarrow [\![E]\!]\ \mathtt{end} \\
\mathcal{G}[\![E]\!]^{\tilde{x}} & = \mathtt{fun(L,M)} \rightarrow [\![E]\!]^{\tilde{x}}\ \mathtt{end}
\end{array}
$$

**Fig. 5.** Code generation

variables $\tilde{x}$, to which the parameterized translation output `msg(x,M)`, otherwise `var(x,V)`. Finally, for complex expressions $f(\tilde{E})$ (or predicates) which do not have a closed form in *AbC* syntax, the translation generates a function call as a place holder, i.e., $\mathtt{f}(\mathcal{G}[\![\tilde{E}]\!])$, and users need to provide the *Erlang* definition for $f$ afterward.

## 5 Experiments

In this section we report some experiments that we have conducted adopting as a case study the graph colouring algorithm of Sect. 2.1 by taking advantage of the two tools [4] that permit translating *AbC* systems into UMC models and into *ABEL*.

**Experimenting with UMC.** In order to model check the UMC model for graph colouring, we instrumented it as follows.

```
Abstractions {
   -- Auto generated
   Action sending($1,$2) -> send($1,$2)
   Action received($1,$2) -> received($1,$2)
   State SYS.colour[0]=$2 ->  has_colour(C1,$2)
   State SYS.assigned[0]=$2 ->  has_assigned(C1,$2)
   ...
   -- Manual instruments
   -- Soundness
   State SYS.matrix[0][1]>0 and SYS.colour[0]=SYS.colour[1] -> not_sound
```

[4] https://doi.org/10.5281/zenodo.3234713

```
    State SYS.matrix[0][2]>0 and SYS.colour[0]=SYS.colour[2] -> not_sound
    ...
    -- \delta + 1 algorithm
    State SYS.maxt<SYS.colour[0] -> bad_alg
    State SYS.maxt<SYS.colour[1] -> bad_alg
    ...
}
```

First, we introduce abstraction rules for the `colour` and `assigned` attributes of each vertex. Second, whenever the `colour` of any two adjacent vertices is the same, we label the state as `not_sound`. For this we rely on a global variable storing the adjacency `matrix` of the graph under consideration. Finally, a label `bad_alg` is exposed if the maximum degree + 1 (stored in `maxt`) is smaller than the colour value of any vertex. We can now specify a number of properties concerning the termination and soundness of the graph colouring procedure:

$G_1$ (*termination*) The system converges to final states:
`AF FINAL`

$G_2$ (*completeness of colouring*) Every vertex has a valid colour:
`AF (FINAL and not has_colour(*,0)) and not has_assigned(*,false))`

$G_3$ (*soundness of colouring*) Adjacent vertices do not share the same colour:
`AF (FINAL and not not_sound)`

$G_4$ *(not bad algorithm)* The algorithm is $(\Delta + 1)$ - colouring:
`AF (FINAL and not bad_alg)`

We have verified the system under consideration for *all* possible graphs with 2 from 5 vertices. Property $G_1$ holds, showing that our colouring algorithm terminates, while property $G_2$ does not hold, because some vertex eventually is not `assigned`. Figure 6 reports the UMC counterexample for a connected graph of 2 vertices. After being assigned a colour, vertex 2 still sends a 'try' message,

```
AF (FINAL and not has_assigned(*,false)) is FOUND_FALSE in State C1
This happens because:
  ...
  C13 --> C68  {} /* nbr:=[[2],[1]]; matrix:=[[-1,1],[1,-1]]; */
  C68 --> C69  {} /* colour[0]:=1; pc[0][0]:=2 */
  C68 --> C69  {} /* colour[0]:=1; */
  C69 --> C70  {send(C1,[try,1,0])} /* send[0]:=false; */
  C70 --> C71  {receive(C2,[try,1,0])} /* counter[1]:=1; */
  C71 --> C228  {} /* colour[1]:=1; */
  C228 --> C238  {send(C2,[donec,1,1])} /* assigned[1]:=true; */
  C238 --> C239  {receive(C1,[donec,1,1])} /* round[0]:=1; done[0]:=1;
                  send[0]:=true; counter[0]:=0; used[0]:=[1];
                  constraints[0]:=[]; */
  C239 --> C240  {} /* colour[0]:=2;  */
  C240 --> C241  {send(C1,[try,2,1])} /* send[0]:=false; */
  C241 --> C242  {receive(C2,[try,2,1])} /* send[1]:=true; counter[1]:=1;
                  constraints[1]:=[]; */
  C242 --> C245  {send(C2,[try,1,1])} /* send[1]:=false;  */
  C245 --> C246  {receive(C1,[try,1,1])} /* counter[0]:=1;
                  constraints[0]:=[1]; */
(C246 is final)
```

**Fig. 6.** Counterexample for property $G_2$

which affects the counting of the neighbours for vertex 1. This prevents the halting condition for colour selection, i.e., $|\texttt{nbr}| = \texttt{done} + \texttt{counter}$ to ever happen. This happens because in the specifications (Sect. 2.1), process, $F$ performs colour selection and sends a 'try' message as two separate actions, therefore there is the possibility that these two actions are interleaved with process $A$.

We can fix the specifications by modifying process $F$ as follows:

$$F' \triangleq \langle \texttt{send} \wedge \neg \texttt{assigned} \rangle (\text{`}try\text{'}, \min\{\text{i} \notin \texttt{used}\}, this.\texttt{round})@(this.\texttt{id} \in \texttt{nbr}).$$
$$[\texttt{colour} := \min\{\text{i} \notin \texttt{used}\}, \texttt{send} := \text{ff}]F'$$

where basically the two actions of $F$ are combined into a single action wherein the sending action sends the value of the expression $\min\{\text{i} \notin \texttt{used}\}$, and `colour` is atomically set via an attribute update. After these changes, we can perform the verification again. The results are summarised in Figure 1. UMC successfully verified that, for all possible graphs of the considered degrees, all the above properties do hold for the fixed version of the specifications. The number of states generated when verifying the property AF FINAL is also included, giving an idea of the state space for each set of input.

| property | $G_1$ | $G_2$ | $G_3$ | $G_4$ |
|---|---|---|---|---|
| original spec. | ✓ | × | n/a | n/a |
| fixed spec. | ✓ | ✓ | ✓ | ✓ |
| n. of vertices | 2 | 3 | 4 | 5 |
| state space | 85 | 1,469 | 52,068 | 2,859,341 |

**Table 1.** Verification results for the initial and the refined specifications

**Experimenting with ABEL.** Using the second translator, we generated an *Erlang* program for the considered case study from the correct version of the specifications. We run this program on some random graphs from [20], and observed the outcome of the colouring procedure. Below we report the outcome of the *ABEL* program that received as input the `Heawood` graph shown on the right in Fig. 7; the corresponding run of the *ABEL* program is reported in the left part of the figure.

It is worth observing that *ABEL* can deal with systems with a very large number of components. For instance, we have also experimented with graphs up to 1000 vertices (components), measuring critical parameters such as the number of message exchanges, the total size of messages for each scenarios. For detailed performance evaluations, we refer the reader to [9].

## 6 Concluding Remarks and Future Work

We have presented an integrated framework for programming and verifying systems that are formally described by a theoretically well-founded process calculus, namely *AbC*. Starting from *AbC* specifications, our framework can be used

| Vertex | Colour | Round |
|---|---|---|
| 14 | 1 | 0 |
| 1 | 2 | 1 |
| 9 | 2 | 1 |
| 13 | 2 | 1 |
| 8 | 1 | 2 |
| 12 | 1 | 2 |
| 2 | 1 | 3 |
| 7 | 2 | 3 |
| 11 | 2 | 4 |
| 3 | 2 | 4 |
| 6 | 1 | 4 |
| 5 | 2 | 5 |
| 4 | 1 | 6 |
| 10 | 1 | 6 |

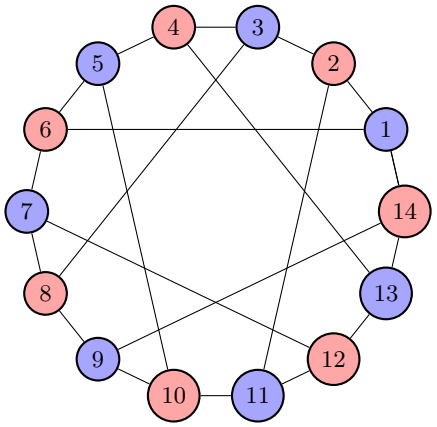



**Fig. 7.** Execution trace of the automatically generated *ABEL* program

for verifying and executing *AbC* systems by relying on external tools. While this work selects only two environments for *AbC*, we think that the outlined methodology is sufficiently general to be extended to analyze and execute *AbC* with other verification and programming environments.

For automatic verification, we would like to experiment with adapting our translation to other frameworks and model checkers such as SPIN, for which the work in [21] can be a good starting point. Recent surveys [22,23] have described a possible approach to translating simple state machines into a formalism supported by different frameworks, but further analysis is needed to fully assess the complexity of the effort. Moreover, it has still to be understood whether the modeling language is sufficiently expressive to define the properties required for an *AbC* specification. For programming environments, other approaches have been considered to provide implementations of *AbC*, see e.g., [24,25,26]. However, these works do not fully capture the the original *AbC* semantics or exhibit a significant gap between their programming constructs and the *AbC* primitives, making the translation not obvious. Among the above mentioned implementations, the most promising alternative to *ABEL* appears to be GoAt [25], although revisions are required for it as well.

Closely related to our work is the one presented in [27] which generates the Java code from the verified spi-calculus specification. They assume the correctness of an underlying Java implementation, and focus on proving the correctness of code translation and generation. We have not yet considered the correctness of our translators.

Static verification may be out of reach for large systems. Some lines of work such as the P programming language [28] promote testing on formal models instead of verification. Runtime-monitoring techniques [29] offload the verifica-

tion to post deployment. It would be interesting to understand whether these alternatives would fit our model-driven approach to develop *AbC* systems.

## References


1. R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori, "An action-based framework for verifying logical and behavioural properties of concurrent systems," *Computer Networks and ISDN Systems*, vol. 25, no. 7, pp. 761–778, 1993.
2. R. De Nicola and F. W. Vaandrager, "Action versus state based logics for transition systems," in *Semantics of Systems of Concurrent Processes, LITP*, ser. Lecture Notes in Computer Science, I. Guessarian, Ed., vol. 469. Springer, 1990, pp. 407–419.
3. S. Gnesi and F. Mazzanti, "On the fly verification of network of automata," in *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA*, H. R. Arabnia, Ed. CSREA Press, 1999, pp. 1040–1046.
4. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti, "A state/event-based model-checking approach for the analysis of abstract system properties," *Science of Computer Programming*, vol. 76, no. 2, pp. 119–135, 2011.
5. Y. Abd Alrahman, R. De Nicola, and M. Loreti, "On the power of attribute-based communication," in *Formal Techniques for Distributed Objects, Components, and Systems - FORTE 2016, Proc.*, ser. Lecture Notes in Computer Science, E. Albert and I. Lanese, Eds., vol. 9688. Springer, 2016, pp. 1–18.
6. ——, "A behavioural theory for interactions in collective-adaptive systems," *CoRR*, vol. abs/1711.09762, 2017. [Online]. Available: http://arxiv.org/abs/1711.09762
7. R. De Nicola, G. L. Ferrari, R. Pugliese, and F. Tiezzi, "A formal approach to the engineering of domain-specific distributed systems," in *Coordination Models and Languages - COORDINATION 2018, Proc.*, ser. Lecture Notes in Computer Science, G. Di Marzo Serugendo and M. Loreti, Eds., vol. 10852. Springer, 2018, pp. 110–141.
8. J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology, Stockholm, 2003.
9. R. De Nicola, T. Duong, and M. Loreti, "ABEL - a domain specific framework for programming with attribute-based communication," in *Coordination Models and Languages - COORDINATION 2019, Proc.*, ser. Lecture Notes in Computer Science. Springer, 2019, to appear.
10. R. De Nicola, T. Duong, O. Inverso, and F. Mazzanti, "Verifying properties of systems relying on attribute-based communication," in *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, J. Katoen, R. Langerak, and A. Rensink, Eds. Springer, 2017, vol. 10500, pp. 169–190.
11. F. Calzolai, R. De Nicola, M. Loreti, and F. Tiezzi, "TAPAs: A tool for the analysis of process algebras," in *Trans. Petri Nets and Other Models of Concurrency*, ser. Lecture Notes in Computer Science, K. Jensen, W. M. P. van der Aalst, and J. Billington, Eds., vol. 5100. Springer, 2008, pp. 54–70.
12. Y. Abd Alrahman, R. De Nicola, and M. Loreti, "Programming the interactions of collective-adaptive systems by relying on attribute-based communication," *CoRR*, vol. abs/1711.06092, 2017. [Online]. Available: http://arxiv.org/abs/1711.06092

13. M. H. ter Beek, S. Gnesi, and F. Mazzanti, "From EU projects to a family of model checkers - from kandinsky to kandisti," in *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, ser. Lecture Notes in Computer Science, R. De Nicola and R. Hennicker, Eds., vol. 8950. Springer, 2015, pp. 312–328.
14. R. De Nicola and F. W. Vaandrager, "Three logics for branching bisimulation," *J. ACM*, vol. 42, no. 2, pp. 458–487, 1995. [Online]. Available: http://doi.acm.org/10.1145/201019.201032
15. A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi, "A logical verification methodology for service-oriented computing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2012.
16. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.
17. OMG, "Unified modeling language version 2.5 - behavioral statemachines," Object Management Group, Tech. Rep., 2015. [Online]. Available: https://www.omg.org/spec/UML/2.5/PDF
18. "The UMC verification framework," http://fmt.isti.cnr.it/umc.
19. Y. Abd Alrahman, R. De Nicola, G. Garbi, and M. Loreti, "A distributed coordination infrastructure for attribute-based interaction," in *Formal Techniques for Distributed Objects, Components, and Systems - FORTE 2018, Proc.*, ser. Lecture Notes in Computer Science, C. Baier and L. Caires, Eds., vol. 10854. Springer, 2018, pp. 1–20.
20. G. Brinkmann, K. Coolsaet, J. Goedgebeur, and H. Mélot, "House of graphs: a database of interesting graphs," *Discrete Applied Mathematics*, vol. 161, no. 1-2, pp. 311–314, 2013.
21. R. De Nicola, A. Lluch-Lafuente, M. Loreti, A. Morichetta, R. Pugliese, V. Senni, and F. Tiezzi, "Programming and verifying component ensembles," in *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, Proc.*, ser. Lecture Notes in Computer Science, S. Bensalem, Y. Lakhnech, and A. Legay, Eds., vol. 8415. Springer, 2014, pp. 69–83.
22. F. Mazzanti, A. Ferrari, and G. O. Spagnolo, "Towards formal methods diversity in railways: an experience report with seven frameworks," *STTT*, vol. 20, no. 3, pp. 263–288, 2018. [Online]. Available: https://doi.org/10.1007/s10009-018-0488-3
23. F. Mazzanti and A. Ferrari, "Ten diverse formal models for a CBTC automatic train supervision system," in *Proc. Third Workshop on Models for Formal Analysis of Real Systems MARS/VPT@ETAPS 2018, Thessaloniki, Greece, 20th April 2018.*, 2018, pp. 104–149. [Online]. Available: https://doi.org/10.4204/EPTCS.268.4
24. R. De Nicola, T. Duong, O. Inverso, and C. Trubiani, "AErlang: Empowering Erlang with Attribute-Based Communication," in *Coordination Models and Languages - COORDINATION 2017, Proc.*, ser. Lecture Notes in Computer Science, J. Jacquet and M. Massink, Eds., vol. 10319. Springer, 2017, pp. 21–39.
25. Y. Abd Alrahman, R. De Nicola, and G. Garbi, "GoAt: Attribute-based interaction in Google Go," in *Leveraging Applications of Formal Methods, Verification and Validation - ISoLA 2018, Proc.*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 11246. Springer, 2018, pp. 288–303.
26. Y. Abd Alrahman, R. De Nicola, and M. Loreti, "Programming of CAS systems by relying on attribute-based communication," in *Leveraging Applications of Formal Methods, Verification and Validation - ISoLA 2016, Proc.*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 9952, 2016, pp. 539–553.

27. A. Pironti and R. Sisto, "Provably correct java implementations of spi calculus security protocols specifications," *Computers & Security*, vol. 29, no. 3, pp. 302–314, 2010.
28. A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: safe asynchronous event-driven programming," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 321–332, 2013.
29. I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "A survey of runtime monitoring instrumentation techniques," in *Proc. Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017.*, ser. EPTCS, A. Francalanza and G. J. Pace, Eds., vol. 254, 2017, pp. 15–28.