Verifying properties of systems relying on attribute-based communication

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. AbC is a process calculus designed for describing collective adaptive systems, whose distinguishing feature is the communication mechanism relying on predicates over attributes exposed by components. A novel approach to the analysis of concurrent systems modelled as AbC terms is presented that relies on the UMC model checker, a tool based on modelling concurrent systems as communicating UML-like state machines. A structural translation from AbC specifications to the UMC internal format is provided and used as the basis for the analysis. Three different algorithmic solutions of the well studied stable marriage problem are described in AbC and their translations are analysed with UMC. It is shown how the proposed approach can be exploited to identify emerging properties of systems and unwanted behaviour.

1 Introduction

In the eighties much work was devoted to formalisms for the specification and verification of concurrent systems. It was already clear that this class of systems was going to become more and more important even if the Internet, as we know it today, was not yet available[4]. In that period in Twente University there was a group of researchers working on the theory of concurrent systems. That theory was based on the explicit synchronization and message passing primitives proposed by Milner [25] and Hoare [19], and the researchers wanted to improve its usability. Indeed, they gave a great contribution to the development of the language LOTOS that in [8] is introduced as "a specification language that has been specifically developed for the formal description of the OSI (Open systems Interconnection) architecture, although it is applicable to distributed, concurrent systems in general. In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment."

[4] Just consider that the email address(es) of the friend to whom this volume is dedicated were something like `uucp: mcvax!utinu1!infed` and `earn: hiddink@henttht5`.

The main actor behind the effort was Ed Brinskma, who contributed to both the definition of the language and to the proof techniques to verify conformance of communication network protocols implementations with their abstract specifications [15,9].

Since then, communication networks have dramatically changed our world and we are now working with autonomous agents that roam over the Internet, adapt to changing situations and environments, interact with other agents or humans and control essential components of our daily life. It is more and more common that such autonomous agents interact anonymously and form groups of peers dynamically according to specific features, or attributes, that the different peers expose. For instance, members of a social network interested in language exchange activities can use their own location and favourite languages to find suitable people nearby.

Thus, the old formalisms and especially their communication primitives, based on broadcast or direct one-to-one, communication are not appropriate anymore for selecting partners and programming so called collective adaptive systems. New formalisms based on alternative communication paradigms and supported by new proof techniques are on demand for dealing with them. Prompted by the needs outlined above, we have defined AbC [4] a novel process calculus that relies on *attribute-based communication* and formalises the above intuition by combining actor-style concurrency with one-to-many message passing. Traditional linguistic approaches struggle in the presence of highly-dynamical environments often seen in real-world situations, from social networks to stock exchanges. AbC can instead cope with these systems quite naturally, usually keeping the specifications compact and intuitively easy to follow [11].

The effectiveness of this new formalism has so far been assessed mostly from a programming standpoint, with prototype implementations of the proposed interaction mechanisms in Java [5] and Erlang [11]. However, the potential benefits of AbC when reasoning about system properties have only been hinted at, through proof-of-concept verification of simple properties of formal models manually built from AbC specifications [10].

In this paper, we report our first attempt to the systematic analysis and verification of attribute-based communication systems. The initial step in our verification approach consists in mechanically translating the AbC specification of a given system into a UML-like state machine. In AbC the supported communications primitives require some kind of global view of the attributes of all the components of a system. The most direct way to model this global status is to see it as the internal status of a nondeterministic state machine, in which the behaviour of a single process term is captured by one or more state machine transitions. In this way, each process can have access to the values of the attributes of the other processes to effectuate AbC-style communication. In order to preserve the structure of the AbC process, we explicitly keep track of the execution point of each process and use to guard the transitions. Depending on the properties of interest, relevant structural and behavioural aspects of the state machine can be made observable and accessible to the logical verification engine, through the definition of specific abstractions rules.

For system analysis, our approach relies on the UMC verification framework [20]. UMC is specifically oriented towards the early analysis of (likely wrong) initial system designs, that trades the capacity of dealing with very large systems with the capacity of helping users to easily understand the source of design errors. This is achieved, among the other things, by providing interactive explanations of the results of the evaluations and by allowing the user to observe and reason on systems at a high level of abstraction without being distracted, if not overwhelmed, by all the details of the specification.

We illustrate the impact of our approach by considering three variants of the well studied stable marriage problem (SMP) [17] that can be naturally expressed in terms of partners' attributes. Solving the SMP problem amounts to finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element. Thus, one has to find an algorithm for pairing each element in one set to an element in the other set in such a way that there are no two elements of different pairs which both would rather have each other than their current partners. When no such pair of elements exists, the set of pairs is deemed stable. The classical algorithm of [17] goes through a sequence of proposals initiated by members of one group (*the initiators*) according to their preferences. Members of the other group (*the responders*) after receiving a proposal, do choose the best initiator between their current partner and the one making advances. It can be proved that such an algorithm guarantees existence of a unique stable matching.

Our variants of SMP allow initiators and responders to express their interests in potential partners by using their attributes rather than their identities ordered by means of an explicit preference list. Member's preferences are represented as predicates over the attributes of potential partners. For one variant, we follow the classical algorithm where initiators first propose to the responder they prefer most and then relax their expectations if no partner is willing to accept their proposal. In the other variant, initiators start proposing with the lowest requirements, to make sure to get a partner, and gradually increase their expectations hoping to find better partners.

We experimented our verification methodology on the three above mentioned algorithmic solutions to SMP by considering a number of properties of interest, such as stability of the matching and its completeness, existence of a unique solution, level of satisfaction of the components. These properties are first described informally and then rendered as logical formulae to be formally checked against the generated models. The outcome of our verification allows us to make some considerations both on the different algorithms and on the used tools.

Indeed, the results of our experiments have shown that systems relying on attribute-based communications can be particularly complex to design and analyse. However, by exhaustively verifying a specification over all possible inputs, despite the small problem size considered, we have experienced that many non-trivial emerging properties and potential problems can indeed be discovered by following our methodology.

The rest of the paper is organised as follows. We briefly introduce the AbC process calculus and the UMC model checker in Section 2. We describe our translation from AbC process terms into UMC textual description of UML-like state machines in Section 3. In Section 4, we show how to specify in AbC solutions for both classical SMP and its attribute-based variants, and then present fragments of the result of our verification and discuss their outcomes. Finally, Section 5 contains some concluding remarks.

2 Background

The AbC calculus. AbC [4] is a process calculus centered on the attribute-based communication paradigm. Its core syntax is reported in Figure 1. An AbC system consists of independent components (C). A component can be either a process (P) and an attribute environment (Γ) or a parallel composition of components $C_1 \parallel C_2$. The behaviour of a component is modeled by process P

$$\begin{array}{rl}
\text{(Components)} & C ::= \Gamma : P \mid C_1 \parallel C_2 \\
\text{(Processes)} & P ::= 0 \mid (E)@\Pi.P \mid \Pi(x).P \mid [a := E]P \mid \langle \Pi \rangle P \\
 & \qquad \mid P_1 + P_2 \mid P_1|P_2 \mid K \mid \mathbf{\Pi ? P_1 \_\_ P_2} \\
\text{(Expressions)} & E ::= v \mid x \mid a \mid \mathit{this.a} \\
\text{(Predicates)} & \Pi ::= \mathrm{tt} \mid E_1 \bowtie E_2 \mid \Pi_1 \wedge \Pi_2 \mid \neg \Pi
\end{array}$$

Fig. 1. AbC syntax

executing actions in the style of process algebra, while its attributes are used to encode some domain aspects (e.g. battery level, component role, identity, ...) and are stored in the component's environment Γ which is a partial mapping from attribute names to their values. Since attributes play a key role in interactions, AbC assumes that their names are agreed in advance among components [4].

A process P can be either an inactive process 0, a prefixing process $\alpha.P$, an update process $[a := E]P$, 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 (under the assumption that each process has a unique definition $K \triangleq P$).

We often omit the inactive process for convenience. The prefixing process executes the action α and continues as P. The update process sets the attribute a to the value of expression E and behaves like P. The awareness process blocks the execution of process P until predicate Π becomes true. The choice process can behave either like P_1 or P_2. The parallel process interleaves the executions of P_1 and P_2. For modelling communication AbC relies on two prefixing actions:

$(E)@\Pi$ is the *attribute-based output* that is used to send the value of expression E to those components whose attributes satisfy predicate Π;

$\Pi(x)$ is the *attribute-based input* that binds to the variable x the message received from any component whose attributes, and possibly the communicated values, satisfy the receiving predicate Π.

The semantics of output actions are asynchronous and non-blocking while that of input actions are blocking. The receiving predicates can also be specified over the message content, in addition to the attributes of sending components. Parallel components communicate using these two primitives, while parallel processes within a component simply interleave their executions. An update operation is performed atomically with the following action, given that the component under the updated environment can perform that action. In some cases, to model an update operation alone [a:=E], we exploit an empty send action ()@(ff) to obtain $[a := E]()@(\mathrm{ff})$.

An expression E may be a constant value v, a variable x, an attribute name a, or a reference *this.a* to attribute a in the local environment. Predicate Π can be either tt, a comparison between two expressions $E_1 \bowtie E_2$, a logical conjunction of two predicates $\Pi_1 \wedge \Pi_2$ or a negation of a predicate $\neg\Pi$. We write $\Gamma \models \Pi$ to state that predicate Π holds in environment Γ.

All the above mentioned constructs have already been introduced in [4], and we refer the interested reader to this paper for the definition of the operational semantics of the full calculus.

There is however a new operator that we introduce for the first time in this paper and is very important to support processes in taking decisions depending on the conditions of the context they are operating in. We have called the new operator, the *awareness operator*:

$$\Pi ? P_1 \_\_ P_2$$

relies on a sort of global awareness and allows the executing system to proceed as P_1 if the environment contains at least one component whose attributes satisfy predicate Π, and as P_2 otherwise. Its operational semantics is modelled by the followng inference rules:

$$\frac{\exists\, C : \Gamma(C) \models \Pi \quad P_1 \xrightarrow{\alpha} P_1'}{\Pi ? P_1 \_\_ P_2 \xrightarrow{\alpha} P_1'} \qquad \frac{\nexists\, C : \Gamma(C) \models \Pi \quad P_2 \xrightarrow{\alpha} P_2'}{\Pi ? P_1 \_\_ P_2 \xrightarrow{\alpha} P_2'}$$

The main difference between the local awareness operator and the global one is that the predicate appearing in the latter can refer to the attributes of external components.

In fact, the attribute-based communication primitives have been introduced while abstracting from the selection mechanism of communication partners, e.g., ignoring how predicates are evaluated and how components address each other. When it comes to practical settings, both for programming and verifying AbC systems, one has to take into consideration those issues which in turn raise the problem of designing a communication infrastructure. Existing implementations of AbC paradigm [5,10] do rest on a centralized component which plays the role of a global registration and a message forwarder. This component has global knowledge of the system while other components are not aware of each other and only interact with this centralized one. In this paper, we also assume that there is a such global component in an AbC system. While this assumption guides our translation strategy which will be presented in Section 3, one important benefit is that it allows implementing operators like $\Pi?P_{1}\_\_\,P_2$, needing global awareness.

The UMC model checker. UMC [20] is one of the model checkers belonging to the KandISTI [6] formal verification framework used for analyzing functional properties of concurrent systems. In UMC, a system is represented as a set of communicating UML-like state machines, each associated with an active object in the system. UMC adopts doubly-labelled transition systems (L2TS) [14] as semantic model of the system behaviour. 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 this graph and to verify behavioral properties specified in the state-event based UCTL [16] logics. UCTL allows to express state predicates over (the labelling of) system states, event predicates over (the labelling of) single-step system evolutions, and combine these with temporal and boolean operators in the style of CTL and ACTL. A UML-like state machine is described in UMC in the form of a class declaration structured as below:

```
class Name is
  Signals:
  -- asynchronous signals accepted by this class
  Vars:
  -- local variables of this object
  Transitions:
  -- transitions that determine the behaviour of the class
end Name
```

where a list of `Signals` summarises the set of events to which an active object may react[5]. A signal denotes an asynchronous event that may trigger the transitions of an object. An object can send signals to itself by executing `self.signal_name`. 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 `Transitions` section declares a set of transition rules which describe the behaviour of the class and have the following general form:

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

to denote a state transition from state `source` to state `target`. The transition is triggered by a suitable trigger event `trigger` (which is a signal name) and if the `guard` expression is satisfied, all actions inside the transition body are executed. The execution of actions may in turn change the state of the object or

[5] UMC also supports an `Operations` section for the definition of synchronous events, which is however not relevant in our study

trigger other transitions. In fact, UMC supports a fairly rich language to specify actions and guards. For more details we refer to the UMC website [2] and the documentation therein.

While the structure of the semantic in terms of L2TS of an UMC specification is directly defined by the system behaviour, the labels associated to nodes and edges of the graph are specified by *abstraction* rules that allow the designer to define the relevant internal aspects of the system. These rules are defined inside the `Abstractions` section:

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

The possibility of obtaining an L2TS which focuses only the aspects of the system that are considered relevant is particularly useful in many cases. For example one can visualize a compact summary of the computation trees, factorized via appropriate behavioural equivalence notions. Or he can model check abstract L2TS (without any knowledge of the underlying UMC), and reason on systems without a detailed knowledge of the underlying concrete implementation.

3 Transforming AbC models into UMC models

In this section we describe a mechanical translation from AbC specifications into to UML-like state machines. The main effort in this part lays in the careful modelling of the attributes and of their visibility, which implicitly require some sort of global view of the the system. In fact, since our target modelling language has no concept of global data, a simple solution would require at least implementing shared states and appropriate synchronisation. We avoid that by gathering all the processes in the initial system along with their attributes into a unique object in the translated model, where the behaviour described by a single process term is captured by one or more transitions. This successfully provides a direct access to attributes to every process. However, some ingenuity is required to respect the process structure of the input system and its precise semantics. We thus introduce an explicit tracking mechanism for the execution points of the processes. This amounts to dynamically labelling new terms while visiting the process structure, and to introducing appropriate guards for the transitions, to guarantee that at any point of the evolution of the system only feasible transitions are allowed. Labels and guards can be combined to model sequentialisation, non-deterministic choice, and parallel composition.

We now describe the translation in detail. Our input system is a collection of AbC components, where the specification for the i-th component, denoted with

$$C_i ::= \Gamma_i : \langle D_i, P_{init_i} \rangle$$

includes an attribute environment Γ_i, a set D_i of process declarations, and an initial behaviour P_{init_i} which refers to the processes defined in D_i. We adopt the following notational conventions. Expressions can be vectors with relevant operators in the UMC style [3], e.g., given a vector v, we can write $v.head, v.tail, v[i]$ for the first element, the rest and the ith element of v, respectively. Predicates can contain tests of membership relation between an element and a vector (denoted by $\in, \notin$). We further assume that the specified system consists of a fixed number of components, and that the parallel operator does not occur inside a recursive definition. The only allowed exception is the definition of a process of the form $P \coloneqq Q|P$, where $|$ is replaced by its bounded version $|^m$, i.e. the number of parallel instances to be created. For example: $P := Q\ |^2\ P$ is interpreted as three processes $P \coloneqq Q_1 \mid Q_2 \mid Q_3$.

The output of our translation is a UMC class whose general structure is depicted in Fig. 2. It includes fixed code snippets such as the necessary signals and data structures to model AbC input and output actions. It also contains vectors to model attribute environments, one for each attribute.

```
Class AbCSystem is
Signals: allowsend(i:int),
         bcast(tgt,msg,j:int);
Vars:
   RANDOMQUEUE;
   receiving:bool := false;
   pc:int[];
   bound:obj[];
   /* Attributes vectors */
   att1:int[]; att2:int[]; ...
State Top Defers allowsend(i)
Transitions:
   /* Initial movement of the system */
   init -> SYS {- /
     for i in 0..pc.length-1 {
        self.allowsend(i);
   }}
   /* Transitions of all components */
   $\mathcal{S}[\![\langle D_1, P_{init1}\rangle]\!]$
   $\mathcal{S}[\![\langle D_2, P_{init2}\rangle]\!]$
   ...
end AbCSystem
```

Fig. 2. Translation of AbC specifications.

The system state is a UML parallel state (`SYS`), where each component is modelled by its own region (`Ck`). Attribute input and output semantics are modelled with the help of unique events. The `bcast(tgt,msg,j)` event,which triggers all the receive actions in all components and contains the actual set `tgt` of components allowed to receive the message, the actual message `msg`, and the index `j` of the sending component. The `allowsend(i)` event, where `i` denotes a component index used to schedule the components through interleaving when sending messages. According to the semantics of AbC, *receive* actions are blocking and executed together, and *send* actions of all the components should be handled in an interleaved way. To accommodate this, we use the event queue of the state machine to store a set of `allowsend(i)` signals, one for each AbC component `i`. These signals are declared in the top state of the system as `Defers`, to prevent them from being removed from the events queue when they do not trigger any transition. Moreover, the queue is defined as `random` so that the relative ordering of signals is not considered relevant. In this way, at each step in which an AbC *send* operation has to be performed, a single `allowsend` signal is nondeterministically selected from the queue, allowing a single component to proceed.

The `Transitions` section collects all the transitions generated from the process terms while visiting the process structure. Transitions have the following form:

```
SYS.Ck.s0 -> Ck.s0 {Trigger[... & pc[k][p]=CNT]/
   -- transition body
   pc[k][p]:=CNT + 1;
}
```

where `CNT` is a *program counter* initially set to 1 and incremented as new transitions are produced. This provides a unique label associated with the transition, its *entry point*. Additionally, `pc[k][p]` is the *execution point* of process `p` in component `k`. The guard on the transitions makes sure that its entry point matches the execution point of the corresponding component and process. At the end of a transition, `pc[k][p]` is assigned a new value, referred to as its *exit point*, in order to correctly enable the next set of feasible transitions. The values of `k`, `p`, `CNT`, and the full guards are worked out according to the structural mapping procedure described below.

Structural Mapping. Let us denote with $\mathcal{S}[\![P]\!]_{\rho}^{k,p,v}$ the function that maps a process term P into a set of UMC transitions, where k is the component index, p the process index, and v the entry value. At the beginning, P is the init behaviour of the component and p, v are both initialised with 1. The information carried while traversing the process structure is stored in ρ. Fig. 3 presents our translation rules from AbC process terms to UMC transitions, while Fig. 4(b)-(d) gives an idea of how transitions are glued together according to the process structure. The translation maintains two variables: the current number of processes *procs* and a program counter $cnt[p]$ for a process with index p, both calculated dynamically while visiting the input.

Inaction. An inaction process is translated into nothing.

$$
\begin{aligned}
\mathcal{S}[\![nil]\!]_{\rho}^{k,p,v} &= \emptyset \\
\mathcal{S}[\![[a := E]P]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v} \\
&\quad \texttt{where } \rho' = \rho\{upd \mapsto \rho(upd) \cup [a := E]\} \\
\mathcal{S}[\![\langle \Pi \rangle P]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v} \\
&\quad \texttt{where } \rho' = \rho\{aware \mapsto \rho(aware) \cup \Pi\} \\
\mathcal{S}[\![P_1 + P_2]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P_1]\!]_{\rho}^{k,p,v}; \mathcal{S}[\![P_2]\!]_{\rho}^{k,p,v} \\
\mathcal{S}[\![P_1 | P_2]\!]_{\rho}^{k,p,v} &= \mathcal{S}[\![P_1]\!]_{\rho'}^{k,p_1,1}; \mathcal{S}[\![P_2]\!]_{\rho'}^{k,p_2,1} \\
&\quad \texttt{where } p_1 = procs,\ p_2 = procs + 1, \\
&\qquad \rho' = \rho\{parent \mapsto \rho(parent) \cup (p, v)\}, \\
&\qquad procs = procs + 2, cnt[p_1] = 1, cnt[p_2] = 1 \\
\mathcal{S}[\![K]\!]_{\rho}^{k,p,v} &= \begin{cases} \mathcal{B}[\![\emptyset]\!]_{\rho}^{k,p,exit} & \text{if } \rho(K_{\text{visit}}) = \text{true} \\ \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v} & \text{otherwise} \end{cases} \\
&\quad \texttt{where } \text{exit} = \rho(K_{\text{entry}}), P = D_k(K), \\
&\qquad \rho' = \rho\{K_{\text{visit}} \mapsto true, K_{\text{entry}} \mapsto v\} \\
\mathcal{S}[\![(E)@(\Pi).P]\!]_{\rho}^{k,p,v} &= \mathcal{B}[\![(E)@(\Pi)]\!]_{\rho}^{k,p,v}; \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v'} \\
&\quad \texttt{where } v' = cnt[p] + 2, cnt[p] = cnt[p] + 2 \\
&\qquad \rho' = \rho\{upd \mapsto \emptyset, aware \mapsto \emptyset, parent \mapsto \emptyset\} \\
\mathcal{S}[\![\Pi(x).P]\!]_{\rho}^{k,p,v} &= \mathcal{B}[\![\Pi(x)]\!]_{\rho}^{k,p,v}; \mathcal{S}[\![P]\!]_{\rho'}^{k,p,v'} \\
&\quad \texttt{where } v' = cnt[p] + 1, cnt[p] = cnt[p] + 1 \\
&\qquad \rho' = \rho\{upd \mapsto \emptyset, aware \mapsto \emptyset, parent \mapsto \emptyset\}
\end{aligned}
$$

Fig. 3. Structural translation of processes: semicolon (;) denotes the completion of a left translation before starting a new one, $\rho(x)$ is the value of variable x in ρ

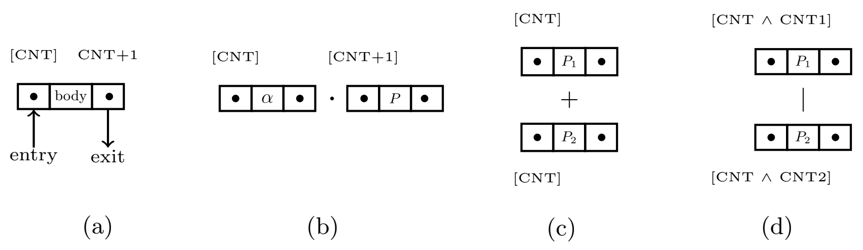

Fig. 4. Structural mapping to combine generated UMC transitions: (a) graphical representation of a transition; (b) an action prefixing process $\alpha.P$ has the entry point of α as `CNT`, and the entry point of P as `CNT + 1`; (c) a choice process $P_1 + P_2$ has the same entry point on both sub-processes P_1, P_2; (d) the entry points of sub-processes P_1, P_2 in a parallel process $P_1|P_2$ contain also the exit point of $P_1|P_2$

Update. The translation of $[a := E]P$ accumulates the update expression $[a := E]$ into variable *upd* of ρ and returns the translation of P under the new environment.

Awareness. The translation of $\langle \Pi \rangle P$ accumulates predicate Π into variable *aware* of ρ and returns the translation of P under the new environment.

Nondeterministic choice. The translation of $P_1 + P_2$ is a sequence of two translations of sub-processes with the same set of parameters and the same environment.

Parallel composition. The translation of $P_1|P_2$ is a sequence of two translations of the sub-processes. It generates two new processes indices p_1 and p_2 which are calculated from the current number of processes *procs*, and initialises two new global counters $cnt[p_1]$, $cnt[p_2]$. In the case of parallel composition, the entry points of sub-processes P_1, P_2 does contain not only their own counters but also the counter of the spawning process $P_1|P_2$. Therefore, the translations of P_1, P_2 store the exit point of the parent process (p, v) in variable *parent* which will be used as an additional guard for prefixing actions of P_1 and of P_2.

Process call. The translation of a process call K looks up its definition P in the process declarations D_k and returns a translation of P. If process K is already translated, function $\mathcal{B}$ generates a dummy transition whose exit point is equal to the entry point of K. Otherwise, it remembers K is visited and stores this fact together with the entry point value of K into ρ for (possibly) later recursions.

Action-prefixing. The translation of $\alpha.P$ is a behavioural translation of α and a translation of the continuation process P. The translation of α is done by the function $\mathcal{B}$, as it will be presented shortly. The translation of P is parameterised with a new environment where the previous accumulated information is reset, and a new entry value v', calculated from the value of program counter $cnt[p]$, is added. In fact, we need two UMC transitions for an output action, thus the value of $cnt[p]$ is increased by two.

Accumulated information. In the above generated transitions, guards may include the accumulated awareness predicates and the transition body may include accumulated update commands. We omit the details for conciseness.

Global Awareness. Finally, the global awareness construct $\Pi?P_1 \_\_ P_2$ is treated as a process whose structure is $\beta.(P_1 + P_2)$ where the transition of β simply evaluates the global predicate Π to enable transitions of P_1 and P_2 (via additional guards) appropriately.

Behavioural mapping. We now describe the function $\mathcal{B}[\![\alpha]\!]^{k,p,v}_{\rho}$ which generates the actual UMC code for a specific action α according to the information accumulated in the environment ρ and the parameter set.

We model the output action in two steps that are forced to occur in a strict sequence: the sending to self of the `bcast` event that dispatches to all the parallel components and the discarding of this very message, as illustrated by the code snippet below. Variable `receiving` works as a lock, to guarantee the correct ordering of the two transitions. Here the (main) transition is guarded by conditions on the execution point of the action and by awareness predicates, while the transition body includes update commands, the computation of potential receivers and a sending operation.

$\mathcal{B}[\![(E)@\Pi]\!]^{k,p,v}_{\rho} =$

```
SYS.Ck.s0 -> Ck.s0 {
    allowsend(i)[i=k & receiving=false & pc[k][p]=v & ⟦ρ(parent)⟧] & ⟦ρ(aware)⟧]/
    ⟦ρ(update)⟧;
    for j in 0..pc.length-1 {
        if (⟦Π⟧) then {tgt[j]:=1;} else {tgt[j]:=0;}
    };
    receiving:=true;
    self.bcast(tgt,⟦E⟧,k);
    pc[k][p] = v + 1;
}
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[pc[k][p] = v + 1]/
    receiving:=false;
    self.allowsend(k);
    pc[k][p] = v + 2;
}
```

An input action is translated into the following transition, triggered by signal `bcast(tgt,msg,j)` from some sender. It is enabled, for a component k, if the message is for it, the receiving predicate Π and, possibly, the preceding awareness predicates are satisfied. Variable binding is done by assigning the received message `msg` to vector `bound`. Similarly to the output action, the transition guard might contain awareness predicates; the transition body might contain update commands.

$\mathcal{B}[\![\Pi(x)]\!]^{k,p,v}_{\rho} =$

```
SYS.Ck.s0 -> Ck.s0 {
    bcast(tgt,msg,j)[tgt[k]=1 & pc[k][p]=v & ⟦ρ(parent)⟧ & ⟦ρ(aware)⟧ & ⟦Π⟧]/
    ⟦ρ(update)⟧;
    bound[k][p] = msg;
    pc[k][p] = v + 1;
}
```

4 A Case Study

The Stable Marriage Problem (SMP) [17] is a well studied problem that has applications in a variety of real-world situations, such as assigning students to colleges or appointing graduating medical students to their first hospital. SMP that has been initially formulated in terms of peers that make offer to potential partners by taking into account a preference list is easily adaptable to a context in which partners are selected according to their attributes. Indeed, due to its simple formulation and its intrinsically concurrent nature, SMP has been already used to show the advantages of AbC as a very high-level formalism to describe complex systems [10,11]. In this section, we use it to show how our framework can be used to reason about properties of attribute based systems.

We apply our verification methodology to three possible algorithmic solutions of stable marriage in order to check a selection of properties of interest. For each solution, we provide a short informal description along with the resulting formal specifications in AbC. Similarly, we present a number of properties first informally and then as a precise logical property of the state machines generated (see Sect. 3) from the formal specifications. We show how to instrument these state machines for property checking. As we go along, we also consider a few additional program-specific properties that we used as a guidance to refine the formal specifications themselves.

The idea of stable marriage is to find a stable matching between two equally sized sets of elements (men and women in the original formulation, whence the word marriage) given an ordering of preferences for each element. Providing a solution to SMP amounts to devising an algorithm for pairing each element in one set to an element in the other set in such a way that there are no two elements of different pairs which both would rather have each other than their current partners. When no such pair of elements exists, the set of pairs is deemed stable. The classical algorithm of [17] goes through a sequence of proposals initiated by members of one group (*the initiators*) according to their preference lists. Members of the other group (*the responders*) after receiving a proposal, do choose the best initiator between their current partner and the one making advances. Our first algorithm implements the classical solution, where preferences are represented as complete ordered lists of identifiers. Initiators and responders are programmed as individual processes that interact using their local preference lists in a point-to-point fashion using their identity. The other two programs adapt the classical solution to the context of attribute-based communication, where partners are selected by considering predicates over the attributes of the potential partners. The two new solutions differ for the way initiators choose their potential partners. They can start by either making proposals to the responder they prefer most and then relax their expectations or making proposals with the lowest requirements, to make sure to get a partner, and gradually increase their expectations.

4.1 Specifications

Classical Stable Marriage. In the classical solution to stable marriage, each initiator actively proposes himself to the most favourite responder according to its preference list. In case it gets a refusal or is dropped, it tries again with the next element in the list. A responder waits for incoming proposals, accepting any proposal when single, or choosing between its current partner and the new proposer according to its preferences. The algorithm terminates when there is no more activity.

AbC Specification. Our first AbC program is based on the idea presented in [5]. Initiators and responders are AbC components whose attributes are the identifier *id*, the preference list *prefs*, and the current *partner*. The behaviour of an individual initiator is specified by process M. It updates attribute *partner* to the first element of *prefs*, and then sends a *propose* message to components whose *id* equals to *partner*. The continuation process Wait waits for a *no* message to reset the *partner*, before restarting with M:

$$\begin{aligned} \mathrm{M} &\triangleq [\mathrm{this}.\mathit{partner} := \mathrm{this}.\mathit{prefs.head}, \mathrm{this}.\mathit{prefs} := \mathrm{this}.\mathit{prefs.tail}] \\ &\quad (\mathit{propose}, \mathrm{this}.\mathit{id})@(\mathit{id} = \mathrm{this}.\mathit{partner}).\mathrm{Wait} \\ \mathrm{Wait} &\triangleq [\mathrm{this}.\mathit{partner} := 0](\$\mathit{msg} = \mathit{no})(\$\mathit{msg}).\mathrm{M} \end{aligned}$$

The behaviour of a responder is specified by process W. In process Handle a responder waits for incoming proposals, and behaves either like A if the responder finds the new initiator is better or like R otherwise. W is composed in parallel with n instances (where n is the problem size) so that it can receive new messages while processing the current one. Notice that both R and A use a reversed form of preference lists to compare the current partner with a new initiator:

$$\begin{aligned} \mathrm{W} &\triangleq \mathrm{Handle} \mid^{n} \mathrm{W} \\ \mathrm{Handle} &\triangleq (\$\mathit{msg} = \mathit{propose})(\$\mathit{msg}, \$\mathit{id}).(\mathrm{A}(\$\mathit{id}) + \mathrm{R}(\$\mathit{id})) \\ \mathrm{A(id)} &\triangleq \langle \mathrm{this}.\mathit{prefs}[\mathrm{this}.\mathit{partner}] < \mathrm{this}.\mathit{prefs}[\$\mathit{id}] \rangle \\ &\quad [\$\mathit{ex} := \mathrm{this}.\mathit{partner}, \mathrm{this}.\mathit{partner} := \$\mathit{id}](\mathit{no})@(\mathit{id} = \$\mathit{ex}) \\ \mathrm{R(id)} &\triangleq \langle \mathrm{this}.\mathit{prefs}[\mathrm{this}.\mathit{partner}] > \mathrm{this}.\mathit{prefs}[\$\mathit{id}] \rangle (\mathit{no})@(\mathit{id} = \$\mathit{id}) \end{aligned}$$

Top-down Stable Marriage. In this case preferences are expressed as predicates over the attributes of partners rather than as lists of people. For example, a person might be interested in finding a partner from a specific country who speaks a specific language. A suitable communication predicate would be $\mathit{country} = \mathrm{this}.\mathit{favcountry} \ \land \ \mathit{language} = \mathrm{this}.\mathit{favlanguage}$, where $\mathit{language}$ and $\mathit{country}$ are two attributes of initiators and responders, and $\mathit{favcountry}$ and $\mathit{favlanguage}$ are used to express preferences.

Following the above idea, in the top-down solution to SMP the initiator starts by making offers to responders that satisfy its highest requirements, i.e., have all wanted attributes. In case nobody satisfy these requirements, the initiator retries after weakening the predicate by eliminating one of the preferred attributes and waits for a reaction. The system then evolves as follows.

A single initiator that receives a *yes* considers himself engaged and sends out a $confirm$ message; it keeps proposing if a *no* is received. An engaged initiator that receives a *yes* notifies the interested responder that meanwhile another partner has been found by sending a *toolate* message. An engaged initiator dropped by its current partner with a *bye* message restarts immediately proposing.

An engaged responder reacts upon receiving a proposal by comparing the new initiator with the current partner. If the new proposer is not better, it will receive a *no* message. Otherwise, the responder sends a *yes* to notify the proposer her availability, and waits for a reply. Upon receiving a $confirm$, the responder changes partner and sends *bye* to the ex partner; in case a *toolate* message is received the responder continues without changes.

AbC Specification. We model a scenario where each participant exposes two characteristics besides their identifiers: $\{id, w, b\}$ for proposers and $\{id, e, h\}$ for responders. Furthermore, participants have their own preferences on which are modeled by $\{pe, ph\}$ and $\{pw, pb\}$. The behaviour of a proposer is modeled as process P, used to make proposals, composed in parallel with process $\mathrm{M}_{\mathrm{Handle}}$ for handling replies.

$$\begin{aligned}
\mathrm{M} &\triangleq \mathrm{P} \mid \mathrm{M}_{\mathrm{Handle}}\\
\mathrm{P} &\triangleq \langle \text{this}.\mathit{partner} = 0 \land \text{this}.\mathit{proposed} = 0\rangle[\text{this}.\mathit{proposed} := 1]\,\mathrm{P}_1\\
\mathrm{P}_1 &\triangleq \mathit{\Pi}_{\{\neg bl,pe,ph\}}?(\tilde{v})@\mathit{\Pi}_{\{\neg bl,pe,ph\}}.\,\mathrm{P} \mathrel{\text{--}} (\mathit{\Pi}_{\{\neg bl,pe\}}?(\tilde{v})@\mathit{\Pi}_{\{\neg bl,pe\}}.\,\mathrm{P} \mathrel{\text{--}} (\tilde{v})@\mathit{\Pi}_{\{\neg bl\}}.\,\mathrm{P})
\end{aligned}$$

Process P, guarded by two conditions this.$partner = 0$ and this.$proposed = 0$, becomes actives when a single proposer has not yet sent a proposal. After that, it sets the flag *proposed* and continues as P_1. To model the adaptive behaviour of proposers needed to relax their preferences, we use the new global awareness operator (see Sect. 2) in the definition of P_1 where we use $\mathit{\Pi}_{\{\neg a_1,a_2,a_3\}}$ to denote the predicate in the form $id \notin a_1 \land e = a_2 \land h = a_3$ and $\tilde{v}$ denotes the sent message, i.e., $\{propose, \text{this}.id, \text{this}.w, \text{this}.b\}$. It is important to add an attribute *bl* which is a list of responders that the proposer does not want to contact. The list is updated when a proposer receives a *no* or a *bye* message. This allows them to know when to relax their requirements.

A proposer may receive multiple replies; process $\mathrm{M}_{\mathrm{Handle}}$ takes care of this according to the message type: Wait is used to handle *bye* and *no* messages while

Yes handles *yes* messages.

$$\begin{aligned}
\mathrm{M}_{\mathrm{Handle}} &\triangleq \mathrm{Yes} \mid \mathrm{Wait} \qquad \mathrm{Yes} \triangleq \mathrm{Loop} \mid^n \mathrm{Yes} \qquad \mathrm{Wait} \triangleq \mathrm{Loop}_1 \mid^n \mathrm{Wait}\\
\mathrm{Loop} &\triangleq (\$msg = yes)(\$msg, \$id).\mathrm{Ans}(\$\mathrm{id})\\
\mathrm{Ans(id)} &\triangleq \langle \mathrm{this}.partner = 0\rangle[\mathrm{this}.partner := \$id, \mathrm{this}.bl := \mathrm{this}.bl + [\$id]]\\
&\quad (confirm)@(id = \mathrm{this}.partner).\mathrm{Loop}\\
&\quad + \langle \mathrm{this}.partner \neq 0\rangle(toolate)@(id = \$id).\mathrm{Loop}\\
\mathrm{Loop}_1 &\triangleq [\mathrm{this}.partner := 0, \mathrm{this}.proposed := 0](\$msg = bye)(\$msg, \$id).\mathrm{Loop}_1\\
&\quad + (\$msg = no)(\$msg, \$id).\\
&\quad [\mathrm{this}.proposed := 0, \mathrm{this}.bl := \mathrm{this}.bl + [\$id]]()@(\mathtt{ff}).\mathrm{Loop}_1
\end{aligned}$$

The behaviour of a responder is specified by $\mathrm{W}_{\mathrm{Handle}}$. On receiving a proposal, a responder can behave like A (accept), R (reject) or D (discard). The local attribute *bl* is updated in *A* and *R* while *D* uses it to avoid unnecessary processing. Acceptance and rejection of a proposal are dealt with similarly as in the classical case, except that the extra message acknowledgement requires an attribute *lock*, to process sequentially possibly parallel messages.

$$\begin{aligned}
\mathrm{W}_{\mathrm{Handle}} &\triangleq (\$msg = propose)(\$msg, \$id, \$w, \$b).\\
&\quad (\mathrm{R}(\$id, \$w, \$b) + \mathrm{A}(\$id, \$w, \$b) + \mathrm{D}(\$id))\\
\mathrm{R(id, w, b)} &\triangleq \langle \$id \notin \mathrm{this}.bl \wedge (new\_init\_is\_not\_better)\rangle\\
&\quad [\mathrm{this}.bl := \mathrm{this}.bl + [\$id]](no, \mathrm{this}.id)@(id = \$id).\mathrm{W}_{\mathrm{Handle}}\\
\mathrm{A(id, w, b)} &\triangleq \langle \$id \notin \mathrm{this}.bl \wedge \mathrm{this}.lock = 0 \wedge (new\_init\_is\_better)\rangle\\
&\quad [\mathrm{this}.lock := 1, \mathrm{this}.bl := \mathrm{this}.bl + [\$id]]\\
&\quad (yes, \mathrm{this}.id)@(id = \$id).\mathrm{Wait}(\$id, \$w, \$b)\\
\mathrm{Wait(id, w, b)} &\triangleq [\mathrm{this}.ex := \mathrm{this}.partner, \mathrm{this}.partner := \$id,\\
&\quad \mathrm{this}.cw := \$w, \mathrm{this}.cb := \$b](\$msg = confirm)(\$msg).\\
&\quad [\mathrm{this}.lock := 0](bye)@(id = \mathrm{this}.ex).\mathrm{W}_{\mathrm{Handle}}\\
&\quad + [\mathrm{this}.lock := 0, \mathrm{this}.bl := \mathrm{this}.bl - [\$id]]\\
&\quad (\$msg = toolate)(\$msg).\mathrm{W}_{\mathrm{Handle}}\\
\mathrm{D(id)} &\triangleq \langle \$id \in \mathrm{this}.bl\rangle()@(\mathtt{ff}).\mathrm{W}_{\mathrm{Handle}}
\end{aligned}$$

In the above specifications, the pair (cw, cb) denotes the characteristics of current partner, which is used by the responder to compare him with a new proposer. For example, predicate *new_init_is_better* is encoded as:

$$\begin{aligned}
&(\mathrm{this}.partner = 0) \vee (\$w = \mathrm{this}.pw \wedge \mathrm{this}.cw \neq \mathrm{this}.pw) \vee\\
&(\$w = \mathrm{this}.cw \wedge \$b = \mathrm{this}.pb \wedge \mathrm{this}.cb \neq \mathrm{this}.pb)
\end{aligned}$$

Bottom-up Stable Marriage. We have also experimented with another approach to SMP, where proposers start looking for the less-liked partner and try to incrementally improve their level of satisfaction by continuously proposing themselves even after finding a partner in the attempt to find someone they like more then their current one. In this case both proposers and responders can be dropped by their current partner if a more appreciated option pops up.

We have implemented this protocol in AbC using a slightly different approach. We used an extra process in components Proposer and Responder that plays the role of a message queue manager. This process appends every incoming messages to the tail of queue, while another process implementing the main behaviour retrieves messages from the queue and processes them sequentially.

Due to space limit, we omit the presentation of this specification. The interested reader can refer to [1] for full specifications of case studies.

4.2 Formal Analysis

UMC models and annotations. We have developed a tool [1] to implement the translation rules presented in Sect. 3. This tool has been used to translate the three AbC solutions for SMP into UMC models. The number of UMC code lines varies depending on specification and on the input instances. For example, in the classical case, the number of UMC lines are the same for component `M`, while it increases proportionally with the size of the problem for component `W` due to the use of operator $|^n$.

The actual UMC model used for the analysis is composed by two objects: an object, triggered by a *start*(<inputdata>) event, modelling the behaviour of the AbC specification with the given input data, and an object which generates all the possible input data and activates the AbC model with them. For checking the generic (i.e for all inputs) validity of a formula ϕ we in practice evaluate the formula A[{not *start*} W {*start*} ϕ], which says that ϕ holds in the initial state of any of the possible scenarios. The number of generated system states reported in the rest of this section refers to the cumulative data over the whole input domain.

In order to verify our properties of interests, we have annotated the generated UMC models with abstraction rules to make observable labels on states and actions.

```
Abstractions {
   State id[0]=$1 and partner[0]=$2 ->  haspartner($1,$2)
   State id[1]=$1 and partner[1]=$2 ->  haspartner($1,$2)
   ...
   Action sending($1,$2) -> send($1,$2)
   Action received($1,$2,$3) -> received($1,$2,$3)
   -- Other instrument
   Action m_decr -> m_decr
   Action w_decr -> w_decr
}
```

Here rules starting with `States` expose labels *haspartner*($1, $2) in all system states, where $1 is the identifier of a component (proposer or responder) and

\$2 is the matching partner. We assume that the identifiers of initiators and responders are in the ranges $[1 \ldots n]$ and $[n+1 \ldots 2n]$ respectively, with n being the problem size. Rules starting with `Actions` instead expose *send* and *receive* labels on all transitions denoting attribute send and receive actions.

We have additionally instrumented the models with more involved annotations. In particular, we store the current level of satisfaction of people, computed when a component updates its partner. In classical SMP, the level of satisfaction of initiators and responders is determined by the position of the current partner in the preference list. In the attribute-based variant, this number is calculated based on the similarity between one's own preferences and the characteristics of partners. The procedure issues a signal `decr` if the current computed satisfaction level is smaller than the previous one.

Solution-independent properties. For all the three AbC specifications, we are interested in checking the following properties:

F_1 (*convergence*) The system converges to final states:
AF FINAL [6]

F_{2a} (*completeness of matching*) Everybody has a partner:
AF (FINAL implies not *haspartner*(*,0))

F_{2b} (*uniqueness of matching*) There exists only one final matching:
AG (((EF(FINAL and *haspartner*(1,4))) implies AF (FINAL and *haspartner*(1,4)))
and ((EF(FINAL and *haspartner*(1,5))) implies AF(FINAL and *haspartner*(1,5)))
and ((EF(FINAL and *haspartner*(1,6))) implies AF(FINAL and *haspartner*(1,6))))

F_{2c} (*symmetry of matching*) The matchings are symmetric:
AG (FINAL implies ((*haspartner*(1,4) implies *haspartner*(4,1))
and (*haspartner*(1,5) implies *haspartner*(5,1))
and (*haspartner*(1,6) implies *haspartner*(6,1)))

F_3 *(satisf. of responders)* The level of satisfaction of responders always increases:
A[{not *w_decr*} U FINAL]

F_4 *(satisf. of proposers)* The level of satisfaction of proposers always increases:
A[{not *m_decr*} U FINAL]

We performed the analysis for the three proposed solutions on the whole input space using a machine with an Intel Core i5 2.6 GHz, 8GB RAM, running OS X and UMC v4.4. For the classical case, we considered problems of 3 (i.e., three proposers and three responders). For the attribute-based variants we considered problems of size 2, where each person has four attributes (two for expressing their preferences about partners, and two for modelling their features), each having two possible values. The results of our verification are reported in Table 1. A [✓] means that the formula is satisfied by all possible inputs, while a [×] means that the formula does not hold for at least one input.

By looking at these results we can attempt some considerations:

[6] FINAL is a shortcut for "not EX {true} true"

property	F_1	F_{2a}	F_{2b}	F_{2c}	F_3	F_4
Classical	✓	✓	✓	✓	✓	×
Top-down	✓	✓	×	✓	✓	×
Bottom-up	×	×	×	✓	×	×

Table 1. Verification results of three algorithmic solutions

Classical. Formulae $F_1, F_{2a}, F_{2b}, F_{2c}$ do hold, confirming that the classical algorithm always returns a unique and complete matching. The fact that formula F_3 holds while F_4 does not hold further reflects that a responder keeps trading up its partners for better ones, while proposers can be dropped at any time.

Top-down. Since formula F_{2a} does hold and F_{2b} does not, we can conclude that the top-down strategy will in general return multiple but complete matchings. This is not surprising, since attribute-based stable marriage is a general case of stable marriage with ties and incomplete list (SMTI) [11], and it is known that one instance of SMTI may have multiple matchings [23]. When verifying F_3 and F_4, we obtain the same results of the classical case.

Bottom-up. F_1 does not hold indicating that this approach is not guaranteed to converge. This happens in any configuration containing a cycle in the preferences which makes partners chasing each other. Formula F_{2b} does not hold because there might be two proposers competing for one responder w.r.t. their lowest requirements, thus one of the two remains single. We also verified that both formulas F_3, and F_4 do not hold. This reflects that the satisfaction levels of components may decrease because partners from both sides may drop them for better ones at any moment.

Solution-dependent properties. In addition to previous properties, we also considered a few protocol-related properties to increase to double check the correctness of the specifications derived from informal requirements.

In particular, we verified the following property of the classical solution:

F_5. After a proposer receives a *no*, it will eventually send a new proposal[7]:
AG ([received($1,*no*,*)] AF {send(%1,*propose*)} true)

As expected, UMC answered true when verifying F_5. This guarantees that the proposer will send a proposal again, thus confirming that our specification in that regard meets the informal requirements.

As we have specified a communication protocol for matching entities, the following properties of the top-down solution are important to determine whether the implementation conforms to the requirements:

[7] $*id* and %*id* are used to match the identities of the sending and receiving components.

F_6. If a proposer receives a *bye*, it will always eventually send a new proposal:
$\mathrm{AG}([\mathrm{received}(\$1, \mathit{bye}, *)]\, \mathrm{AF}\{\mathrm{send}(\%1, \mathit{propose})\}\, \mathrm{true})$

F_7. If a responder sends *yes* it will eventually receive a *toolate* or *confirm*:
$\mathrm{AG}[\mathrm{send}(\$1, \mathit{yes})]\, \mathrm{AF}\{\mathrm{received}(\%1, \mathit{confirm}, *)\ \mathrm{or}\ \mathrm{received}(\%1, \mathit{toolate}, *)\}$

F_8. After sending a proposal an initiator does not send further proposals until it receives a *no*:
$\mathrm{AG}[\mathrm{send}(\$1, \mathit{propose})]\, \mathrm{A}[\{\mathit{not}\ \mathrm{send}(\%1, \mathit{propose})\}\, \mathrm{W}\{\mathrm{received}(\%1, \mathit{no}, *)\}]$

By verifying the above properties, we have found out that F_7 holds, while F_6 and F_8 do not. Formula F_8 can be false, because, after sending a proposal, an initiator may receive a *yes*, and then a *bye* message which forces it to send a new proposal. F_6 does not always hold because a initiator after receiving a *bye* message from his partner, may immediately receive a *yes* message from another responder. In this case, it can confirm the new responder without the needs of sending new proposals.

Notice that the informal description of the top-down strategy is not quite rigorous. We have two statements somewhat in contrast, one statement saying that after a *yes* an initiator without a partner should send a *confirm*, and another statement saying that after a *bye* an initiator should send a new proposal. When a *bye* and a *confirm* arrive in sequence, the informal description is not clear in describing the intended behaviour. The formalisation of this requirement in terms of a logical formula, its verification w.r.t. the formal specification of the system, and the observation of the generated counter-example has allowed us to detect and understand this kind of ambiguities.

State Space. Among others, the top-down solution requires the largest number of states with almost 18 millions in the worst case, compared with 0.5 and 4 millions states of the classical and of the bottom-up solution, respectively.

One of the main reasons for this is in the different size of the input space. The attribute-based variant of stable marriage used four attributes with two possible values for each, the space of problems of size 2 has $16^4 = 65536$ configurations. In the classical solution, each agent is characterised by its preference list and thus the space of problems of size 3 only has $3^6 = 729$ configurations.

The complexity of the top-down specification is also a reason for its state explosion, which stems from the use of attribute-based send. In fact, initiators and responders consist of parallel components performing more actions than their classical counterparts: after sending a proposal message, a proposer needs extra acknowledgment messages for selecting his partner. This greatly increases the interleaving of actions by the sub-processes of the components and thus the state space.

5 Concluding Remarks

We have presented a model-checking approach to the verification of attribute-based communication systems. Starting from informal requirements, we have devised formal specifications in AbC. We have then shown how to systematically translate these into verifiable models accepted by UMC. We have exploited the approach for analysis of an algorithmic solution to the classical stable matching problem, as well as for two variants that extend the problem by introducing attribute-based communication among components. We have considered a set of interesting properties for the above programs and described them first informally and then as explicit properties of the generated models.

The results of our experiments have shown that systems relying on attribute-based communications can be particularly complex to design and analyse. However, by exhaustively verifying a specification over all possible inputs, despite the small size of the problem considered, we have experienced that many non-trivial emerging properties and potential problems can indeed be discovered by following our methodology. This confirms once more that concurrency bugs can be detected by only considering a very small number of processes [22].

Experiments with different implementations of SMP in AErlang, an attribute-based extension of Erlang, have been presented in [11]. Also in some previous work [10], we modeled and verified an example instance of stable marriage using attributes. However there the translation was done manually and the verification considered only one configuration.

The analysis of concurrent systems modelled by process algebras has been thoroughly investigated in [18] by relying on powerful abstractions techniques. Other research groups [24,26,13] have taken an approach similar to ours and perform verification by translating a specification formalism into a verifiable one that could make use of existing model checkers.

Techniques for constructing a model for stable marriage and analyzing its convergence has been presented in [7]. There, the authors encoded classical SMP in a DTMC model and analyzed it with the tools provided by PRISM to study different instances of stochastic matching markets.

There are interesting future directions for this work. An extensive experimentation with additional case studies would certainly contribute to refine our approach [21]. Extending AbC with new constructs to model the spatial and mobility aspects of components would allow handling larger classes of systems [12]. Extending our verification approach to quantitative reasoning will improve usefulness, while investigating state reduction techniques will improve tractability.

References

1. AbC2UMC. `http://github.com/ArBITRAL/AbC2UMC`
2. UMC. `http://fmt.isti.cnr.it/umc`
3. UMC Docs. `http://fmt.isti.cnr.it/umc/DOCS`
4. Alrahman, Y.A., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 1–18. Springer (2016)
5. Alrahman, Y.A., De Nicola, R., Loreti, M.: Programming of CAS systems by relying on attribute-based communication. In: International Symposium on Leveraging Applications of Formal Methods. pp. 539–553. Springer (2016)

6. ter Beek, M.H., Gnesi, S., Mazzanti, F.: From EU projects to a family of model checkers. In: Software, Services, and Systems, volume 8950 of LNCS. pp. 312–328. Springer (2015)
7. Biró, P., Norman, G.: Analysis of stochastic matching markets. International Journal of Game Theory 42(4), 1021–1040 (2013)
8. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks 14, 25–59 (1987), `https://doi.org/10.1016/0169-7552(87)90085-7`
9. Brinksma, E.: On the design of extended LOTOS. Doctoral Dissertation, Univ. of Twente (1988)
10. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: AErlang at Work. In: International Conference on Current Trends in Theory and Practice of Informatics. pp. 485–497. Springer (2017)
11. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: Aerlang: Empowering erlang with attribute-based communication. In: International Conference of Coordination Models and Languages. pp. 21–39 (2017)
12. De Nicola, R., Gorla, D., Pugliese, R.: On the expressive power of klaim-based calculi. Theoretical Computer Science 356(3), 387–421 (2006)
13. De Nicola, R., Lluch-Lafuente, A., Loreti, M., Morichetta, A., Pugliese, R., Senni, V., Tiezzi, F.: Programming and verifying component ensembles. In: Joint European Conferences on Theory and Practice of Software. pp. 69–83. Springer (2014)
14. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (1995), `http://doi.acm.org/10.1145/201019.201032`
15. Ed Brinksma, Giuseppe Scollo, C.S.: Lotos specifications, their implementations and their tests. In: Proc. IFIP WG6.1, Protocol Specification, Testing, and Verification VI. pp. 349–360 (1987)
16. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A logical verification methodology for service-oriented computing. ACM Transactions on Software Engineering and Methodology (TOSEM) (2012)
17. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly 69(1), 9–15 (1962)
18. Groote, J.F., Reniers, M.A.: Algebraic process verification. Eindhoven University of Technology, Department of Mathematics and Computing Science (2000)
19. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc. (1985)
20. H.ter Beek, M., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming 76(2), 119–135 (2011)
21. Kümmel, M., Busch, F., Wang, D.Z.: Taxi dispatching and stable marriage. Procedia Computer Science 83, 163–170 (2016)
22. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ACM Sigplan Notices. vol. 43, pp. 329–339. ACM (2008)
23. Manlove, D.F., Irving, R.W., Iwama, K., Miyazaki, S., Morita, Y.: Hard variants of stable marriage. Theoretical Computer Science 276(1-2), 261–279 (2002)
24. Mateescu, R., Salaün, G.: Translating pi-calculus into lotos nt. In: International Conference on Integrated Formal Methods. pp. 229–244. Springer (2010)
25. Milner, R.: A Calculus of Communicating Systems., Lecture Notes in Computer Science, vol. 92. Springer–Verlag (1980)
26. Song, H., Compton, K.J.: Verifying π-calculus processes by promela translation. Technical Report CSE-TR-472-03 (2003)