Verifying AbC Specifications via Emulation

Rocco De Nicola[1](✉), Tan Duong[1](✉), and Omar Inverso[2]

[1] IMT School for Advanced Studies, Lucca, Italy
rocco.denicola@imtlucca.it, tan.duong@imtlucca.it
[2] Gran Sasso Science Institute, L'Aquila, Italy
omar.inverso@gssi.it

Abstract. We propose a methodology for verifying specifications written in *AbC*, a process calculus for collective systems with a novel communication mechanism relying on predicates over attributes exposed by the components. We emulate the execution of *AbC* actions and the operators that compose them as programs where guarded sequential functions are non-deterministically invoked; specifically, we translate *AbC* specifications into sequential C programs. This enables us to use state-of-the-art bounded model checkers for verifying properties of *AbC* systems. To vindicate our approach, we consider different case studies from different areas and model them as *AbC* systems, then we translate these *AbC* specifications into C and instrument the resulting program for verification, finally we perform actual verification of properties of interest.

Keywords: Attribute-based communication · Formal analysis · Bounded model checking

1 Introduction

Collective adaptive systems (CAS) [1] are typically characterised by a massive number of interacting components and by the absence of central control. Examples of these systems can often be found in many natural and artificial systems, from biological systems to smart cities. Guaranteeing the correctness of such systems is very difficult due to the dynamic changes in the operating environment and the delay or loss of messages that at any time may trigger unexpected behaviour. Due to interleaving, explicit-state analysis and testing may not always be appropriate for studying these systems. Rigorous theories, methods, techniques, and tools are being developed to formally reason about their chaotic behaviour and verifying their emergent properties [2].

Process calculi, traditionally seen as paradigms for studying foundational aspects of a particular domain have also been used as specification languages [3]. This is due to their well-specified operational semantics, which enables formal verification and compact descriptions of systems under consideration. On

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

the other hand, due to minimalism in their designs, encoding complex agents behaviour could be tedious. Our work aims at offering a tool for studying CAS by using AbC [4], a kernel calculus centered on attribute-based communication [5], specifically designed to model complex interactions in collective systems. AbC takes into account the runtime status of the components while forming communication groups. Each component is equipped with a set of attributes whose values may be affected by the communication actions. Components interact according to their mutual interests, specified as predicates over the attributes of the potential communication partners. This way, complex interactions can be expressed in a natural way. The AbC communication model is defined on top of broadcast following a style similar to [6]. A distinctive feature is that only components whose attributes satisfy the predicate of a sender can receive the communicating message, provided that they are willing to do so and that the sender attributes also satisfy the predicates specified by the receiving components.

In this paper, we show how AbC can be used to specify different kinds of systems and how some of their emergent properties can be verified. Specifically, we translate AbC specifications into C programs, instrument the latter with properties of interest, and finally analyse the programs by means of classical verifiers for C.

Our translation turns AbC actions into a set of emulation functions guarded by appropriate enabling conditions that, when satisfied, make it possible for the functions to be non-deterministically executed. Since the actions are originally composed by process-algebraic operators, our translation works out such enabling conditions by taking into account the semantics of process operators, and possibly specific code extracted from the actions themselves. A main function plays the role of a scheduler that orchestrates the non-deterministic invocation of the emulation functions. The evolution of the original system is emulated according to the *lock-step* semantics of AbC, which in turn is modelled in the form of a loop. The encoding at each emulation step allows for non-deterministic selection of a single component such that any of its output actions is able to initiate a multi-party synchronization.

Having obtained the C programs for a set of systems from different contexts, we manually annotate them with appropriate assertions that encode properties of interest in the form of expressions over the attributes of the components. We then focus on under-approximate analysis, exploiting mature bounded model checkers, traditionally used for bug hunting in C programs [7,8], to analyse the translated AbC specifications. We have implemented a tool to automatically translate all the examples in this paper. The translator, the source AbC specifications, and the target C programs are available at http://github.com/ArBITRAL/AbC2C.

The rest of the paper is organised as follows. In Sect. 2 we briefly present the fragment of the AbC calculus that we consider in the rest of the paper, present our translation from AbC to C, and describe how to instrument state-based properties of interest. In Sect. 3 we use AbC to model a number of systems borrowed from classical papers in the literature and show how some of their key properties can be verified by means of two bounded model checkers for C. The final section recaps the work done, draws some conclusions, briefly describes related works and suggests directions for future research.

2 Translating AbC into C

Before going into the details of the translation, we briefly review the syntax and semantics of *AbC* with the help of a running example, namely the two-phase commit protocol [9]. The reader is referred to [4,10] for the full description of the calculus.

2.1 AbC in a Nutshell

An AbC *component* (C) is either a pair $\Gamma : P$, where Γ is an attribute environment and P a process, or the parallel composition $C_1 \parallel C_2$ of two components. The environment Γ is a mapping from a set of attribute names $a \in \mathcal{A}$ to ground values $v \in \mathcal{V}$; $\Gamma(a)$ indicates the value of a in Γ.

Let us now consider the two-phase commit protocol (2PC), where a *manager* asks a number of participants for their votes on either "commit" or "abort" a transaction. The transaction is aborted if at least one *participant* votes "abort", and is committed in case all participants vote "commit".

In *AbC* we can model the manager as a component $M \triangleq \Gamma_m : P_m$. The attribute environment Γ_m has three attributes: `role`, initially set to 1 and representing the manager role in the protocol; `n`, initially set to the number of participants; `c`, a counter used for counting participant votes, initially set to 0. Each participant is modelled by a component $C_j \triangleq \Gamma_j : P_p$ and three attributes: `role`, initially set to 0 representing the participant role, `vote` specifying the vote, i.e., "commit" or "abort" casted by the participant, and `d`, used for storing the final decision sent by the manager. A scenario consisting of n participants and one manager is then rendered as an *AbC* system: $M \parallel C_1 \parallel \ldots \parallel C_n$.

In the general case, an *AbC* process (P) is defined by the following grammar that involves standard constructors such as inactive process ($\mathtt{0}$), action prefixing (.), non-deterministic choice (+), interleaving (|), and invocation of a unique process identifier K.

$$P ::= \mathtt{0} \mid Q \mid P + P \mid P|P \mid K \quad Q ::= \langle \Pi \rangle P \mid \alpha.P \mid \alpha.[\tilde{a} := \tilde{E}]P \quad \alpha ::= \Pi(\tilde{x}) \mid (\tilde{E})@\Pi$$

We use $\tilde{\cdot}$ to denote a finite sequence whose length is not relevant. Process P in the construct $\langle \Pi \rangle P$ is blocked until the awareness predicate Π is satisfied within the local environment. In a prefixed process, an action α may be associated with an attribute update $[\tilde{a} := \tilde{E}]$ that, when the action is executed, sets the values of attributes $\tilde{a}$ to that of expressions $\tilde{E}$. Output action $(\tilde{E})@\Pi$ is used to send the values of expressions $\tilde{E}$ to all components whose attributes satisfy the sending predicate Π. Input action $\Pi(\tilde{x})$ is used to receive a message (to be bound to $\tilde{x}$) from any sending component whose attributes (and the message) satisfy the receiving predicate Π.

A predicate Π can be either *true*, a comparison $\bowtie$ (e.g., $<, >, =, \leq, \ldots$) on two expressions E, a logical conjunction of two predicates, or the negation of a predicate. Other logical connectives may be built from these. An expression E is either a value v, a variable x, an attribute name a or an attribute of the current component *this*.a. A component must use *this*.a in its communication predicates (sending or receiving) for distinguishing its own attributes from other different components.

$$\Pi ::= \mathit{true} \mid E \bowtie E \mid \Pi \land \Pi \mid \neg \Pi \qquad E ::= v \mid x \mid a \mid \mathit{this}.a \mid \ \dots$$

Larger expressions can also be built using binary operators such as $+, -$, etc. Expressions E can be evaluated under Γ $\{\!|E|\!\}_\Gamma$. The satisfaction relation $\models$ defines when a predicate Π is satisfied in an environment Γ.

$$\begin{array}{ll} \Gamma \models \mathit{true} \text{ for all } \Gamma & \Gamma \models \Pi_1 \land \Pi_2 \text{ if } \Gamma \models \Pi_1 \text{ and } \Gamma \models \Pi_2 \\ \Gamma \models E_1 \bowtie E_2 \text{ if } \{\!|E_1|\!\}_\Gamma \bowtie \{\!|E_2|\!\}_\Gamma & \Gamma \models \neg\Pi \text{ if not } \Gamma \models \Pi \end{array}$$

Continuing with our example, the behaviour P_m of the manager component is specified as: $P_m \triangleq A|B$, where $A \triangleq (\text{“}\mathit{req}\text{”})@(\mathtt{role} = 0).\mathsf{0}$ and

$$\begin{aligned} B \triangleq \quad & \langle \mathtt{c} < \mathtt{n} \rangle (x = \text{“}\mathit{commit}\text{”})(x).[\mathtt{c} := \mathtt{c} + 1]B \\ & + \langle \mathtt{c} = \mathtt{n} \rangle (\text{“}\mathit{commit}\text{”})@(\mathtt{role} = 0).\mathsf{0} \\ & + (x = \text{“}\mathit{abort}\text{”})(x).(\text{“}\mathit{abort}\text{”})@(\mathtt{role} = 0).\mathsf{0}. \end{aligned}$$

Process A sends a request to all participants using the predicate ($\mathtt{role} = 0$), and terminates. Process B is a choice between different behaviours. The first branch stores the number of “commit” votes in $\mathtt{c}$ using recursion. The second branch deals with the case when all votes are of “commit”, i.e., the counter $\mathtt{c}$ is equal to the number of participants $\mathtt{n}$. The third branch enforces early termination: as soon as an “abort” arrives, the manager can send an “abort” message regardless of the other votes.

All participants have the same behaviour P_p which is specified as:

$$\begin{aligned} P_p \triangleq (x = \text{“}\mathit{req}\text{”})(x).((\mathtt{vote})@(\mathtt{role} = 1).(\mathtt{role} = 1)(x).[\mathtt{d} := x]\mathsf{0} \\ + (\mathtt{role} = 1)(x).[\mathtt{d} := x]\mathsf{0}) \end{aligned}$$

Upon receiving a vote request from the manager, a participant faces two possibilities encoded as a choice (+) in the continuation of P_p: it may reply with its `vote` and continue to wait for a final decision to arrive, or it may receive the decision before sending `vote`. Either possibility updates the final decision to attribute `d` and terminates.

Communication. In *AbC*, output actions are non-blocking while input actions must wait to synchronize on available messages. If multiple components are willing to offer output actions at the same time, only one of them is allowed to move. Communication among components takes place in a broadcast fashion: when a component sends a message, all other receiving components take part in the synchronization by checking the satisfactions of both sending and receiving predicates for reception. Components that accept the message evolve together with the sending component. Components who are not offering a successful input action or reject the message stay unchanged.

As a side note, we mention that since *AbC* is an untyped calculus, it is the responsibility of the modeller to specify appropriately attributes values, expressions, and predicates so that their evaluations make sense.

2.2 Emulating *AbC* Systems in C

Our translation takes as input an *AbC* specification, possibly composed of multiple component specifications. Each component specification is in the form $\langle \Gamma_i, P_{init_i}, D_i \rangle$ where Γ_i is the attribute environment, P_{init_i} the component's top-level behaviour, and D_i the set of process definitions. For example, the specification of the manager component illustrated in the previous section would be $\langle \Gamma_m, P_m, \{P_m \triangleq A|B, A \triangleq \ldots, B \triangleq \ldots\}\rangle$.

The translation produces a single C program whose structure follows a predesigned template (Fig. 1). The encoding is parameterized in the total number of components (`N`), and the maximum number of parallel processes, of process definitions, and of input-bindings variables across all component specifications (`P_MAX`, `D_MAX`, and `V_MAX`, respectively). These constants are extracted from the input specification and defined at the beginning of the output program. The components' attributes are represented as global vectors (line 4), so that each component can access the attributes via its index. Note that, as shown in the figure, we encode the values of the attributes as integers. For each component i, we declare a vector of program counters `pc[i]` for keeping track of the executions of the component's actions during the emulation, a vector `bound[i]` for storing inbound messages, and vector `tgt` to store the indexes of potential receivers when a component sends a message (line 7).

For each (specification of) component i, we translate its behaviour, i.e., $\langle P_{init_i}, D_i \rangle$ into a set of emulation functions. In particular, each action α is translated into a uniquely named function, denoted as $\mathtt{Name}_\alpha$, parameterized (among others) with the component index (lines 10–17). The function body is guarded by an enabling condition whereas a return value indicates whether it is executed.

In order to emulate the executions of all functions in the set with respect to P_{init_i}, the translation visits all actions reachable from the process (by using D_i for looking up process code when necessary); while traversing, it calculates for each action α an index $\mathtt{j}_\alpha$, used for accessing program counter $\mathtt{pc}[\mathtt{i}][\mathtt{j}_\alpha]$ and two execution points, namely entry point $\mathtt{en}_\alpha$ and exit point $\mathtt{ex}_\alpha$ used for controlling the action's execution. A guard $\mathtt{pc}[\mathtt{i}][\mathtt{j}_\alpha] == \mathtt{en}_\alpha$, called *entry condition* means that the function body may be executed, among other conditions, if the program counter of α satisfies such condition. At the end of the function, the program counter is set to $\mathtt{ex}_\alpha$, i.e., its *exit condition*, to enable the next set of feasible actions. Intuitively, the entry and exit conditions of the translated functions must behave according to the intended behavior of the corresponding process operators $(., +, |)$. For example, in a prefix process $\alpha.P$, the exit point of α must be equal to the entry point of the continuation process P. In a choice process $P_1 + P_2$, the entry points of both P_1 and P_2 must be the same but those of their continuations are not. In a parallel process $P_1|P_2$, the entry conditions of the subprocesses must be independent. For additional details on how to determine entry and exit conditions, we refer the reader to [11].

Whether or not an action can really be executed, however, does not depend only on the guarding mechanisms just described, but it depends also on action-

```
#define ...  // define key constants N,P_MAX,D_MAX,V_MAX

... //other declarations
int attr$_1$[N]; int attr$_2$[N]; ... ; // attributes

/* program counter, input-binding variables, receiving components */
int pc[N][P_MAX]; int bound[N][D_MAX][V_MAX]; int tgt[N];

/* A function $Name_{\alpha}$ emulates action $\alpha$ of component i */
int $Name_{\alpha}$(int i, ...) {
   if ($pc[i][j_{\alpha}] == en_{\alpha}$ && ...) {
       ... // action code
       $pc[i][j_{\alpha}] = ex_{\alpha}$;
       return 1;
    }
    return 0;
}
... // all other actions

/* lookup table for the translated input and output actions */
struct {...} lookup[N];

/* initializations for attributes environments and lookup table */
void init() { ... } ;

/* Driver functions  */
int Schedule() { //non deterministic scheduling
  int i = nondet_int();
  assume(i >= 0 && i < N);
  return i;
};

int Evolve(int i) { // lock-step evolution
  int ns = ...; // lookup for the number of output actions of i
  pts* sa = ...; // lookup for the vector of output actions of i
  // nondeterministically selects an index
  int k = nondet_int();
  assume(k >= 0 && k < ns);
  return (sa[k])(i,1); // executes the kth output action of i
};

int Available() {...};   //check the existence of enabled output actions

void Sync(int i, int *m); {...}; // multi-party synchronization

/* Emulation loop */
int main() {
   int i;       // selected component index
   init();
   while (Available()) {   //there exists enabled output actions
       i = Schedule();      // choose one component
       assume(Evolve(i));  // perform lock-step evolution, assuming i sending
   }
   return 0;
}
```

Fig. 1. Structure of the output C program.

specific aspects, such as satisfactions of awareness and receiving predicates. In fact, the emulation function will have different input parameters and body, depending on whether the action being encoded is an input or an output action. More on this will be explained later.

All the emulation functions are organized in a `lookup` table to conveniently invoke them using the component index. Intuitively, each entry i in the table contains, among others, the sets of (pointers to the emulation functions of) input and output actions of component i. The `init()` function (line 24) is responsible for initializing all attribute environments by translating $\Gamma_0, \Gamma_1, \ldots$ and for filling up the `lookup` table.

Towards the end of the output program, we provide several fixed driver functions whose functionalities are as follows.

- `Schedule()`: non-deterministically selects a component index and returns it;
- `Evolve(i)`: non-deterministically selects an output action of component `i` and returns the result of performing the action;
- `Available()`: checks whether there exists an enabled output action;
- `Sync(i,m)`: delivers message `m` of component `i` to potential receiving components (used by output actions).

The source of non-determinism in an *AbC* system is due to non-deterministic choice and to the possibility that different components or different processes within a component perform output actions. To model such non-determinism in the target program, we rely on the common library functions supported by C verifiers: i) `nondet_int` to choose a non-deterministic value from the `int` data type; and ii) `assume` to restrict non-deterministic choices made by the program according to a given condition. We use these primitives in `Schedule()`, as shown in lines 27–31. The implementation of `Evolve(i)` additionally relies on `lookup` to non-deterministically execute an output action of component `i` (lines 33–40). Other driver functions do not consider nondeterminism; their implementation (omitted for brevity) is straightforward by relying on `lookup` and taking advantage of the fact that the number of components `N` is known.

Using the above functions, we emulate the evolution of the translated system through a loop, as shown in lines 50–53. Since an *AbC* system can only evolve when there are components willing to send, the loop iterations are guarded by `Available()`.

At each iteration, an index `i` is selected by `Schedule` and passed to `Evolve` which in turn performs actual computation. The output action called by `Evolve` relies on `Sync` for sending its message, allowing multi-party synchronization in one pass. Moreover, the emulation considers only non-deterministic choices over component index and the corresponding output action that results in valid computations, i.e., the selected output action can actually be executed. This is achieved by wrapping `Evolve` in the library function `assume` (line 52). Our scheduling mechanism based on the idea of non-deterministically selecting the emulation functions has been inspired from [12].

In the rest of the section, we describe the translation for input and output actions in detail. Note that an action may be associated with a preceding awareness predicate and a following attribute update. For example, process B of the component manager (Sect. 2.1) contains such an action. Thus we consider the general form of an action α to be $\langle \Pi \rangle \alpha.[\tilde{a} := \tilde{E}]$. If no awareness construct $\langle \Pi \rangle$ is present, we just consider $\langle \Pi \rangle = true$. If no attribute updates occurs, $[\tilde{a} := \tilde{E}]$ is regarded as an empty assignment. In the following, we write Π_g, Π_s, Π_r to differentiate between the three kinds of predicates: awareness, sending and receiving, respectively.

An output action $\langle \Pi_g \rangle (\tilde{E})@\Pi_s.[\tilde{a} := \tilde{E}]$, where the first $\tilde{E}$ is the expression(s) to be sent, is translated into a function depicted in Fig. 2 (left).

```
int $Name_\alpha$(int i, int f) {
 if (!f) return (pc[i][$j_\alpha$] == $en_\alpha$ && $\llbracket \Pi_g \rrbracket$);
 if (pc[i][$j_\alpha$] == $en_\alpha$ && $\llbracket \Pi_a \rrbracket$) {
    for (int j = 0; j < N; j++)
       if (j != i && $\lceil\!\lceil \Pi_s \rceil\!\rceil$) tgt[j] = 1;
       else tgt[j] = 0;
    int m[$|\tilde{E}|$];  // a vector of size $|\tilde{E}|$
    m[0] = $\llbracket E_0 \rrbracket$; ...
    Sync(i,m); // sending
    $\llbracket \tilde{a} := \tilde{E} \rrbracket$; // attr update
    pc[i][$j_\alpha$] = $ex_\alpha$;
    return 1;}
 return 0;}
```

```
int $Name_\alpha$(int i, int j, int* m) {
  if (pc[i][$j_\alpha$] == $en_\alpha$ && $\llbracket \Pi_g \rrbracket$ && $\lceil\!\lceil \Pi_r \rceil\!\rceil^{\tilde{x}}$) {
  // receive
  bound[i][$d$][$j_{x_0}$] = m[0]; ...
  //--- attr update ---
  $\llbracket \tilde{a} := \tilde{E} \rrbracket$;
  pc[i][$j_\alpha$] = $ex_\alpha$;
  return 1;}
 return 0;}
```

Fig. 2. The translation of output (left) and input (right)

The translation generates a unique identifier $\mathtt{Name}_\alpha$ for the function. The two function parameters are the index `i` of the component containing α, and a flag `f` used to check the enabled condition of the action/function without actually executing it. This flag is specifically used by the driver function `Available`() mentioned before.

The enabling condition of an output action, besides the guard on its program counter, includes the satisfaction of the associated awareness predicate, if any. Note that we use two auxiliary translations $\llbracket \cdot \rrbracket$ for translating Π_g and local expressions and $\lceil\!\lceil \cdot \rceil\!\rceil$ for translating Π_s. In the function body, the set `tgt` of potential receivers is calculated based on the satisfaction of Π_s. Since all components' attributes are globally declared in the emulation program, the output action can evaluate its sending predicate immediately.

After that, a message is prepared as a vector `m` that contains the values of output expressions, and sent via driver function `Sync`. This function retrieves the set of potential receivers in `tgt` and invokes their receiving functions. `Sync` stops delivering `m` to a potential component when one of its receiving functions returns success (i.e., the message is accepted), or none of them do (i.e., the potential component rejects the message). The translation is completed by a sequence of assignments to model attribute updates, if any.

An input action, whose general form is $\langle \Pi_g \rangle \Pi_r(\tilde{x}).[\tilde{a} := \tilde{E}]$, is translated into a function depicted in Fig. 2 (right).

The function takes as input the second argument as sending component index and the third as a communicated message. The enabling condition for executing this function includes satisfaction of awareness and receiving predicates. Here we denote by $\llbracket \Pi_r \rrbracket^{\tilde{x}}$ the translation of Π_r parameterized with input-binding variables $\tilde{x}$. In order to model variable binding, we store the inbound message `m` in vector `bound[i][d]` where d is (the index of) the *process definition* that contains α. Moreover, all input-binding variables y in d are also indexed according to their names, denoted by j_y. In this way, the message is stored by assigning each of its element k to $\texttt{bound[i][d][j}_{\texttt{x}_\texttt{k}}\texttt{]}$, where x_k is the k^{th} variable in the sequence $\tilde{x}$.

In the above translation, we have used $\llfloor \cdot \rrfloor$ for translating awareness predicates and local expressions. This function is defined as follows:

$$\begin{array}{ll} \llfloor \Pi_1 \wedge \Pi_2 \rrfloor = \llfloor \Pi_1 \rrfloor \; \&\& \; \llfloor \Pi_2 \rrfloor & \llfloor a \rrfloor = \texttt{attr[i]} \\ \llfloor \neg \Pi \rrfloor = \; ! \; \llfloor \Pi \rrfloor & \llfloor this.a \rrfloor = \texttt{attr[i]} \\ \llfloor true \rrfloor = \texttt{true} & \llfloor v \rrfloor = \texttt{v} \\ \llfloor E_1 \bowtie E_2 \rrfloor = \llfloor E_1 \rrfloor \bowtie \llfloor E_2 \rrfloor & \llfloor x \rrfloor = \texttt{bound[i][d][j}_\texttt{x}\texttt{]} \end{array}$$

The functions $\llceil \cdot \rrceil$ and $\llceil \cdot \rrceil^{\tilde{x}}$ are used for translating the sending and receiving predicates, respectively. They have the same definition as above, except when translating attributes and variables:

$$\llceil a \rrceil = \llceil a \rrceil^{\tilde{x}} = \texttt{a[j]} \qquad \llceil this.a \rrceil = \llceil this.a \rrceil^{\tilde{x}} = \texttt{attr[i]}$$

$$\llceil y \rrceil^{\tilde{x}} = \begin{cases} \texttt{m[k]} \text{ if } y \in \tilde{x} \text{ and } y = x_k \\ \texttt{bound[i][d][j}_\texttt{y}\texttt{]} \text{ otherwise} \end{cases}$$

Thus, $\llfloor \cdot \rrfloor$ does not differentiate between a and $this.a$ whereas $\llceil \cdot \rrceil$ and $\llceil \cdot \rrceil^{\tilde{x}}$ do. In $\llceil y \rrceil^{\tilde{x}}$, if a variable appears in the list of input-binding variables then it is translated into the corresponding element of the communicated message. In all other cases, the variable is already bound and its value can be looked up in vector `bound[i][d]`.

2.3 Encoding Properties

The evolution of an *AbC* system over time can be viewed as a tree rooted at the initial state, i.e., the union of all the components' initial states. An edge from a node to a child represents a lock-step evolution in which a component sends and others receive (hence, changing the overall system state). From a node there may be multiple edges, each corresponding to a synchronization initiated by a non-deterministically selected sending component.

Typically, program verifiers do not allow to directly express temporal logics for specifying properties; users must use `assert` statements to check that their intentions hold. We use assertions to express state-based formulae, e.g., conditions over the components' attributes. In practice, we encode such a formula as a boolean-valued C expression, denoted as p, and insert a statement `assert(p)` within the emulation loop of the main function.

Notice that, due to the emulation mechanism explained in previous section, the C program emulates all possible execution paths of the translated *AbC* system. This means that checking the assertions at every iteration of the emulation loop in the C program corresponds to checking whether p holds in (every state of) all possible execution traces of the initial AbC system. Naturally, in this way the system is verified against a safety property denoted as $\mathsf{S}\ p$. In this paper, we focus on bounded analysis, i.e., limit the emulation up to a given number B of system evolutions of the AbC system by bounding the number of iterations of the emulation loop. Thus, in practice we only analyse safety up to the given bound. In other words, we deal with bounded safety $(\mathsf{S}\ p)^B$.

For AbC systems one may also be interested in eventual properties, e.g., "good" system states that eventually emerge from the interactions of individual components. Since we focus on bounded analysis, we must resort to a bounded variant of liveness, and reduce liveness checking to safety checking [13] by expressing the former via the latter. We consider three (bounded) versions of eventuality and describe their encodings in the following.

"Possibly" - $(\mathsf{P}\ p)^B$ is encoded as $(\mathsf{S}\ \neg p)^B$: we simply assert the negation of p inside the emulation loop, and wait for the model checker to report a counter example (of $\neg p$) that would be the evidence that there is at least one execution trace that satisfies p.

"Eventually" - $(\mathsf{E}\ p)^B$. We translate this property into safety by following the idea of [13], the intuition being that if the formula p fails, it must do so within B steps. Thus, when the bound is reached, we need to assert whether p has been true at least once. Our encoding is illustrated in the left of Fig. 3. A variable `step` counts the number of steps performed by the system up to that point. In each step of the emulation, the truth value of p is accumulated into a boolean variable `live` via alternation. The `assert` statement checks the value of `live` when the emulation stops, i.e., either the bound B is reached or there are no available sending components.

```
int step = 0;
_Bool aa = Available();
_Bool live = false;
while (aa) {
   step = step + 1;
   ... //AbC system's evolution
   live = live || p();
   aa = Available();
   assert((!(step == B) && aa) || live);
}
```

```
int step = 0;
_Bool aa = Available();
_Bool live = false, saved = false;
while (aa) {
    step = step + 1;
    ... // Abc system's evolution
    live = saved?(live && p()):(live || p());
    if (!saved && live) saved = true;
    aa = Available();
    assert((!(step == B) && aa)|| live);
}
```

Fig. 3. Encoding of $(\mathsf{E}\ p)^B$ (left), and $(\mathsf{EI}\ p))^B$ (right)

"Eventually then Inevitably" - $(\mathsf{EI}\ p)^B$. Compared to "Eventually", this property further requires that once p holds, it remains true afterward. The encoding is shown on the right of Fig. 3 where we extend the previous code snippet to enforce this additional requirement. Specifically, we use a variable `saved` to record the first time when p holds. When this happens, `live` is conjuncted with p rather than alternating. Because of this p must remain true after `saved` is set, otherwise the property fails.

3 Experimental Evaluation

We now illustrate the effectiveness of the proposed method by considering some case studies and systematically verifying their properties.

3.1 Case Studies

Max-Element. This system is an attribute-based variant of the one presented in [14]. Given N agents, each associated with an integer $\in \{1, \dots N\}$, we wish to find an agent holding the maximum value. This problem can be modeled in *AbC* by using one component type with two attributes, namely `s`, initially set to 1, indicating that the current component is the max, and `n`, that stores the component's value. The behavior of a component is specified by the following choice process P:

$$P \triangleq \langle \mathtt{s} = 1 \rangle (\mathtt{n})@(\mathtt{n} \leq \mathit{this}.\mathtt{n}).\mathsf{0} + (x \geq \mathit{this}.\mathtt{n})(x).[\mathtt{s} := 0]\mathsf{0}.$$

Thus, a component either announces its own value by sending its attribute `n` to all other components with smaller numbers, or non-deterministically accepts a greater value from another. In addition, upon receiving a greater value a component sets its flag `s` to 0 since it can not possibly be the max.

For this system, we are interested in verifying whether eventually there exists only one component whose attribute `s` is equal to 1, and that holds the maximum value of `n`.

Or-Bit. This example and the next one are adapted from [15]. Given a number of agents, where each agent is given an input bit, the agents must collaborate with each other in order to compute the logical disjunction of all their bits. We model this system with an *AbC* component with two attributes `ib` and `ob`, representing the input and output bits, respectively. Initially, `ob` is set to 0, but this may change during interactions. The behaviour of a component is specified by a choice process P:

$$P \triangleq \langle \mathtt{ib} = 1 \rangle (\mathtt{ib})@(\mathtt{ib} = 0).[\mathtt{ob} := 1]\mathsf{0} + \langle \mathtt{ib} = 0 \rangle (x = 1)(x).[\mathtt{ob} := 1]\mathsf{0}.$$

The first branch of P controls the behaviour of components whose input bits is equal to 1. These announce the bits and update their outputs to 1. In other cases, components with 0 input bits keep waiting for any positive bit to arrive in order to update their outputs to 1. For this example, we are interested in checking whether each component correctly calculates (in `ob`) the disjunction of all the bits.

Majority. Given a system composed of *dancers* and *followers*, we would like to determine whether the dancers are the majority, i.e., no less than the number of followers, without any centralized control.

This scenario is rendered in *AbC* by using two types of components, representing dancers and followers respectively. The approach is to design a protocol for matching pairs of dancers and followers. Eventually, by looking at the ones without partners, we can conclude about the majority of one type over the other. For both kinds of components, we introduce an attribute $\mathtt{r}$ for representing the component's role, where value 1 encodes a dancer, an attribute $\mathtt{p}$ indicating if a current component is already paired up with another member of the other role. Furthermore, for dancers that are responsible for initiating the matching, we add an additional attribute $\mathtt{u}$, used as a unique tag when sending messages.

A dancer starts by announcing a unique message (using $\mathtt{u}$) and waits for a follower to show up. The first branch in the choice process D below illustrates this behaviour:

$$D \triangleq (\mathtt{u})@(\mathit{true}).(x = \mathit{this}.\mathtt{u})(x).[\mathtt{p} := 1]\mathsf{0} + (\mathtt{r} = \mathit{this}.\mathtt{r})(x).(y \neq \mathit{this}.\mathtt{u})(y).D.$$

The second branch of D instead models the situation in which the dancer's turn to announce has been preempted by some other dancer, i.e., the input action with a receiving predicate $(\mathtt{r} = \mathit{this}.\mathtt{r})$ is executed. In that case, it has to listen to (and discards silently) a reply from some follower (to the announcer) in order to try again.

Any announcements sent by process D is broadcast but only followers answer. A follower listens to the announcements, i.e., via the predicate $(\mathtt{r} \neq \mathit{this}.\mathtt{r})$, then it either replies to the sender using the sender's message tag or silently discards a message from some other follower (who has replied to this sender before it). The process F defined below captures the described intuition:

$$F \triangleq (\mathtt{r} \neq \mathit{this}.\mathtt{r})(x).((x)@(\mathit{true}).[\mathtt{p} := 1]\mathsf{0} + (y = x \land \mathtt{r} = \mathit{this}.\mathtt{r})(y).F).$$

For this case study, we check for the majority of dancers by asserting that either there is at least one dancer left without partner or everyone has a partner.

Two-Phase Commit. This example has already been described in Sect. 2. The property of interest is that all participants consistently agree on either abort or commit the concerned transaction. In the latter case, we must check that all participants voted for commit.

Debating Philosophers. This example is taken from [16]. A number N of philosophers hold two different opinions on a thesis, possibly against it ($^-$) or in favor of it ($^+$). Each philosopher has also a physical condition, either rested (R) or tired (T). When two philosophers with different opinions debate, a rested philosopher convinces the tired one of his opinion; if the two philosophers are in the same physical condition, the positive one convinces the negative one and both get tired afterwards. On the other hand, philosophers holding the same opinion do not debate.

Each philosopher has the following attributes: attribute $\mathtt{u}$, a unique identifier used for announcement, attribute $\mathtt{o} \in \{0, 1\}$ is the initial opinion with a value 1 indicating positive opinion ($^+$), and attribute $\mathtt{c} \in \{0, 1\}$ represent the initial physical condition with a value 1 denoting rested (R). We design the behaviour for philosophers to interact with each other by a parallel process $P \triangleq F \mid A$. The property of interest is a majority one which states that the number of positive philosophers is not less than that of negative ones. Our protocol is as follows. Any philosopher supporting the thesis repeatedly convince the members of the other group by using process F, specified as:

$$\begin{aligned} F \triangleq \langle \mathsf{o} = 1\rangle(\mathsf{u}, \mathsf{c})@(true).((&this.\mathsf{u} = x \wedge y \neq this.\mathsf{c})(x, y).[\mathsf{o} := \mathsf{c}]F \\ &+ (this.\mathsf{u} = x \wedge y = this.\mathsf{c})(x, y).[\mathsf{c} := 0]F) \\ + \langle \mathsf{o} = 1\rangle(\mathsf{o} = this.\mathsf{o})(x, y).(\mathsf{o} \neq this.\mathsf{o})(x, y).F \end{aligned}$$

Each branch of F is guarded by an awareness predicate $\langle \mathsf{o} = 1\rangle$. In the first branch, a positive philosopher announces its unique message u and the physical condition c. It waits for a message from a negative one by checking on the condition $(this.\mathsf{u} = x)$, and additionally consider two possibilities of the opponent's physical condition y. The following choice process implements concisely the first two debate rules described above for this philosopher where his opinion o or condition c is updated. In the second branch, the philosopher receives an announcement from another positive one, i.e., $(\mathsf{o} = this.\mathsf{o})$. When this happens, it must listen silently until the debate started by the sender finishes.

Philosophers who are against (or negative about) the thesis listen to the opinion of the others and debate according to process A as follows:

$$\begin{aligned} A \triangleq \langle \mathsf{o} = 0\rangle(\mathsf{o} \neq this.\mathsf{o})(x, y).(&\langle y \neq \mathsf{c}\rangle(x, \mathsf{c})@(true).[\mathsf{o} := y]A \\ &+ \langle y = \mathsf{c}\rangle(x, \mathsf{c})@(true).[\mathsf{o} := 1, \mathsf{c} := 0]A \\ &+ (z = x)(z, y).A). \end{aligned}$$

A is guarded by an awareness predicate $\langle \mathsf{o} = 0\rangle$ to limit such behavior to negative philosophers. When a negative philosopher receives an announcement, he may involve into the debate with the sender by replying with the sender's unique value, stored in x and its current physical condition. During this action, changes to opinion and physical conditions (in the form of attribute updates) are implemented by following the first two debate rules. On the other hand, there may happen that another negative one is already engaged into the debate with the same sender, in such case the philosopher waits for another announcement.

Experimental Setup. We developed a translator implementing the method described in Sect. 2 and used it on the AbC specifications of the case studies. We manually instrumented the generated C programs with appropriate assertions to express the properties of interest described above. While instrumenting, we applied the scheme shown in the right of Fig. 3 (Sect. 2.3) for all case studies, i.e., to consider the "Eventually then Inevitability" properties. Additionally, we applied the other two schemes "Eventually" and "Possibly" for the last case study. Hereafter for brevity we refer to the instrumented properties by using their versions, i.e., EI, E and P respectively.

We used two mature bounded model checkers, namely CBMC v5.11[1] and ESBMC v6.2[2] respectively SAT- and SMT-based, for the actual analysis of the

[1] http://www.cprover.org/cbmc/.
[2] https://github.com/esbmc/esbmc/releases/tag/v6.2.

instrumented C programs up to different bounds of the emulation loop. Note that all the other loops in the program (namely, those within the driver functions) are always bounded by constants, and thus are fully unrolled by the model checker in any case.

3.2 Verification Results

The verification results of the three case studies Maximum Element (max), Or Bit (bit) and Two-Phase Commit (2pc) are presented in Fig. 4. The subtable on the left contains the results obtained from CBMC; the right one corresponds to ESBMC. In the tables, we include the numbers of components (N), the unwinding bound for the emulation loop (B), the property, the verification times (in seconds), and the verification result. A [✓] means that the verification succeeds, i.e., when the property holds, otherwise [×].

For each of these case studies, we fix a number N of components and experiment by verifying the property EI of interest while varying B. Figure 4 shows that when B is sufficiently large both verifiers confirm that all considered instances satisfy their properties EI. This means that, within the considered number of evolution steps, the behaviour of the system is guaranteed to preserve the property of interest. Furthermore, we have that once the property is guaranteed within a bound B, it continues to hold with a larger bound. This confirms our intuition on the encoding scheme of property EI.

CBMC	N	B	Property	Time	Result
Max	10	9	EI	12	×
	10	10	EI	15	✓
	10	11	EI	241	✓
Bit	8	7	EI	6	×
	8	8	EI	10	✓
	8	9	EI	12	✓
2pc	7	7	EI	187	×
	7	8	EI	252	✓
	7	9	EI	325	✓

ESBMC	N	B	Property	Time	Result
Max	7	6	EI	46	×
	7	7	EI	106	✓
	7	8	EI	108	✓
Bit	8	7	EI	11	×
	8	8	EI	13	✓
	8	9	EI	15	✓
2pc	5	5	EI	70	×
	5	6	EI	127	✓
	5	7	EI	265	✓

Fig. 4. Experiments with Max Elem, Or Bit and Two Phase Commit

As for the Majority example, we experimented with a few input configurations. A configuration is completely defined as a pair (D,F) indicating the number of dancers and followers. For each considered pair we experiment with varying the bound B and present the relevant results in Fig. 5. The first column of each table represents input configurations whereas other columns have the same meaning as before.

The results show that for configurations with at least as many dancers as followers, the verification of majority returns success across different values of bound B. When considering minority configurations, i.e., (3,4) and (3,5), the verification succeeds with small values of B but fails with larger values. To see why, we note it takes steps to match one dancer with one follower (i.e., one request and one response). Thus, for example, with $B = 4$ the systems can only match two pairs, leaving one dancer un-matched; this results in a majority of dancers. When increasing the bound to at least 6, we have that all three dancers are paired, but the property fails because not everybody has a partner.

CBMC	N	B	Property	Time	Result
(3,5)	8	4	EI	10	✓
(3,5)	8	6	EI	22	×
(3,5)	8	8	EI	41	×
(4,4)	8	6	EI	23	✓
(4,4)	8	8	EI	42	✓
(4,4)	8	10	EI	78	✓
(5,3)	8	6	EI	25	✓
(5,3)	8	8	EI	46	✓
(5,3)	8	10	EI	74	✓

ESBMC	N	B	Property	Time	Result
(3,4)	7	4	EI	23	✓
(3,4)	7	6	EI	75	×
(3,4)	7	8	EI	246	×
(3,3)	6	4	EI	15	✓
(3,3)	6	6	EI	43	✓
(3,3)	6	8	EI	67	✓
(4,3)	7	4	EI	19	✓
(4,3)	7	6	EI	31	✓
(4,3)	7	8	EI	50	✓

Fig. 5. Experiments with Majority example

We use the case study Debating Philosophers to experiment with different property encodings. An input configuration is a pair consisting of the numbers of positive and negative philosophers equipped with their physical conditions (i.e., either R or T). Furthermore, in each element of the pair, we may use a – to separate the rested philosophers from tired ones. We analyzed the properties "Eventually then Inevitably" (EI), "Eventually" (E) and "Possibly" (P) for several configurations, and report some interesting cases in the Fig. 6.

CBMC	N	B	Property	Time	Result
(2R,2R)	4	4	EI	37	✓
(2R,2R)	4	8	EI	165	×
(2R,2R)	4	10	EI	257	×
(1T,6T)	7	6	EI	289	✓
(3R,3R)	6	10	E	605	✓
(2T,3R)	5	8	E	250	×
(1R,2R-4T)	7	4	P	101	✓
(1R,2R-4T)	7	6	P	273	×

ESBMC	N	B	Property	Time	Result
(2R,2R)	4	2	EI	17	✓
(2R,2R)	4	4	EI	122	✓
(2R,2R)	4	6	EI	462	×
(1T,5T)	6	4	EI	359	✓
(2R,2R)	4	10	E	205	✓
(2T,3R)	5	8	E	1120	×
(1R,2R-3T)	6	2	P	27	✓
(1R,2R-3T)	6	4	P	206	×

Fig. 6. Experiments with Philosophers example

When verifying property EI for configuration (2R,2R), we observe that the property only holds for small values of B. Similar to Majority, in this case study the systems need two steps in order to accomplish one debate. Thus, for example, with $B = 4$, there can only happen two debates which results in majority of the positive opinions. However, when we allow for at least three debates to happen, the majority property will not hold any longer. A counter example returned by one of the tools explain the following trace. First, $1R^+$ convinces $1R^-$, which leads to $2T^+$. It is followed that each of $2T^+$ is convinced by the other $1R^-$ to join him. Then, the resulting configuration (1R,1R-2T) does not satisfy majority.

When verifying the property for somewhat trivial configurations, i.e., (1T,5T), (1T,6T) both tools confirm that when the bound B is large enough a single positive philosopher can convince any set of tired negative ones.

We also tried with E property, i.e., allowing the majority property to fail after it became true. Then, the property E holds for a previously failed configuration, in particular, configurations of the same number of positive and negative philosophers (in the Fig. 6 (2R,2R) and (3R,3R)). In another configuration, eventually a number of positive philosophers cannot convince a bigger group of negative philosophers to get to majority of positive opinions.

Finally, we experimented with verifying property P for the configurations of the form (1R, 2R-4T), for CBMC and (1R, 2R-3T) for ESBMC. By checking P, we are interested to see whether the majority can happen by using only one rested, positive philosopher. As shown in Fig. 6, when the bound is too low the only $1\mathrm{R}^+$ does not have enough time to convince the others; the verifications succeed and no counter example is reported. However with the bound increased enough to afford one more debate, the verifications fail. By inspecting the returned counter examples, we observed expected traces in which the only positive philosopher continuously convinces the others without being interfered by (at least) $1R^-$.

In summary, the verifications results obtained from the two model checkers are consistent in all considered instances of all case studies. This demonstrates the feasibility of our approach. In addition, between the two tools CBMC seems a bit more efficient when analyzing our programs.

4 Concluding Remarks

We have presented a translation from AbC process specifications to C programs. The translation enables us to reduce automated checking of (some classes of) properties of interest for the AbC system under consideration to simple reachability queries on the generated C program. To experiment with our method on a series of naturally challenging examples, we have first encoded them as AbC systems, then translated their specifications into C programs, and finally instrumented the programs to express properties of interest of the initial AbC system. We have reported and discussed our experimental evaluation on the automated analysis of such programs via SAT- and SMT-based bounded model checking, under different execution bounds and parameters of the initial system. Our work suggests that the novel communication style of the attribute-based calculus, complemented with appropriate verification techniques, can be effective for studying complex systems.

Our work is closely related to the encoding proposed in [11], which models systems evolutions by entry and exit conditions on the individual actions. In that work, AbC specifications are transformed into doubly-labelled transition systems and then analyzed via the UMC explicit-state model checker [17]. That approach does support a more expressive logic for properties than the current one, that also requires additional effort for instrumenting properties of interest with assertions and for interpreting counter examples associated to C programs. This is less of a burden in [11], since the target formalism has a direct correspondence with the initial specifications. In [11], however, the target representation can grow very quickly in size, for instance to model non-deterministic initialisation of an attribute (e.g., the initial position of an agent on a grid), for which an explicit transition will be added for each possible value of the attribute. In our translation we can introduce a non-deterministic variable (by using the non-deterministic initialisation construct) to represent this symbolically. A similar argument holds for value passing in general. We plan to compare our approach with [11] in terms of efficiency.

Another work closely related to ours is [12]; it uses a mechanism similar to the one used in [11] to guard the actions of a component and to transform the initial system into a sequential C program with a scheduling mechanism similar to the one proposed here. However, the input language of the translation is quite different from AbC, especially for the primitives for components interaction, which are based on stigmergy.

The SCEL language [5] has been the main source of inspiration for AbC. In [18], SCEL specifications are translated into Promela and analyzed by the SPIN model checker to prove safety and liveness properties. Promela has a C-like syntax and supports the dynamic creation of threads and message passing. Because of this, the modelling in [18] is more straightforward than ours. However, in [18] only a fragment of SCEL is considered while we consider full AbC systems; moreover, the program produced by our encoding is sequential, and because of this some ingenuity is required for emulating processes and for encoding properties.

The considered experiments seem to provide some evidence that the proposed encoding favors the use of specific tools. As future work, we would like to reconsider it in order to make the use of other verification back-ends possible. It would be interesting to see whether the SAT-based parallel bounded model checking technique proposed in [19] could be adapted to our case, given the similarity between the programs generated from AbC by our translation and the sequentialised programs considered in [19]. Completeness thresholds for bounded model checking [20] would allow to adapt our technique to unbounded analysis.

We also plan to develop an interactive simulator for AbC to explore specifications through simulations. This would enable one to remove coding errors introduced during the early steps of the design and to gain confidence about specifications before formally verifying them against the properties of interest.

Finally, we would like to use our approach to consider more interesting case studies, e.g., those that involve spatial reasoning, such as flocks. However, to do this, we think that it is important to extend AbC with a notion of globally-shared environment and global awareness, thereby facilitating reasoning about agents locations which is considered as an important feature of CAS [21].

References

1. Anderson, S., Bredeche, N., Eiben, A., Kampis, G., van Steen, M.: Adaptive collective systems: herding black sheep. In: BookSprints for ICT Research (2013)

2. De Nicola, R., Jähnichen, S., Wirsing, M.: Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. **22**(4), 389–397 (2020). https://doi.org/10.1007/s10009-020-00565-0
3. De Nicola, R., Ferrari, G.L., Pugliese, R., Tiezzi, F.: A formal approach to the engineering of domain-specific distributed systems. J. Logic Algebraic Methods Program. **111**, 100511 (2020)
4. Abd Alrahman, Y., De Nicola, R., Loreti, M.: A calculus for collective-adaptive systems and its behavioural theory. Inf. Comput. **268**, 104457 (2019)
5. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014)
6. Ene, C., Muntean, T.: A broadcast-based calculus for communicating systems. In: IPDPS, p. 149. IEEE Computer Society (2001)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
8. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: ASE, pp. 888–891. ACM (2018)
9. Lampson, B., Sturgis, H.E.: Crash recovery in a distributed data storage system (1979). https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/
10. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. **192**, 102428 (2020)
11. De Nicola, R., Duong, T., Inverso, O., Mazzanti, F.: Verifying properties of systems relying on attribute-based communication. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 169–190. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_9
12. De Nicola, R., Di Stefano, L., Inverso, O.: Multi-agent systems with virtual stigmergy. Sci. Comput. Program. **187**, 102345 (2020)
13. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electron. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002)
14. Prasad, K.V.S.: Programming with broadcasts. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 173–187. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_13
15. Aspnes, J., Ruppert, E.: An introduction to population protocols. In: Garbinato, B., Miranda, H., Rodrigues, L. (eds.) Middleware for Network Eccentric and Mobile Applications, pp. 97–120. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-89707-1_5
16. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Verification of population protocols. In: CONCUR. LIPIcs, vol. 42, pp. 470–482. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
17. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Sci. Comput. Program. **76**(2), 119–135 (2011)
18. De Nicola, R., et al.: Programming and verifying component ensembles. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 69–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54848-2_5
19. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: PPoPP, pp. 202–216. ACM (2020)

20. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 557–572. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_44
21. Loreti, M., Hillston, J.: Modelling and analysis of collective adaptive systems with CARMA and its tools. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 83–119. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_4