Model Checking Distributed Systems against Temporal-Epistemic Specifications

Andreas Griesmayer, Alessio Lomuscio

Model Checking Distributed Systems against Temporal-Epistemic Specifications

Andreas Griesmayer[1] and Alessio Lomuscio[2]

[1] ARM, Cambridge, UK
[2] Imperial College London, London, UK

Abstract. Concurrency and message reordering are two main causes for the state-explosion in distributed systems with asynchronous communication. We study this domain by analysing ABS, an executable modelling language for object-based distributed systems and present a symbolic model checking methodology for verifying ABS programs against temporal-epistemic specifications. Specifically, we show how to map an ABS program into an ISPL program for verification with MCMAS, a model checker for multi-agent systems. We present a compiler implementing the formal map, exemplify the methodology on a mesh network use case and report experimental results.

1 Introduction

Significant advances have been made in the development of model checking techniques [5] for a variety of high level programming languages for distributed systems. These techniques support many features of distributed systems including *concurrency*, *modularity* and *object-orientation*. Much less attention has been given so far to the exploration of symbolic verification for *asynchronous message passing* using *futures* [1], a concept supported in modern programming languages like Java, Scala, or the latest C++ standard C++11. By using *futures*, message calls can be made without blocking the caller and messages may be delayed and reach their destinations out of order.

The aim of this paper is to put forward a verification technique targeting this specific class of systems. To make the investigation grounded in a concrete framework we base our analysis on ABS [10], an object-oriented modelling language for distributed systems. Advanced concurrency and synchronisation mechanisms, as well as a formally defined operational semantics and execution environment make ABS an ideal language for the formal modelling and analysis of distributed systems. A significant class of properties of distributed systems concern the *knowledge* that components have about the system they inhabit. To handle these specifications, we adopt techniques from verification of multi-agent systems (MAS). MAS are distributed systems in which the underlying components, or *agents*, interact with one another in order to maximise their own design objectives. MCMAS [15] is a model checker supporting a wide-range of specifications that commonly arise in MAS, including temporal-epistemic ones,

and has been used successfully to verify a number of MAS scenarios, including agent-based web-services [16]. To utilise these techniques we provide a formal map from ABS to MAS, thus lifting the verification of temporal-epistemic MAS specifications to executable models in ABS.

We briefly summarise ABS, MCMAS, and the specification languages employed in the verification of MAS in Section 2. In Section 3 we define a formal map from a subset of ABS into ISPL and discuss how method invocation is handled. Experimental results and the implementation are described in Section 4 before we conclude in Section 5.

1.1 Related Work

While model checking has traditionally targeted the verification of finite systems, more recent work is concerned with constructs for software that lead to undecidable problems [19], including concurrency and recursion. These efforts have led to verification approaches for different programming paradigms like large subsets of C [4], Haskell [21], and Java [22]. Model checking is now routinely applied in well defined domains like driver development [23, 2]. Furthermore, dedicated languages for modelling and verifying component based systems [3], protocols [9], etc., have been developed. Notwithstanding these important contributions, none of these languages provides direct support for asynchronous message passing, a fundamental feature in distributed systems. The semantics of ABS is formally defined and executed by means of the rewriting engine Maude [6], resulting in a well-defined semantics and enabling the user to access Maude's model checking capabilities. However, the interpretation of the semantic rules adds additional complexity and restricts model checking ABS to small sub-goals. By mapping ABS to MCMAS, we avoid some of this complexity and acquire the ability of verifying epistemic specifications.

In the context of distributed systems, traditional languages provide an unnatural transfer of control from the caller to the callee while waiting for the return from a method call. *Future*s [1] provide a fundamentally different control flow model that is centred on asynchronous message passing thereby enabling the calling process to carry on with its activities. This is a model that is much closer to Internet-based distributed systems. In this work we use ABS [10], a modelling language based on objects encapsulating their own processor, to analyse such systems. The models are executable, but because of their complexity, they can currently only be evaluated by using test case generation and simulation [12].

We show an application of the approach by using AODV, a well-understood protocol for ad-hoc message routing [18]. The AODV protocol has been analysed before through other model checkers [20, 8] While these approaches can explore larger configurations than what we present here, their models are crafted manually and not generated automatically from executable programs. Closer to the approach we present here is CMC [17], which directly supports detailed, event-driven implementations in C and can be used to detect safety properties, including receipt of invalid packets. In contrast to these approaches, we here

$$\frac{\overline{v} = [\![\overline{e}]\!]_{(a \circ l)} \;\; fresh(f)}{ob(o,a,\{l \mid x = \hat{o}!m(\overline{e}); s\}, q) \rightarrow ob(o,a,\{l \mid x = f; s\}, q) \; invoc(\hat{o}, f, m, \overline{v}) \; fut(f, \bot)} \;(Asynch\text{-}Call)$$

$$\frac{p' = bind(o, f, m, \overline{v}, class(o))}{ob(o,a,p,q) \; invoc(o, f, m, \overline{v}) \rightarrow ob(o,a,p, q \cup p')} \;(Bind\text{-}Mtd)$$

Fig. 1: Semantics rules for message passing in ABS.

also present an automated translation from executable ABS programs into MCMAS and consider the asynchronous and concurrent nature of the underlying distributed systems. Based on this mapping, we perform symbolic model checking to verify temporal epistemic-properties, which requires the exploration of the full state space, hence inherently more expensive than safety checks.

2 Preliminaries

We give a short introduction to the concepts and techniques for the specification and verification framework of ABS. The semantic of the language is formally defined using Structural Operational Semantics (SOS) rules, which are given in rewrite logic and executable using the rewrite engine Maude [6]. We concentrate on the executable core language [10] with concurrent objects, in particular we give the details on the asynchronous message passing mechanisms. The behaviour of the system is analysed using temporal epistemic logic.

2.1 The Modelling Language ABS

ABS [10] uses objects that maintain local sets of processes to model independent communicating entities. Method calls are not accompanied with transfer of control to the called method. Instead, messages are transferred between caller and callee and a *future variable* serves as placeholder that can be polled at a later point in the execution to obtain the returned values. Messages *received* by the callee trigger the creation of new processes, which are added to its process set. ABS enforces strong data encapsulation, which means that variables can only be accessed locally or via method calls. Furthermore, each object performs local cooperative multitasking controlled by the *await*, *suspend* and *return* statements, where *return* publishes its return values in the *future* and terminates the execution, *suspend* unconditionally suspends the execution and makes an entry in a set of pending processes for a later resume, and *await* evaluates its condition and suspends the process while the result is false.

We follow the operational semantics of [10], given as SOS (Structural Operational Semantics) rules (Fig. 1). A state is represented as a *configuration (cn)*; a set of objects, messages and future variables. An *object* is a term $ob(o, a, p, q)$ where o is the object's identifier, a maps variable names (fields) to values, p is the currently executing process and q a set of suspended processes. A process consists of local variable bindings l and a list of statements s, denoted $\{l \mid s\}$.

The full variable context is denoted as $a \circ l$, where variables in l hide variables in a if they have the same name. An *invocation message* $invoc(o, f, m, \overline{v})$ is sent to the *callee* o when the method m is called with parameters $\overline{v}$. The return message can be accessed via the *future* variable f. Variables can be combined with the usual Boolean and arithmetic operators to form expressions e where $[\![e]\!]_a$ denotes its evaluation with values from a. The details for the semantic rules that describe message passing are shown in Fig. 1 and described in the following:

(Asynch-Call) describes the execution of the statement $x = \hat{o}!m(\overline{e})$, which is on top of the statement list of the active process, followed by the statement list s. The premise of the SOS rule sets up the environment and states, where $\overline{v} = [\![\overline{e}]\!]_{(aol)}$ is the evaluation of the arguments for the call in the current context and f is a fresh variable. In the consequence of the rule we find that the method call is removed from the statement list, and an invocation message $invoc(\hat{o}, f, m, v)$ and an empty future $fut(f, \bot)$ is created and assigned to the fresh variable f.

(Bind-Mtd) formalises the state change of the object receiving the *invoc* message by adding a new process p' to its idle set. $bind(o, f, m, \overline{v}, class(o))$ creates a new process to execute method m from object o with the parameters $\overline{v}$. The future variable f is assigned to a reserved local variable *destiny*, which is accessed later by the return statement.

Similar rules *(Return)* and *(Read-Fut)* define the return message of a call and access to the future variables.

2.2 Temporal-Epistemic Logic and Interpreted Systems

To reason about knowledge, we associate objects in ABS to *agents* in multi-agent systems (MAS). A popular semantics in MAS is that of interpreted systems [7], where agents interact with each other and their environment by means of actions. We adhere to standard naming conventions and characterise each agent $i \in \{1, \ldots, n\}$ in the system by finite sets of local states L_i and local actions Act_i. Actions are performed in compliance with a local protocol $P_i : L_i \to 2^{Act_i}$ specifying which actions may be performed in a given state. The environment in which agents live is modelled by a special agent E with a set of local states L_E, a set of local actions Act_E, and a local protocol P_E. A tuple $g = (l_1, \ldots, l_n, l_E) \in L_1 \times \ldots \times L_n \times L_E$, where $l_i \in L_i$ for each agent i and each $l_E \in L_E$, is a *global state* describing the system at a particular instant of time.

The evolution of the agents' local states is described by a function $t_i : L_i \times L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_i$ which returns the next local state for agent i given the current local state of the agent, the current action and state L_E of the environment as well as all the agents' actions. Similarly, the evolution of the environment's local states is described by a function $t_E : L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_E$, returning the next environment state given the current round of actions. It is assumed that in every state agents evolve simultaneously. The evolution of the global states of the whole system is described

by a function $t : G \times Act \to G$, where $G \subseteq L_1 \times \ldots \times L_n \times L_E$ is the set of global states for the system reachable from a set of initial global states $I \subseteq G$, and $Act \subseteq Act_1 \times \ldots \times Act_n \times Act_E$ is the set of enabled joint actions. The function t is defined as $t(g, a) = g'$ if and only if for all $i, t_i(l_i(g), a) = l_i(g')$ and $t_E(l_E(g), a) = l_E(g')$, where $l_i(g)$ denotes the i-th component of global state g (corresponding to the local state of agent i). Finally, an interpreted system includes a set of atomic propositions AP together with a valuation function $V \subseteq AP \times G$. Formally, an *interpreted system* is defined as the tuple $IS = \langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (L_E, Act_E, P_E, t_E), I, V \rangle$.

Temporal-Epistemic Logic. Interpreted systems provide a natural semantics to epistemic logic, or logic of knowledge, which is routinely used to specify MAS [7] like web-services [16]. We consider the following syntax defining our specification language:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi, i \in \{1, \ldots, n\}$$

In the grammar above $p \in AP$ is an atomic proposition, and we have the usual negation and disjunction. Furthermore, $EX\varphi$ is read as *there is a global next state of computation in which* φ *holds*; $EG\varphi$ as *there exists a sequences of global states (or,* runs*) where* φ *holds in every state*, and $E(\varphi U \psi)$ as *there exists a run in which* φ *holds until* ψ *holds*; The knowledge operator $K_i\varphi$ is used to express that *agent* i *knows* φ. Based on this operators, we can define further useful operators to express often used properties. For instance, *there exists a run where* φ *is eventually true* ($EF\varphi$) can be written as $E(true\ U \varphi)$ and φ *holds on all runs in every state* ($AG\varphi$) is equivalent to *there is no run where* φ *does not hold* ($\neg EF \neg \varphi$).

Any interpreted system is associated to a model $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible worlds W is the set G of reachable global states. The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by considering the transition function t of the corresponding IS: two worlds w and w' are such that $R_t(w, w')$ if and only if there exists a joint action $a \in Act$ such that $t(w, a) = w'$. The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by considering the equality of the local components of the global states. Two worlds $w, w' \in W$ are such that $w \sim_i w'$ if and only if $l_i(w) = l_i(w')$ (i.e., two worlds w and w' are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same). The labelling relation $L \subseteq AP \times W$ is defined in terms of the valuation relation V.

We write $(M, w) \models \varphi$ to represent that a formula φ is true at a world w in a Kripke model M. Temporal formulae are interpreted in M_{IS} as standard. For the epistemic operator K we have: $(M, w) \models K_i\varphi$ iff for all $w' \in W$ $w \sim_i w'$ implies $(M, w') \models \varphi$. We refer the reader to [7] for more details on this and related epistemic concepts widely discussed in the epistemic logic literature.

MCMAS [15] is a BDD-based model checker for the automatic verification of multi-agent systems. It provides ISPL (Interpreted Systems Programming Language) as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulae as specifications of the system. The structure of an ISPL program allows the local states to be defined using *Boolean*, *bounded integer*, and *enumeration* variables. ISPL programs are closely related to interpreted systems; specifically, each ISPL program describes a unique interpreted system. MCMAS supports the verification for all formulae in the language above as well as others.

3 Mapping ABS Programs into ISPL

We define a map from a core subset of ABS to ISPL to verify the key innovative aspects of ABS. Given that any ISPL program uniquely defines an interpreted system, our map alternatively can be seen as the definition of a transition-based semantics for the subset of ABS we investigate.

To create a finite model for checking with MCMAS, we make a number of typical restrictions on the models to check. Specifically, we assume finite data types (Booleans, enumerations, bounded integers), introduce bounds on message and process queues (which also restricts recursion), and restrict object creation to an explicit *main* block. No *new* operator is used afterwards at run-time. Within these limitations, we support standard control flow and assignment statements, as well as the ABS-typical statements for local cooperative multitasking and message passing including data and object references. We refer to this bounded subset of the language as ABS_B. Extension by abstraction and introduction of parameterised techniques to handle infinite systems is left to future work.

The key correspondence we make is to associate objects in ABS_B to agents in ISPL and use the environment to handle message passing. Following this, we map ABS_B configurations to global states in interpreted systems and SOS rules to transitions, respectively. Our modelling of assignments and local conditionals follows the usual translation based on a current position in the control flow graph and corresponding updates to the variables. For more details see, e.g., the treatment of Boolean programs in [2]. We focus the discussion on message passing mechanisms and show that the semantic rules of Fig. 1 are preserved. The key element is the message buffer, which is encoded in the ISPL environment agent E. The buffer provides a bounded storage for n *invoc* messages, whose index also serves as value for future variables *fut*.

Definition 1 (Message Buffer). *A message buffer for n pending message calls in an ABS_B program is mapped to the ISPL environment E with local states $L_E \subseteq (B^1 \times \cdots \times B^n \times f \times agt \times tr)$. We write $b^i \in B^i$ for an entry in the buffer that holds a single call. It consists of a set of variables $b^i.\overline{v}$ and a status $b^i.stat$, which can be* `empty`*,* `o m` *to represent a call to method m in object o,* `wr` *while waiting for a return, or* `pr` *while holding a return message. The remaining fields are used for communication where f holds the index of the active entry, agt the index of the connected agent, and tr the index of the next parameter to transfer.*

As previously remarked, objects in ABS_B are modelled as agents in ISPL, where each agent contains a bounded number of execution slots for processes.

Definition 2 (Object Agent). *Each object o in ABS_B is mapped to a corresponding agent in ISPL with local states $L_o \subseteq S^1 \times \cdots \times S^k \times stat \times tr$. We write $s^i \in S^i$ for a process slot with a pointer $s^j.exec$ to the next statement to execute, a reference $s^j.caller$ to the calling method buffer and local variables $s^j.\overline{v}$. The state of the object is stored in stat and is either* `idle`*, initialising (*`init`*), or executing the processes in* $\texttt{slot}^\texttt{k}$*. The remaining variable tr holds the index of a slot that is communicating with the environment, or is 0 otherwise.*

We can give now the unique mapping between the ISPL model and the corresponding constructs from the ABS_B semantics. For a more succinct presentation, we use indices and exponents instead of the dot notation above when the meaning is clear from the context (e.g., we write $stat_E^i$ to refer to $l_E.b^i.stat$).

Definition 3 (Mapping Relation). *We define a mapping relation $\mu : cn \leftrightarrow L_E \times L_o$ between configurations and global states in the corresponding interpreted systems as*

$$\begin{aligned}
\mu(fresh(f)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{empty} \\
\mu(fut(f, \bot)) &\leftrightarrow \quad stat_E^{val(f)} \in \{\texttt{o_m}, \texttt{wr}\} \\
\mu(fut(f, true)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{pr} \\
\mu(invoc(o, f, m, v)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{o_m} \\
\mu(ob(o, a, p, q)) &\leftrightarrow \qquad stat_o = \texttt{slot}^{id(p)} \wedge exec_o^{id(p)} \neq 0 \wedge \forall_{j \in id(q)} exec_o^j \neq 0
\end{aligned}$$

where $val(f)$ is the value of the future variable f and $id(...)$ gives the (set of) indices of the process slots running respective processes. We denote the inverse as μ^{-1}.

Example 1. Fig. 2 gives an example of ABS code from a class *Node* of the AODV case study of Section 4 along with the relevant states of the control graph associated with the code. The special state ⓪ corresponds to an empty execution slot, states ① and ② represent that the execution of the object entered the corresponding method. The translation supports parameters, object references, global and local variables, maps and the usual assign and control flow statements.

3.1 Temporal Progress

Object agents and the environment progress simultaneously, synchronised by actions all participants need to agree on. We introduce the specific actions when needed and give the temporal progress of the system as transition relation for the object agent (t_o) and the environment (t_E) of the form

$$\begin{aligned}
t_o &= \{(l_o, l_E, a_E, a_o, l'_o) \mid \gamma\} \\
t_E &= \{(l_E, a_E, a_o, l'_E) \mid \gamma\}
\end{aligned}$$

```
1  Unit RREP(Node origin, Node src, Node dest, Int count){...}
   Unit RREQ(Node origin, Node src, Node dest, Int count){
3    if(src != this){
       rmap = store(rmap, origin, origin);
5      dmap = store(dmap, origin, 1);
       route = lookup(rmap, src);
7      if(route != null){
         distance = lookup(dmap, src);
9        if (count < distance){
           rmap = store(rmap, src, origin);
11         dmap = store(dmap, src, count);
       }}else{
13         rmap = store(rmap, src, origin);
           dmap = store(dmap, src, count);
15       }
       if(dest == this)
17       origin!RREP(this, src, dest, 1);
       else{
19       [...]
} } } } }
```

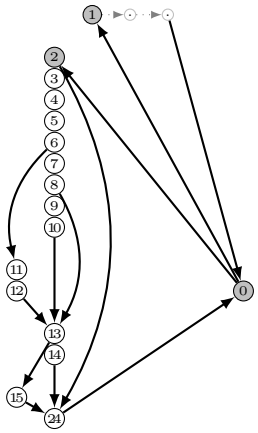

Fig. 2: AODV implementation in ABS along with the locations used in $exec_o^j$

where l_o and l_E are the current states of object agent and environment, a_E and a_o are the actions that are executed, and l'_o and l'_E are the new states for the object and the environment respectively. A transition only takes place if the environment and object agent agree on their actions. Formula γ is an update function that gives the values for the next state variables (primed) in terms of the variables of the current state (unprimed). In what follows, we give details on the update function γ representing message transfers involved in an ABS asynchronous method call.

Recall from Section 2 that protocols are used to determine which actions are enabled at a given state, whereas the local evolution function determines the successor state given some joint action. Consequently, the protocol of an object agent enables actions according to the *exec* pointer of the active execution slot, while the evolution function implements the variable updates for an executed action. For the message buffer, the protocol enables actions according to the status of its buffer entries, while the evolution function performs the actual data transfer and updates the variables.

To simplify the presentation, we give the temporal progress in the following as a single transition relation that contains the protocol function implicitly by defining that an action (and the corresponding transition) is only enabled if the unprimed variables match the current state. Because all communication is performed in one-to-one connections, we only describe the transitions concerned with this communication. However, the rules are instantiated for all agent combinations and agents not currently involved in a communication can perform local steps. Finally, we assume inertia for the variables, i.e, variables that are not explicitly changed by an assignment to their primed versions are held constant.

We give the transitions for sending a method call (SC) from the object to the environment. This is the first of four steps for a method call, which is followed by delivering the call (DC) from the environment to the callee, sending the return

to the environment (SR) and receiving the return by the caller (RR). We will denote the caller object with o, the callee with $\hat{o}$, and the environment agent with the buffers as usual with E.

Sending a Method Call. The transfer of a method call $x := \hat{o}.m(\overline{v})$ from object agent o is performed in three phases: 1) initiating the transfer `(init)`, 2) transferring the data `(trf)`, and 3) closing the transfer `(cls)`.

The environment can initiate a transfer if there is an empty buffer ($stat^i_E = \texttt{empty}$) and no transfer is already performed ($tr_E = 0$). If this is the case, the action $\texttt{st\_o}_E$ is enabled. In the `(init)` step, the environment stores the object agent it communicates with in agt_E and the id of the buffer in f_E; tr_E is used as index of the parameter to transfer. In the second phase `(trf)`, action $\texttt{val\_x}_o$ transfers a value x from the caller, which is stored by action $\texttt{rc}_E$ in the buffer variable $v^{i,k}_E = x$, where k is the id of the transferred variable. Action $\texttt{ct}_E$ closes the transfer, where $\texttt{sto}_o$ indicates that the object agent stores a future variable for later reference ($rret^i_E = T$). If the object agent sends $\texttt{ign}_o$, no return value is required and the buffer can be freed after submitting the call to the callee ($rret^i_E = F$).

$$
\begin{aligned}
t^{SC}_E = \{(l_E, \texttt{st\_o}_E, \texttt{ô\_m}_o, l'_E) \mid\ & stat^i_E = \texttt{empty} \wedge tr_E = 0 \wedge tr'_E = 1 && (\texttt{init})\\
& \wedge {stat^i_E}' = \texttt{ô\_m} \wedge {agt_E}' = o \wedge f'_E = i\} \cup \\
\{(l_E, \texttt{rc}_E, \texttt{val\_x}_o, l'_E) \mid\ & agt_E = o \wedge tr_E = k \wedge f_E = i \wedge v^{i,k}_E = x && (\texttt{trf})\\
& \wedge tr'_E = k + 1\} \cup \\
\{(l_E, \texttt{ct}_E, \texttt{sto}_o, l'_E) \mid\ & stat^i_E = \texttt{ô\_m} \wedge f_E = i \wedge f'_E = 0 && (\texttt{cls})\\
& \wedge tr'_E = 0 \wedge {agt_E}' = 0 \wedge rret^i_E = T\} \cup \\
\{(l_E, \texttt{ct}_E, \texttt{ign}_o, l'_E) \mid\ & stat^i_E = \texttt{ô\_m} \wedge f_E = i \wedge f'_E = 0 && (\texttt{cls})\\
& \wedge tr'_E = 0 \wedge {agt_E}' = 0 \wedge rret^i_E = F\}
\end{aligned}
$$

The object agent o monitors the environment to signal the start of the communication with it by action $\texttt{st\_o}_E$, the caller at the same time indicates the object and method to call with $\texttt{ô\_m}_o$. The clause $stat_o = \texttt{slot}^\texttt{j}$ ensures that the process with the call is indeed currently executed and variable tr_o is set to j in the next state to mark an ongoing transfer (note that `(init)` is not enabled if tr_o is not 0 in the first place). In the second phase `(trf)`, action $\texttt{val\_x}_o$ transfers a value x from the caller, which is acknowledged by the environment by action $\texttt{rc}_E$. The index of the variable to transfer is given by environment variable tr_E. When all variables are transferred the transfer is closed in phase `(cls)` with the actions $\texttt{ct}_E$ and $\texttt{sto}_o$. In this step, object agent o stores f_E in a future variable of its local state and sets the execution pointer to the next statement.

$$t_o^{SC} = \{(l_o, l_E, \texttt{st\_o}_E, \texttt{ô\_m}_o, l_o') \mid exec_o^j = (x := \hat{o}.m(\overline{v})) \wedge tr_o = 0 \quad (\texttt{init})$$
$$\wedge\, stat_o = \texttt{slot}^{\texttt{j}} \wedge tr_o' = j\} \cup$$
$$\{(l_o, l_E, \texttt{rc}_E, \texttt{val\_x}_o, l_o') \mid exec_o^j = (x := \hat{o}.m(\overline{v})) \wedge tr_o = j \wedge v^k = x \quad (\texttt{trf})$$
$$\wedge\, stat_o = \texttt{slot}^{\texttt{j}} \wedge agt_E = o \wedge tr_E = k\} \cup$$
$$\{(l_o, l_E, \texttt{ct}_E, \texttt{sto}_o, l_o') \mid tr_o = j \wedge stat_o = \texttt{slot}^{\texttt{j}} \quad (\texttt{cls})$$
$$\wedge\, x_o^{j\prime} = f_E \wedge exec_o^{j\prime} = next() \wedge tr_o{}' = 0\}$$

Method calls without assignment to a local variable (i.e. of the form $\hat{o}.m(\overline{v})$) do not store a reference to the future and therefore do not allow the caller to access any return value. In such a case, the buffer in the environment can be freed as soon as the callee is informed about the call. This is communicated to the environment by performing the action $\texttt{ign}_o$ in phase `(cls)` corresponds to second `(cls)` rule of the environment, which sets the `rret` field to false.

$$t_o^{SC} = \{(l_o, l_E, \texttt{st\_o}_E, \texttt{ô\_m}_o, l_o') \mid exec_o^j = (\hat{o}.m(v)) \wedge tr_o = 0 \quad (\texttt{init})$$
$$\wedge\, stat_o = \texttt{slot}^{\texttt{j}} \wedge tr_o' = j\} \cup$$
$$\{(l_o, l_E, \texttt{rc}_E, \texttt{val\_x}_o, l_o') \mid exec_o^j = (\hat{o}.m(v)) \wedge tr_o = j \wedge v^k = x \quad (\texttt{trf})$$
$$\wedge\, stat_o = \texttt{slot}^{\texttt{j}} \wedge agt_E = o \wedge tr_E = k\} \cup$$
$$\{(l_o, l_E, \texttt{ct}_E, \texttt{ign}_o, l_o') \mid tr_o = j \wedge stat_o = \texttt{slot}^{\texttt{j}} \quad (\texttt{cls})$$
$$\wedge\, exec_o^{j\prime} = next() \wedge tr_o{}' = 0\}$$

Note that variable agt_E ensures the strict one to one connection for a transfer.

Similar protocols were defined for *delivering* the call to the callee, and sending the return message back to the caller. Correct synchronisation between agent and environment are essential for these operations. In contrast, ABS scheduling statements like `await`, which suspends the current process if an required return message has not arrived yet, does not require an action from the environment and is therefore implemented as local transition.

Partial Orders. The operational semantics of ABS does not impose any constraints on the execution order of statements in different objects or on the delivery order of sent messages. A full examination of the state space therefore involves the execution of all possible orders of the co-enabled statements, which leads to a state space and execution path explosion that is hard to handle even with succinct symbolic representations like BDDs. To reduce the data structures to a manageable size we employ *partial order reduction* (POR), by identifying partial orders of statements under which the same properties are fulfilled. For epistemic logic, however, the partial orders of actions that change the state must not be reduced even if the sequence of visible events is equivalent [14]. This is because

the knowledge operators of epistemic logic consider the reachable state space to determine what an agent can know about the state of the other components. A reduction of the reachable states thus wrongly increases the computed knowledge of an agent.

To apply POR, we use the strong data encapsulation of ABS, which prevents information from being exchanged during method execution, and evaluate the epistemic properties over the full execution only at states where variables can be accessed, resp. when the agent is idle and can receive method calls to read a variable. This decision allows us to combine the statements of an execution block into a *macro step* [11] and remove the interleavings that stem from concurrent execution of methods from different agents. We furthermore treat the transitions for message passing given above as atomic blocks.

3.2 Preservation of ABS Semantics

The attention in the rest of this section is given to showing that the mapping preserves the semantics for the described bounded subset of the language ABS_B. An essential condition for that to happen is that the bounds on message buffers and execution slots are not overrun. To monitor this we introduce a flag *overrun* to the resulting ISPL file that is set to true when a buffer is requested but none is free. Checking for sufficient buffer sizes now corresponds to checking for $AG\neg overrun$, which holds if the bounds suffice.

Theorem 1 (Preservation of ABS semantics). *An ABS_B program P satisfies an epistemic specification ϕ iff $\mu(P) \models (\phi \wedge AG\neg overrun)$, where $\mu(P)$ is the interpreted system corresponding to the program P.*

Proof (sketch). Given sufficiently high bounds for the execution slots and buffers, executions in ISPL preserve the ABS semantics. This preservation can be shown by following the construction in [13], that is by establishing a Galois connection between the ABS operational semantics and ISPL transitions. We sketch the proof for the semantic rules from Fig. 1. First, we show that for a configuration c and an ISPL state u, $c \xrightarrow{(Asynch\text{-}Call)} \mu^{-1}(u)$ if and only if $\mu(c) \xrightarrow{t_o \circ t_E} u$.

From the premise and left hand side of *(Asynch-Call)* we get $c = ob(o, a, \{l \mid x = \hat{o}!m(\overline{e}); s\}, q)$ with $\overline{v} = [\![\overline{e}]\!]_{(a \circ l)}$ and $fresh(f)$. Using $p := x = \hat{o}!m(\overline{e})$ and Def. 3 gives $\mu(c) = (stat_o = \texttt{slot}^{id(p)} \wedge exec_o^{id(p)} = (x = \hat{o}.m(\overline{e})) \wedge next() = s)$, which corresponds to the states enabling t_o^{SC} and t_E^{SC}. Thus, we have that *(Asynch-Call)* is enabled if and only if t_o^{SC} and t_E^{SC} are enabled. The execution of all stages in t_o^{SC} and t_E^{SC} results in $u = (stat_E^{val(f)} = \texttt{ô\_m} \wedge v_E^{val(f)} = \overline{v} \wedge x_o^{val(f)} = val(f))$. The result of *(Asynch-Call)* and a subsequent store of the future as local variable gives $ob(o, a, \{x = f \mid s\}, q) \wedge invoc(\hat{o}, f, m, v) \wedge fut(f, \bot)$. Def. 3 shows that this result indeed is $\mu^{-1}(u)$, which concludes the proof.

The connection for *(Bind-Mtd)* and transitions $t_o^{DC} \circ t_E^{DC}$ is analogous where $p' = bind(o, f, m, \overline{v}, class(o)))$ is the instantiation of a new process and corresponds to $exec_o^{id(p')} = start(m) \wedge caller_o^{id(p')} = val(f) \wedge v_o^{id(p')} = \overline{v}$.

Limitations. Observe that the bounds on message buffer and execution slots may lead to delays in sending or receiving message calls. In particular, the *free* and *bind* predicates from the premises of the *(Asynch-Call)* and *(Bind-Mtd)* rules are not enabled when no buffer, or execution slot, respectively, is free. While this limits the examined behaviour if the bounds cannot be increased, the checked runs and any counterexample returned are still valid. This implies that checking properties for existential formulae is sound but not complete if the bounds are not sufficient.

4 Implementation and Experimental Results

We implemented the mapping from ABS to ISPL as a new back-end (ABSMC) to the ABS compiler framework, which already provides the parsing, type checking and generation of the abstract syntax tree. ABS programs have an explicit main block to set up the model. The ABSMC back-end executes this block and creates an agent with a unique name for each `new` statement, where arguments to the constructor are evaluated and passed as constants to the agent to reduce the state space.

The finite set of object names is used as enumerations to handle object references. Further variable types are Boolean, bounded integer, and enumerations. To allow generic variables in ISPL, the ABS variable types are mapped to integer of appropriate size. ABS allows annotations to statements and classes to provide additional information to the model. We use the annotations define the initial instances of an object and to provide the specifications we want to verify. The ISPL file generated by the ABSMC back-end is directly passed to the MCMAS model checker, which computes the set of reachable states and evaluates the formulae. MCMAS also can compute counterexamples and witnesses in a number of cases. The experiments were performed on an Intel i5 machine with 3.5GHz and 8GB RAM running Linux.

4.1 Verification of AODV Routing

To evaluate the applicability of the approach, we verified a distributed system implementing the *Ad-hoc On-Demand Distance Vector* (AODV) routing algorithm for mesh networks [18]. The purpose of the algorithm is to establish paths between two nodes from a set of arbitrarily distributed nodes, where not all nodes are within reach of each other. The algorithm is distributed in the sense that there is no central node deciding the routes of the packets. Instead, every node only knows the next hop to a given destination.

Initially no information about reachable nodes or routing is available to the nodes. The nodes communicate using RREQ (route request) or RREP (route reply) messages that carry the *origin* of the message, the *source* and the *destination* of the route to establish, and a *hop* count for the length of the route. The algorithm proceeds in two phases. First, an RREQ message is broadcast; this is relayed by its recipients until the destination is reached. The route is then

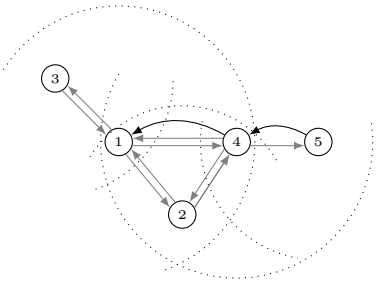

Fig. 3: AODV messages to establish a route between nodes 1 and 5.

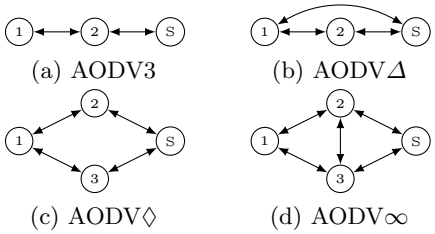

Fig. 4: Basic network topologies to verify properties of AODV.

established along the path of the RREQ messages using RREP messages. This is illustrated in Fig. 3, where RREQ broadcast messages are represented by straight grey lines, RREP by black curved lines, and dotted lines indicate the range of a node. Note that, since messages can be delayed, the first route that is established is not necessarily the shortest one. However, when several routes are available, the shortest is selected using the hop count.

We modelled the protocol in ABS using maps to store the next hop and the length of the route to a given node. Fig. 2 shows the ABS code executed at a node when receiving an RREQ message[3]. Upon receipt, if the source node is not the current node itself, the node first records that the origin of the message is a direct neighbour (distance 1), then it stores the route to the source via the origin unless there is already a shorter route known. If there is no route to the requested destination, the broadcast is relayed with increased hop count. Once a node with a route to the destination (or the actual destination itself) is reached, the second stage of the algorithm is performed by sending RREP messages towards the source of the route request. Otherwise, the message is broadcasted to its neighbours (omitted in the listing).

In our experiments we set up the different topologies from Fig. 4 and initiated the network by sending a request to node 1 (`N1`) to establish a route to node `S`. The compiler duly produced the resulting ISPL code that was fed to MCMAS as described earlier. We verified properties over the predicates `rXY` (representing that node `X` has a route to node `Y`), `rXZY` (node `X` has a route via node `Z` to node `Y`) and `finish` to denote that all messages have been delivered. We also added specifications to verify that, in all cases, routes are eventually established (represented as `AF r1S`) and that eventually the shortest route is found (`AF (finish` $\rightarrow$ `r1SS)` for Fig. 4b), even if intermediate routes might be longer (`EF r12S` in Fig. 4b).

Since MCMAS also supports an epistemic language we were further able to add specifications representing the information the agents have. In this context, we were able to check that when an agent sets a route, it knows that the next hop has a route to the destination (`AG (r12S -> K(N1, r2S))`). The topologies in

[3] The examples are available at `http://www.doc.ic.ac.uk/~agriesma/ABSMC`

Table 1: Experimental results of the AODV case study.

AODV3 | 250/24 | 20s | 50MB	
`AF r12S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T

AODVΔ | 250/24 | 32s | 54MB	
`AF r1SS`	<1 T
`AG (finish` $\rightarrow$ `r1SS)`	<1 T
`AF r12S`	10 F
`EF r12S`	2 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r1SS -> (! K(N1, r2S))`	3 F
`AG (lostRREP & r1SS -> (! K(N1, r2S))`	<1 T

AODV$\Diamond$ | 430/30 | 220s | 84MB	
`AF r12S`	64 F
`EF r12S`	19 T
`AF r1S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r13S -> (! K(N1, r2S))`	36 F
`AG (lostRREP & r13S -> (! K(N1, r2S))`	<1 T

AODV∞ | 430/30 | 2262s | 351MB	
`AF r12S`	854 F
`EF r12S`	505 T
`AF r1S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r13S -> (! K(N1, r2S))`	899 F

Fig. 4c and Fig. 4d have different shortest routes. Which one is selected depends on the order in which the RREP messages arrive at the source. If one route is selected, the initial node does not know about the node on the other route; that is `AG (r12S -> K(N1, r3S))` is false, as MCMAS confirmed.

In the two topologies above, we might expect `AG (r12S -> !K(N1, r3S))` to hold. Our verification results showed, however, that this intuition is incorrect. The counterexample produced by MCMAS demonstrates that even in cases where a route via `N2` is selected, `N1` knows that `N3` has a valid route when it receives its RREP message. In 4d there are cases where `N1` can deduce some information even when this RREP is not taken into account and when its route goes via `N2`, namely the case when the route has 3 steps, in which case `N2` must route via `N3` and thus `N3` must have a route to `S`. Note that `N1` can deduce this knowledge solely from its local state and observations, and that they hold only for some of the possible orders of the messages. Model checking and examination of the counterexamples greatly helps to find such corner cases. To check for stability of the protocol, we added message loss to model 4a by allowing the environment to drop messages (in which case the predicate `lostRREP` is set to true). This corresponds to a change of topology where the corresponding connection is temporarily lost. The results show that for topologies with multiple routes, a route can be established even if one message is lost. We do, however, lose the certainty of finding the shortest route.

The results are summarised in Table 1, where for each topology the number of bits required for states/actions, the time for computing the reachable states, and the used memory are given. For each formula the result and computation time is given. We see that besides the number of nodes, which influences the number of possible states in the model, the number of connections is the main source of complexity. This is because the extra method calls from the additional messages increase the order of method calls that have to be considered, and thus the total number of reachable states. Note that the sizes of the checked systems

are comparable to other work on AODV [20, 8, 17], while the used logic is more expressive and harder to check.

5 Conclusions

In this paper we presented a formal mapping between the modelling language for distributed systems ABS and interpreted systems, a formalism for multi-agent systems. We implemented the map into a compiler for the MCMAS model checker and evaluated the performance of the approach by studying a well-known networking protocol. While the approach requires us to provide bounds manually on the number of active processes, we were able to conclude the approach is mathematically sound and reasonably efficient. We are not aware of comparable literature on the subject as ABS models are currently only validated by simulation and test case generation [12]. Besides optimisations to the integration of the mapping with the model checker, our current direction of work in this line consists in providing automatic estimations on the required bounds for building the model as well as automatic abstraction techniques.

Acknowledgements

The authors would like to thank the members of the HATS (FP7-231620) project, and in particular Rudolf Schlatte and Jan Schäfer, for providing access and help to the ABS compiler framework.

The work reported in this paper was conducted at Imperial College London while Andreas Griesmayer was supported by the Marie Curie Fellowship "DiVerMAS" (FP7-PEOPLE-252184). Alessio Lomuscio acknowledges support from the UK Engineering and Physical Sciences Research Council through the Leadership Fellowship grant "Trusted Autonomous Systems" (EP/I00520X/1).

References

1. Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, 1977.
2. T. Ball and S. Rajamani. The SLAM toolkit. In *CAV'01*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.
3. S. Bensalem, M. Bozga, T.H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. In *ATVA'08*, volume 5311 of *LNCS*, pages 181–193. Springer, 2010.
4. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS'05*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All about Maude - a high-performance logical framework: how to specify, program and verify systems in rewriting logic*, volume 4350 of *LNCS*. Springer, 2007.

7. R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about knowledge*. The MIT Press, Cambridge, Massachusetts, 1995.
8. A. Fehnker, R.J. van Glabbeek, P. Höfner, A. McIver, M. Portmann, and W.L. Tan. Automated analysis of AODV using UPPAAL. In *TACAS'12*, volume 7214 of *LNCS*, pages 173–187. Springer, 2012.
9. G. J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2003.
10. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
11. R.K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *PPPC'09*, pages 11–20, New York, New York, USA, 2009. ACM Press.
12. W. Leister, J. Bjørk, R. Schlatte, and A. Griesmayer. Verifying distributed algorithms with executable Creol models. In *PESARO'11*, pages 1–6. IARIA, 2011.
13. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, and D. Probst. Property preserving abstractions for the verification of concurrent systems. *Formal methods in system design*, 6(1):11–44, 1995.
14. A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1):71–90, 2010.
15. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *CAV'09*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.
16. A. Lomuscio, H. Qu, and M. Solanki. Towards verifying contract regulated service composition. *Journal of Autonomous Agents and Multi-Agent Systems*, 24(3):345–373, 2010.
17. M. Musuvathi, D. Y. W. Park, A. Chou, D. R.Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. *SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.
18. C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.
19. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):416–430, 2000.
20. F. De Renesse and A.H. Aghvami. Formal verification of ad-hoc routing protocols using SPIN model checker. In *MELECON'04*, volume 3, pages 1177–1182. IEEE, 2004.
21. V. Stolz and F. Huch. Runtime verification of concurrent haskell programs. In *RV'04*, volume 113 of *ENTCS*, pages 201–216. Elsevier Science Publishers, 2005.
22. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.
23. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *ASE'07*, pages 501–504. ACM, 2007.