Automated verification of resource requirements in multi-agent systems using abstraction*

Natasha Alechina, Brian Logan, Hoang Nga Nguyen, and Abdur Rakib

University of Nottingham, UK
nza,bsl,hnn,rza@cs.nott.ac.uk

Abstract. We describe a framework for the automated verification of multi-agent systems which do distributed problem solving, e.g., query answering. Each reasoner uses facts, messages and Horn clause rules to derive new information. We show how to verify correctness of distributed problem solving under resource constraints, such as the time required to answer queries and the number of messages exchanged by the agents. The framework allows the use of abstract specifications consisting of Linear Time Temporal Logic (LTL) formulas to specify some of the agents in the system. We illustrate the use of the framework on a simple example.

1 Introduction

Much current work in multi-agent systems (MAS) development relies on the developer specifying agent behaviour in terms of pre-defined plans [1]. While the use of pre-defined plans makes it easier to guarantee the behaviour of the multi-agent system, e.g., [2], it can make it harder for the system to solve novel problems not anticipated by the system designers. As a result, there has recently been increasing interest in providing general reasoning capabilities to agents in multi-agent systems (see, for example, [3, 4]). However, while the incorporation of reasoning abilities into agents brings great benefits in terms of flexibility and ease of development, these approaches also raise new challenges for the agent developer, namely, how to ensure correctness (will an agent produce the correct output for all legal inputs), termination (will an agent produce an output at all), and response time (how much computation will an agent have to do before it generates an output). For example, when developing a distributed problem solving system which provides subway routes to users of the London Underground, a developer may wish to verify that the system does not provide invalid routes (e.g., that it takes current service disruptions into account), and that it provides bounded response times under expected system loads (e.g., asynchronous queries from multiple simultaneous users). Proving correctness or resource bounds for such large complex reasoning systems is infeasible with current verification technologies.

In [5], an approach to verifying resource requirements in systems of communicating rule-based reasoners was proposed. The main emphasis of that paper was on modelling

* This work was supported by the UK Engineering and Physical Sciences Research Council [grant EP/E031226/1].

systems of communicating reasoners as state transition systems, where states correspond to beliefs of the agents and transitions correspond to the application of a rule of inference or sending a message. Properties of the system, for example, that a system of two agents will be able to produce an answer to a query after exchanging at most one message and applying 4 rules, were specified in modal logic, and proof-of-concept verification experiments using Mocha model-checker [6] reported. However, the encoding of the system in the Mocha specification language had to be handcrafted, rules had to be propositionalised using all possible substitutions for variables, and scalability of the verification approach was not explored.

In this paper we describe an automated verification framework for resource-bounded reasoners, which takes rules specified in Hornlog RuleML with negation as failure [7] augmented with communication primitives, and automatically produces a Maude [8] specification of the system which can be efficiently verified. The properties that we wish to verify are response-time guarantees of the form: if the system receives a query, then a response will be produced within n timesteps. To allow larger systems to be verified, *abstract specifications* can be used to model some agents in the system. Abstract specifications are given as LTL formulas which describe the external behaviour of agents, and allow their temporal behaviour (the response time behaviour of the agent), to be compactly modelled. We illustrate the scalability of our approach by comparing it to results presented in [9] for a synthetic distributed reasoning problem, and presenting results for a more complex multi-agent reasoning example.

The remainder of the paper is organised as follows. In section 2 we describe our model of communicating rule-based reasoners. In section 3 we describe the basic components and ideas behind the verification framework, including our approach to producing abstractions of agents. In section 4 we briefly describe a tool for translating rule-based specification of the agents into Maude, and in section 5 we evaluate its performance. We discuss related work and open problems in section 6 and conclude in section 7.

2 Communicating Reasoners

We adopt a general model of distributed reasoners. A distributed reasoning system consists of $n\,(\geq 1)$ individual reasoners or *agents*. Each agent is identified by a value in $\{1, 2, \ldots, n\}$ and we use variables i and j over $\{1, 2, \ldots, n\}$ to refer to agents. Each agent i has a program, consisting of first-order Horn clause rules with negation-as-failure allowed in the premises[1], and a working memory, which contains facts (ground atomic formulas) representing the initial state of the system. The agents execute synchronously. At each cycle, each agent matches (unifies) the conditions of its rules against the contents of its working memory. The conditions of a rule are evaluated using the closed world assumption (i.e., $not\ P$ evaluates to true if P is not in working memory). A match for every condition of a rule constitutes an instance of that rule (a rule may have more than one instance). The set of all rule instances for an agent form the agent's *conflict set*. Each agent then chooses a subset of rule instances from the conflict

[1] Rules are of the form $P_1 \wedge \ldots \wedge P_n \rightarrow P$ where P is an atomic formula and P_i are atomic formulas or atomic formulas preceded by the negation as failure operator.

set to be applied. Applying a rule adds the consequent of the rule as a new fact to the agent's working memory. The cycle begins again with the match phase and the process continues until no more rules can be matched and all agents have an empty conflict set.[2]

We assume that each reasoner has a *reasoning strategy* (or conflict resolution strategy) which determines the order in which rules are applied when more than one rule matches the contents of the agent's working memory. The choice of reasoning strategy is important in determining the capabilities of the agent. For example, different reasoning strategies may determine how quickly/efficiently an answer to a query can be derived, or even whether an answer can be produced at all. The reasoning strategy is also important in determining trade-offs between the resources required to process a query. For example, if multiple queries arrive at about the same time, processing them sequentially may reduce the memory required at the cost of increasing the worst case response time for queries. Conversely, processing the queries in parallel may reduce the worst case response time at the cost of increasing the peak memory usage.

We assume that each reasoner executes in a separate process and that reasoners communicate via message passing. For concreteness, we assume a simple query-response scheme based on asynchronous message passing. Each agent's rules may contain two distinguished communication primitives: $ASK(i, j, P)$, and $TELL(i, j, P)$, where i and j are agents and P is an atomic formula not containing an ASK or a $TELL$. $ASK(i, j, P)$ means 'i asks j whether P is the case' and $TELL(i, j, P)$ means 'i tells j that P ($i \neq j$). The positions in which the ASK and $TELL$ primitives may appear in a rule depends on which agent's program the rule belongs to. Agent i may have an ASK or a $TELL$ with arguments (i, j, P) in the consequent of a rule, e.g.,

$$P_1 \wedge P_2 \wedge \ldots \wedge P_n \Longrightarrow ASK(i, j, P)$$

Agent j may have the same expressions in the antecedent of the rule, e.g.,

$$TELL(i, j, P) \Longrightarrow P$$

is a well-formed rule for agent j, that causes it to believe i when i informs it that P is the case. No other occurrences of ASK or $TELL$ are allowed. For simplicity, we assume that communication is error-free and takes one tick of time.

3 Verification Framework

We would like to be able to verify properties of systems consisting of arbitrary numbers of complex communicating reasoners. However verifying such large, complex reasoning systems is infeasible with current verification technologies.

The most straightforward approach to defining the global state of a multi-agent system is as a (parallel) composition of the local states of the agents. At each step in the evolution of the system, each agent chooses from a set of possible actions (we assume that an agent can always perform an 'idle' action which does not change its state). The

[2] Note that, although execution is synchronous, agents can return a 'null action' at any given cycle, allowing the modelling of multi-agent systems in which each agent deliberates at a rate which is a multiple of the cycle time of the fastest agent.

actions selected by the agents are then performed in parallel and the system advances to the next state. In a multi-agent system composed of n (≥ 1) agents, if each agent i can choose between performing at most m (≥ 1) actions, then the system as a whole can move in m^n different ways from a given state at a given point in time. Along with state space size, model checking performance is heavily dependent on the branching factor of states in the reachable state space and the solution depth of a given problem. In general, the model checking algorithm for reachability analysis performs a breadth-first exploration of the state transition graph. When checking invariant (safety) properties, the model-checker will either determine that no states violate the invariant by exploring the entire state space, or will find a state violating the invariant and produce a counter-example.[3] However, even with state-of-the-art BDD-based model-checkers, memory exhaustion can occur when computing the reachable state space due to the large size of the intermediate BDDs (because of the high branching factor).

To overcome this problem, our modelling approach abstracts from some aspects of system behaviour to obtain a system model that is tractable for a standard model-checker. Our use of abstraction is however different from classic approaches in model-checking, such as [11, 12]. We assume that, at any given point in the design of the overall system, the detailed behaviour of only a small number of agents (perhaps only a single agent) is of interest to the system designer, and the remaining agents in the system can be considered at a high level of abstraction. When verifying response time guarantees of the 'focal' agent(s), the concrete representation of 'peripheral' agents can be replaced by an abstract specification of their external (communication) behaviour, so long as the abstract specification results in behaviour that is indistinguishable from the original concrete representation for the purposes of verification, i.e., it produces queries and responds to queries within specified bounds. All other details of an abstract agent's internal behaviour are omitted.

The decision regarding which agents to abstract and how their external behaviour should be specified rests with the modeller/system designer. Specifications of the external (observable) behaviour of abstract agents may be derived from, e.g., assumed characteristics of as-yet-unimplemented parts of the system, assumptions regarding the behaviour of parts of the overall system the designer does not control (e.g., quality of service guarantees offered by an existing web service) or from the prior verification of the behaviour of other (concrete) agents in the system. The behaviour of abstract agents is specified using the language of the temporal logic LTL containing epistemic operators. The general form of the formulas which can be used to represent the external behaviour of abstract agents is given below, where X is the next step temporal operator, X^n is a sequence of n X operators, G is the temporal 'in all future states' operator, and B_i for each agent i is a syntactic epistemic operator used to specify agent i's 'beliefs' or the contents of its working memory.

$$\rho :: X^n \phi_1 \mid G(\phi_2 \rightarrow \phi_3)$$
$$\phi_1 :: B_i \; ASK(i, j, P)$$

[3] Even with on-the-fly model-checking [10], the model checker has to explore the state space at least until the solution depth.

$$
\begin{aligned}
&|B_i\ TELL(i,j,P)\\
&|B_i\ ASK(j,i,P)\\
&|B_i\ TELL(j,i,P)\\
&|B_i\ P\\
\phi_2 &:: Bj\ ASK(i,j,P)\\
\phi_3 &:: X^n\ B_i\ TELL(j,i,P)
\end{aligned}
$$

Formulas of the form $X^n\phi_1$ describe agents which produce a certain message or input to the system within n time steps. The $G(\phi_2 \rightarrow \phi_3)$ formulas describe agents which are always guaranteed to reply to a request for information within n timesteps. Note that we do not need the full language of LTL (for example, the Until operator) in order to specify abstract agents. The verification language of Maude contains full LTL, but abstract specifications and the response-time guarantee properties we wish to verify can be expressed in the fragment above.

Formulas expressing abstract specifications are translated into the specification language of the model checker. This is a kind of backward modeling, which basically imposes a restrictions on possible runs of a model. The multi-agent system is then simply a parallel composition of both the concrete and abstract agents in the system.

4 Automated Verification Tool

In this section, we describe a tool based on the Maude [8] rewriting system which implements the approach to verification described above. The tool generates an encoding of a distributed system of reasoning agents for the Maude LTL model checker, which is then used to verify the desired properties of the system. We chose the Maude LTL model checker because it can model check systems whose states involve arbitrary algebraic data types. The only assumption is that the set of states reachable from a given initial state is finite. This simplifies modelling of the agents' (first-order) rules and reasoning strategies. For example, a rule used by a route planning agent such as

$$
\begin{aligned}
&Connected(station1, station2, line1) \wedge Reachable(station2, station3, [route])\\
&\quad \rightarrow Reachable(station1, station3, [station2|route])
\end{aligned}
$$

where $station1$, $station2$, $station3$, $line1$ and $route$ are variables, can be represented directly in the Maude encoding, without having to generate all ground instances resulting from possible variable substitutions.

The tool consists of three main components: the user interface, the encoding generator and the system verifier. The tool takes as input: (a) a set of concrete agent descriptions, each comprising a set of rules, a set of initial working memory facts, and a control strategy, (b) a set of abstract agent descriptions specified by a set of temporal epistemic logic formulas, and (c) the properties of the system to be verified specified in temporal epistemic logic. Rules and facts can be expressed in RuleML or in a simplified ASCII syntax e.g., $< n{:}P_1\ \&\ \ldots\ \&\ P_n => P >, P_k$. The general XML syntax of rules

accepted by the framework corresponds to Hornlog RuleML with negation as failure, and is shown below.

```
<!- -Representation of rules - ->
<Implies>
 <head>
  <Atom>
   <Rel>Predicate< /Rel>
   <Var>variable< /Var>
          ⋮
   <Ind>constant< /Ind>
          ⋮
  < /Atom>
 < /head>
 <body>
  <And>
   <Atom>
    <Rel>Predicate< /Rel>
    <Var>variable< /Var>
          ⋮
    <Ind>constant< /Ind>
          ⋮
   < /Atom>
    ⋮
   <Naf>
    <Atom>
     <Rel>Predicate< /Rel>
     <Var>variable< /Var>
          ⋮
     <Ind>constant< /Ind>
          ⋮
    < /Atom>
   < /Naf>
  < /And>
 < /body>
< /Implies>
⋮
<!- -Representation of facts - ->
<Atom>
 <Rel>Predicate< /Rel>
 <Ind>constant< /Ind>
```

$\vdots$
$< /\text{Atom}>$
$\vdots$

Rules are translated internally into the simplified ASCII syntax. Once translated, they can be annotated by the user with rule priorities, and these annotated rules are then used to produce Maude specification. Rule priorities are required by some of the supported inference (conflict resolution) strategies. The tool supports a wide range of inference strategies including those provided by the CLIPS expert system shell [13], the Jess rule engine [14], and others [15]. Different agents in the system may use different strategies. The LTL specification of the behaviour of abstract agents and properties to be verified are given in a simplified ASCII notation.

4.1 Maude Implementation

The overall structure of the implementation is shown in Figure 1. Each agent has a configuration (local state) and the composition of all these (local state) configurations make the (global state) configuration of the multi-agent system.

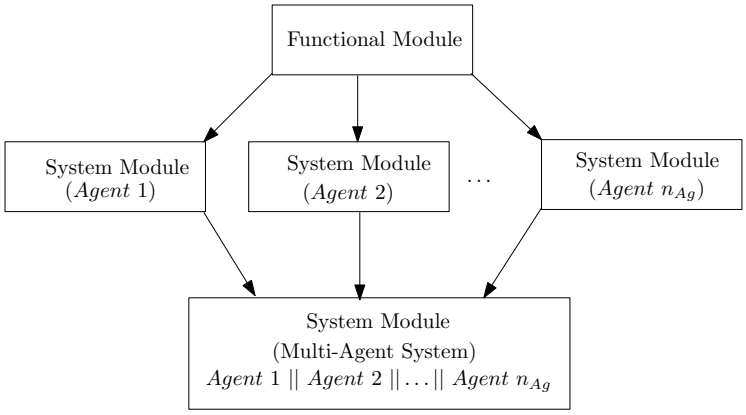

Fig. 1. Structure of the Maude implementation

The types necessary to implement the local state of an agent (working memory, program, control strategy, message counters, timestep etc.) are declared in a generic Maude functional module. The local configuration of each agent is represented as a tuple $Si[a : Agenda, a' : Agenda, tw : TimeWM, w : WM, t : Nat, t' : Nat, mc : Nat, b : Bool]iS$, where t represents the system cycle time, mc is the message counter, and b is a Boolean flag which is used for synchronisation. The rules of each agent are defined using an operator which takes as arguments a set of patterns (of sort $TimeWM$) specifying the antecedents of the rule and a single patten (of sort $TimeP$) specifying the consequent, and returns an element of sort $Rule$. In the case of concrete agents,

each of the agent's Horn clause rules is represented by an element of sort $Rule$. As an example, a rule (expressed in the ASCII syntax) $\langle 1{:}Father(x,y) => Male(x)\rangle$ is represented as follows

$ceq\ rule\text{-}ins(A, [t1 : Father(x,y)]\ TM, M) = Rl\langle 1 : [t1 : Father(x,y)] - >> [0 : Male(x)]\rangle lR\ rule\text{-}ins(Rl\langle 1 : [t1 : Father(x,y)] - >> [0 : Male(x)]\rangle lR\ A, [t1 : Father(x,y)]\ TM, M)\ if\ (not\ inAgenda(Rl\langle 1 : [t1 : Father(x,y)] - >> [0 : Male(x)]\rangle lR, A)) \wedge (not\ inWorkingMemory(Male(x), M))$.

Note that the rule is translated using the corresponding time pattern for efficiency purposes. In the rule the number 1 represents rule salience and the placeholder $t1$ represents time stamp of the corresponding pattern. Each equation may give rise to more than one rule instance depending on the elements in working memory. To prevent the regeneration of the same rule instance, the conditional equation checks whether the rule instance and its consequent are already present in the agenda and working memory. A sort $Agenda$ is declared as a supersort of $Rule$. These data types are manipulated by a set of equations, e.g., to check whether or not a given pattern (used to represent fact) is already in the agent's working memory, whether or not a rule instance is already in the agenda etc. Additional equations are used to implement control strategies, e.g., to determine the highest priority rule instance in the agenda, or the pattern with highest time stamp in working memory etc.

We model each (concrete and abstract) agent using a Maude system module which imports the generic functional module. System modules contain both functions and rewrite rules which are used to implement the dynamic behaviour of the system. For concrete agents, the agent's inference cycle is implemented using three Maude rules:

$rl[match] : [A \mid RL \mid TM \mid M \mid t \mid msg \mid 1 \mid true] => [rule\text{-}ins(A, TM, M)A \mid RL \mid TM \mid M \mid t \mid msg \mid 2 \mid false]$.

$rl[select] : [A \mid RL \mid TM \mid M \mid t \mid msg \mid 2 \mid true] => [del(strategy(A, A), A) \mid strategy(A, A)\ RL \mid TM \mid M \mid t \mid msg \mid 3 \mid false]$.

$crl[execute] : [A \mid Rl\langle n : Ant - >> Cons\rangle lR\ RL \mid Ant\ TM \mid M \mid t \mid msg \mid 3 \mid true] => [A \mid RL \mid Ant\ time(Cons, t+1)TM \mid pattern(Cons)\ M \mid t+1 \mid msg \mid 1 \mid false]\ if\ (not\ inWorkingMemory(pattern(Cons), M))$.

The *match* phase is implemented by the *match* rule, which generates a set of rule instances based on the elements of $TimeWM$. The *conflict resolution* phase is implemented using the *select* rule, which selects a subset of rule instances from the agenda for execution based on the agent's control strategy. Finally, the *execute* phase is implemented using the *execute* rule, which executes the rule instances selected for execution. These three Maude rules are controlled using a flag which ensures that only one rule is applied at each system cycle. When the *match* and *select* rules execute, the time counter in the agent's configuration remain unchanged. However, the time counter is increased by one when the *execute* executes. All three phases, *match*, *select* and *execute*, therefore happen in one timestep.

The external behaviour of abstract agents are represented by means of temporal epistemic formulas. These formulas are translated into Maude agent specifications. For example, the formula $G(B_j ASK(i, j, P) \rightarrow X^n B_i TELL(j, i, P)))$ which states that

if the abstract agent j believes that (concrete or abstract) agent i asks whether P is the case, then j should respond to agent i within n time steps, is translated as

$op\ halt\text{-}condition : TimeWM\ Nat\ WM -> Bool.$

$eq\ halt\text{-}condition([t' : ASK(i,j,p)]TM, t, M) = if\ ((t == t' + m)\ and\ (not\ inWorkingMemory(p, M)))\ then\ true\ else\ halt\text{-}condition(TM, t, M)\ fi.$

$eq\ halt\text{-}condition(TM, t, M) = false\ [owise].$

$crl\ [reply] : Sj[A \mid RL \mid [t' : ASK(1,2,P)]\ TM \mid M \mid t \mid msg \mid 3 \mid true]jS => Sj[A \mid RL \mid [t' : ASK(1,2,P)][t+1 : P]\ TM \mid P\ M \mid t+1 \mid msg \mid 1 \mid false]jS \quad if\ (not\ inWorkingMemory(P, M)) \wedge t < t' + m.$

$crl\ [idle] : S2[A \mid RL \mid TM \mid M \mid t \mid msg \mid 3 \mid true]2S => S2[A \mid RL \mid TM \mid M \mid t+1 \mid msg \mid 1 \mid\ false]2S\ if\ (not\ halt\text{-}condition(TM, t, M)).$

where t is the current cycle time, t' is the time stamp when agent j came to believe that agent i asked for P and m is the bound defined above. The two Maude rules *reply* and *idle* execute non-deterministically when $t < t' + m$, but the *idle* rule cannot be applied when $t = t' + m$, forcing the agent to reply at $t' + m$ if it has not already done so.

Once all the agents of the system have been defined using system modules, we import them all into a single MAS system module. The MAS module defines two Maude rules, *parallel-comp*, which implements the parallel composition of agent configurations in the system, and *sync-rule*, which is used to synchronise the time cycle of the global system.

$op\ \_||\_ : Config\ Config -> Config\ [comm\ assoc].$

$crl\ [parallel\text{-}comp] : C1{:}Config \,||\, C2{:}Config => C1'{:}Config \,||\, C2'{:}Config\ if\ C1{:}Config => C1 : Config \wedge C2 : Config => C2' : Config \wedge C1' : Config \neq C1{:}Config \wedge C2'{:}Config \neq C2{:}Config.$

$rl\ [sync\text{-}rule] : S1[A1 \mid RL1 \mid TM1 \mid M1 \mid t1 \mid msg1 \mid rc1 \mid false]1S \,||\, \ldots \,||\, Sn[An \mid RLn \mid TMn \mid Mn \mid tn \mid msgn \mid rcn \mid false]nS => S1[A1 \mid RL1 \mid TM1 \mid M1 \mid t1 \mid msg1 \mid rc1 \mid true]1S \,||\, \ldots \,||\, Sn[An \mid RLn \mid TMn \mid Mn \mid tn \mid msgn \mid rcn \mid true]nS.$

Communication between agents is also implemented using rules in the MAS module. For example, if agent i fires a communication rule of the form $< n{:}P_1 \& \ldots \& P_n => ASK(i,j,P) >$ which adds a fact $ASK(i,j,P)$ to its working memory, this fact is communicated to agent j using the following Maude rule

$$
\begin{array}{l}
crl\ [comm]\ : \\
S1[A1 \mid RL1 \mid TM1 \mid M1 \mid t1 \mid msg1 \mid 3 \mid true]1S \\
\vdots \\
\,||\, Si[Ai \mid RLi \mid TMi \mid ASK(i,j,P)\, Mi \mid ti \mid msgi \mid 3 \mid true]iS \\
\vdots \\
\,||\, Sj[Aj \mid RLj \mid TMj \mid Mj \mid tj \mid msgj \mid 3 \mid true]jS \\
\vdots \\
\,||\, Sn[An \mid RLn \mid TMn \mid Mn \mid tn \mid msgn \mid 3 \mid true]nS \\
=>
\end{array}
$$

$C1' : \mathit{Config}$

$\vdots$

$|| \, Si[Ai \,|\, RLi \,|\, TMi \,|\, ASK(i,j,P)\, Mi \,|\, ti+1 \,|\, msgi+1 \,|\, 1 \,|\, false]iS$

$\vdots$

$|| \, Sj[Aj \,|\, RLj \,|\, [tj{+}1 : ASK(i,j,P)]TMj \,|\, ASK(i,j,P)\, Mj \,|\, tj{+}1 \,|\, msgj{+}1 \,|\, 1 \,|\, false]jS$

$\vdots$

$|| \, Cn'{:}\mathit{Config}$
$\mathit{if}\ (\mathit{not}\ \mathit{inWorkingMemory}(ASK(i,j,P), Mj))$
$\wedge\ S1[A1 \,|\, RL1 \,|\, TM1 \,|\, M1 \,|\, t1 \,|\, msg1 \,|\, 3 \,|\, true]1S \Rightarrow C1'{:}\mathit{Config}$

$\vdots$

$\bigwedge\ Sn[An \,|\, RLn \,|\, TMn \,|\, Mn \,|\, tn \,|\, msgn \,|\, 3 \,|\, true]nS \Rightarrow Cn'{:}\mathit{Config}$
$\bigwedge\ C1'{:}\mathit{Config} \neq S1[A1 \,|\, RL1 \,|\, TM1 \,|\, M1 \,|\, t1 \,|\, msg1 \,|\, 3 \,|\, true]1S$

$\vdots$

$\bigwedge\ Cn'{:}\mathit{Config} \neq Sn[An \,|\, RLn \,|\, TMn \,|\, Mn \,|\, tn \,|\, msgn \,|\, 3 \,|\, true]nS.$

When $ASK(i,j,P)$ is added to agent j's working memory, j may perform some computation if it does not know whether P is the case. In this model, communication requires a single timestep, i.e., when agent i asks agent j whether P is the case at time step t, agent j will receive the request at time cycle $t+1$. However the time agent i has to wait for a response to its query depends on the reasoning j must (or chooses) to do (if j is concrete), or j's specification (if j is abstract). A similar approach is used when j tells i that P.

5 Experimental Evaluation

In this section we report experiments designed to illustrate the scalability and expressiveness of our approach. All the experiments reported here were performed on an Intel Pentium 4 CPU 3.20GHz with 2GB of RAM under CentOS release 4.8.

5.1 Scalability

To illustrate the scalability of our approach we implemented an example scenario reported in [9]. In this scenario, a system of communicating reasoners attempt to solve a (synthetic) distributed reasoning problem in which the set of rules and facts that describes agents' knowledge base are constructed from a complete binary tree. For example, a complete binary tree with 8 leaf facts has the following set of rules

$$\textbf{RuleB1}\ A_1(x) \wedge A_2(x) \rightarrow B_1(x) \quad \textbf{RuleB2}\ A_3(x) \wedge A_4(x) \rightarrow B_2(x)$$
$$\textbf{RuleB3}\ A_5(x) \wedge A_6(x) \rightarrow B_3(x) \quad \textbf{RuleB4}\ A_7(x) \wedge A_8(x) \rightarrow B_4(x)$$
$$\textbf{RuleC1}\ B_1(x) \wedge B_2(x) \rightarrow C_1(x) \quad \textbf{RuleC2}\ B_3(x) \wedge B_4(x) \rightarrow C_2(x)$$
$$\textbf{RuleD1}\ C_1(x) \wedge C_2(x) \rightarrow D_1(x)$$

For compatibility with the propositional example considered in [9], we assume that the variable x is substituted by a single constant value 'a', and the goal is to derive $D_1(a)$. One can easily see that a larger system can be generated using 16 'leaf' facts $A_1(x), \ldots, A_{16}(x)$, adding extra rules to derive $B_5(x)$ from $A_9(x)$ and $A_{10}(x)$, etc., and a new goal $E_1(x)$ derivable from $D_1(x)$ and $D_2(x)$ to give a '16 leaf example'. Similarly, we can consider systems with 32, 64, 128,. . .,2048 etc. leaf facts. Such generic distributed reasoning problems can be easily parameterised by the number of leaf facts and the distribution of facts and rules among the agents.

In [9], the results of experiments on such problems using the Mocha model-checker [6] are reported. In the simplest case of a single agent, the largest problem that could be verified using Mocha had 128 leaf facts. However, using our tool we are able to verify a system with 2048 leaf facts. The experimental results are summarised in Table 1.

		CPU Time	
# leaves	# steps	Mocha	Maude
128	127	1:47:52	0:0:1
512	511	—	0:1:37
1024	1023	—	0:15:03
2048	2047	—	3:40:52

Table 1. Resource requirements for a single agent

In case of a multi-agent systems, the exchange of information between agents was modelled as an abstract Copy operation in [9]. Each copy operation takes one tick of system time and does not require any special communication rules. As an example, to verify a multi-agent system consisting of two agents with 16 leaf facts, the Mocha encoding requires 1 hour and 36 minutes of CPU time. In our framework, communication between agents is achieved using ASK and $TELL$ actions. The results presented in [9] and those for our tool are therefore not directly comparable in the multi-agent case. Nevertheless, we can show that much larger multi-agent systems can be modelled using our approach.

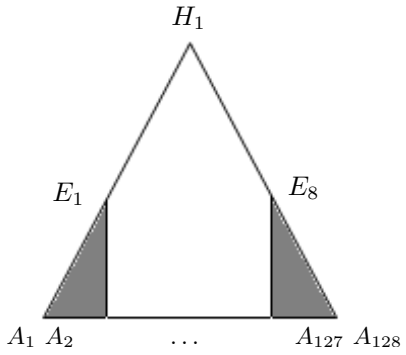

Fig. 2. Binary tree

Consider a multi-agent system consisting of two agents each with a knowledge base of facts and rules for the 128 leaf example (i.e., both agents have all the rules and leaf facts). Agent1 uses a reasoning strategy which assigns lower priority to rules in the right-hand shaded triangular region depicted in Fig. 2. In contrast, agent2 uses a reasoning strategy which assigns lower priority to rules in the left-hand shaded triangular region of Figure 2. Suppose agent1 asks agent2 if $E_8(a)$ is the case. If agent1 receives the fact $E_8(a)$ from agent2 before deriving $E_8(a)$ itself, it can avoid firing 15 rules, and the agents are able to derive the goal $H_1(a)$ in 115 steps while exchanging two messages.

Similarly, consider the scenario in which there are three agents, each with a knowledge base of facts and rules for the 128 leaf example. Assume agent1 asks agent2 if $E_1(a)$ is the case and also that agent1 asks agent3 if $E_8(a)$ is the case. Suppose the agents utilise reasoning strategies similar to the previous case where the set of rules in the unshaded region have higher priority for agent1, the rules in left hand shaded region have higher priority for agent2, and the rules in the right hand shaded region have higher priority for agent3. Then the agents can derive the goal $H_1(a)$ in 103 steps while exchanging four messages. The experimental results are summarised in Table 2. Although these examples are very simple, they point to the possibility of complex trade-offs between time and communication bounds in systems of reasoning agents.

# agents	# leaves	# steps	#msgs	CPU Time
2	128	115	2	0:0:7
3	128	103	4	0:0:18

Table 2. Resource requirements for multiple agents

5.2 A More Complex Example

To illustrate the application of the framework on a more complex example we consider the following scenario. The system consists of several agents representing users who have queries about possible subway routes on the London Underground denoted by u_i, and two agents that provide travel advice: a 'route planning' agent, p, which computes routes between stations and an 'engineering work' agent, e, which has information about line closures and other service disruptions. The user agents ask for route information to the route planning agent, that is, they generate queries of the form:

$$ASK(u_i, p, Route(start\_station, destination\_station)).$$

The route planning agent has a set of facts corresponding to connections between stations, and a set of rules for finding a path between stations which returns a route (a list of intermediate stations). Upon receiveing a request from the user agent, the route planning agent tries to find a route from the $start\_station$ to the $destination\_station$ by firing a sequence of rules based on the facts in its working memory. To ensure a route is valid, the planner must check that it is not affected by service disruptions caused by engineering work, which it does by querying the engineering work agent. If the route is

open, the planner returns the route from $source\_station$ to the $destination\_station$ to the user agent.

The user agents are modelled as abstract agents, which generate a query at a nondeterministically chosen timestep within a specified interval, e.g.:

$$X^5 B_{u_i} ASK(u_i, p, Route(MarbleArch, Victoria))$$

The engineering work agent is also modelled as an abstract agent which is assumed to respond to a query within some bounded number of timesteps, e.g., n timesteps:

$$\begin{gathered} G(B_e ASK(p, e, RouteList(start\_station, destination\_station, \\ [\ station_1 \mid station_2 \mid \ldots \mid station_n\ ])) \rightarrow \\ X^n\ B_e\, TELL(e, p, RouteList(start\_station, destination\_station, \\ [\ station_1 \mid station_2 \mid \ldots \mid station_n\ ])) \end{gathered}$$

where $[\ station_1 \mid station_2 \mid \ldots \mid station_n\ ]$ is a list of intermediate stations from the $start\_station$ to the $destination\_station$, and the response from the engineering agent indicates that the route from the $start\_station$ to the $destination\_station$ via $station_1,\ station_2\ , \ldots, station_n$ is open.

The system designer may wish to verify that the proposed design of the route planning agent, together with the assumed or known properties of the engineering work agent, is able to respond to a given number of user queries arriving within a specified interval, within a specified period of time. For a typical routing query, e.g., for an abstract user agent u_i asking for a route between station1 and station2, we can verify that response is received within n timesteps:

$$\begin{gathered} G(B_{u_i} ASK(u_i, p, Route(s1, s2)) \rightarrow \\ X^n\ B_{u_i}\, TELL(p, u_i, RouteList(s1, s2, [t_1 | t_2 | \ldots | t_n]))) \end{gathered}$$

Table 3 reports experimental results for a multi-agent system consisting of a planner agent, an engineering agent and varying number of user agents. In this experiment, we have used 6 stations connected by 3 different lines (a total of 7 facts) and the planner can derive 8 different routes. Different user agents in the system make queries about different routes at different times in the interval $[1, 10]$. For example, the user agent u_i may request a route between *Marble Arch* and *Victoria*:

$$ASK(u_i, p, Route(MarbleArch, Victoria))$$

and receive the reply

$$TELL(p, u_i, RouteList(MarbleArch, Victoria, [BondStreet | GreenPark]))$$

The *timesteps* value in Table 3 gives the maximum number of timesteps necessary to return a route to a user agent under the specified system load.

# user agents	# timesteps	CPU Time
2	21	00:00:39
4	29	00:03:56
5	33	00:08:50

Table 3. Resource requirements for the route planning example

6 Related Work

There has been considerable work on the execution properties of rule-based systems, both in AI and in the active database community. In AI, perhaps the most relevant is that of Chen and Cheng on predicting the response time of OPS5-style production systems. In [16], they show how to compute the response time of a rule-based program in terms of the maximum number of rule firings and the maximum number of basic comparisons made by the Rete network. In [17], Cheng and Tsai describe a tool for detecting the worst-case response time of an OPS5 program by generating inputs which are guaranteed to force the system into worst-case behaviour, and timing the program with those inputs. However, the results obtained using these approaches are specific to a particular rule-based system (OPS5 in this case), and cannot easily be extended to systems with different rule formats or rule execution strategies. Nor are they capable of dealing with the asynchronous inputs found in communicating rule-based systems. The problem of termination and query boundedness has also been studied in deductive databases [18]. However, again this work considers a special (and rather restricted with respect to rule format and execution strategy) class of rule-based systems.

In [19] the Datalaude system is presented, which essentially implements a Datalog interpreter in Maude. However the encoding of rules and rule execution strategy is very different from that proposed in this paper, in using functional modules and implementing a backward chaining rule execution strategy. The aim of the Datalaude project is not to analyse Datalog programs as such, but to provide a fast and 'declarative' (in the sense of functional programming) specification of memory management in Java programs (the example application in [19] uses Datalog facts represent information about references, and some simple rules ensure transitivity of the reference relation).

There has been considerable work on the use of abstraction in model-checking, e.g., [11, 12]. These approaches use a mapping between an abstract transition system and a concrete program. Depending on this mapping, verification results may be correct but not complete. In contrast, our approach uses a very specific kind of abstraction, which replaces a concrete agent with an abstract one that implements guarantees of its response time behaviour. If those guarantees are correct, then our approach gives both correct and complete results. Agents can be modelled as abstract if their response time guarantees have already been verified or the system designer is prepared to assume them.

7 Conclusion

We described an automated verification framework for communicating resource-bounded reasoners which takes a set of agents specified in terms of facts and Horn clause rules

and automatically produces a Maude [8] specification of the system which can be efficiently verified. We illustrated the scalability of our approach by comparing it to results presented in [9] for a synthetic distributed reasoning problem. We also showed how to further improve scalability by using abstract agents specified in terms of temporal epistemic formulas.

The tool described in the paper is a simple prototype. In future work, we plan to extend the language for specifying the rules of concrete agents to include function terms, and introduce a language for specifying reasoning strategies.

References

1. Bordini, R.H., Hübner, J.F., Vieira, R.: ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer-Verlag (2005)
2. Bordini, R., Fisher, M., Visser, W., Wooldridge, M.: State-space reduction techniques in agent verification. In Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M., eds.: Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, ACM Press (2004) 896–903
3. Adjiman, P., Chatalic, P., Goasdoué, F., Rousset, M.C., Simon, L.: Distributed reasoning in a peer-to-peer setting. In de Mántaras, R.L., Saitta, L., eds.: Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'04), Valencia, Spain, IOS Press (August 2004) 945–946
4. Claßen, J., Eyerich, P., Lakemeyer, G., Nebel, B.: Towards an integration of Golog and planning. In: Proceedings of the 20th international joint conference on Artifical intelligence (IJCAI'07), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2007) 1846–1851
5. Alechina, N., Jago, M., Logan, B.: Modal logics for communicating rule-based agents. In Brewka, G., Coradeschi, S., Perini, A., , Traverso, P., eds.: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), IOS Press (2006) 322–326
6. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in model checking. In: Computer Aided Verification. (1998) 521–525
7. Hirtle, D., Boley, H., Grosof, B., Kifer, M., Sintek, M., Tabet, S., Wagner., G.: Schema Specification of RuleML 0.91. http://ruleml.org/0.91/ (2006)
8. Clavel, M., Eker, S., Lincoln, P., Meseguer, J.: Principles of maude. Electr. Notes Theor. Comput. Sci. **4** (1996)
9. Alechina, N., Logan, B., Nga, N.H., Rakib, A.: Verifying time and communication costs of rule-based reasoners. In Peled, D., Wooldridge, M., eds.: Model Checking and Artificial Intelligence, 5th International Workshop MoChArt 2008, Patras Greece, July 21, 2008. Revised Selected and Invited Papers. Volume 5348 of LNCS., Berlin/Heidelberg, Springer (2009)
10. Holzmann, G.J.: On-the-fly model checking. ACM Computing Surveys **28**(4) (December 1996)
11. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: In Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1992) 342–354
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: In Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages. (1977) 238–252
13. Culbert., C.: CLIPS reference manual. NASA. (2007)

14. Friedman-Hill., E.J.: Jess, The Rule Engine for the Java Platform. Sandia National Laboratories. (2008)
15. Tzafestas, S., Ata-Doss, S., Papakonstantinou., G.: Knowledge-Base System Diagnosis, Supervision and Control. New York, London, Plenum Press (1989)
16. Chen, J.R., Cheng, A.M.K.: Predicting the response time of OPS5-style production systems. In: Proceedings of the 11th Conference on Artificial Intelligence for Applications, IEEE Computer Society (1995) 203
17. Cheng, A.M.K., yen Tsai, H.: A graph-based approach for timing analysis and refinement of OPS5 knowledge-based systems. IEEE Transactions on Knowledge and Data Engineering **16**(2) (2004) 271–288
18. Brodsky, A., Sagiv, Y.: On termination of Datalog programs. In: International Conference on Deductive and Object-Oriented Databases (DOOD). (1989) 47–64
19. Alpuente, M., Feliu, M.A., Joubert, C., Villanueva, A.: Defining Datalog in rewriting logic. In: Proceedings of the 19th International Symposium on Logic-Based Program Synthesis and Transformation LOPSTR09. (2009)