Model Checking Ontology-Driven Reasoning Agents using Strategy and Abstraction

Abdur Rakib[1] and Rokan Uddin Faruqui[2,3]

[1] Department of Computer Science and Creative Technologies
The University of the West of England, Bristol, UK
Rakib.Abdur@uwe.ac.uk
[2] Department of Computing and Software
McMaster University, Canada
[3] Department of Computer Science and Engineering, University of Chittagong, Bangladesh.
rufaruqui@cu.ac.bd

Abstract. We present a framework for the modelling, specification and verification of ontology-driven multi-agent rule-based systems (MASs). We assume that each agent executes in a separate process and that they communicate via message passing. The proposed approach makes use of abstract specifications to model the behaviour of some of the agents in the system, and exploits information about the reasoning strategy adopted by the agents. Abstract specifications are given as Linear Temporal Logic (LTL) formulas which describe the external behaviour of the agents, allowing their temporal behaviour to be compactly modelled. Both abstraction and strategy have been combined in an automated model checking encoding tool TOVRBA for rule-based multi-agent systems which allows the system designer to specify information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The TOVRBA tool generates an encoding of the system for the Maude LTL model checker, allowing properties of the system to be verified.

Keywords: Semantic Web, Ontology, Rule-based reasoning, Multi-agent Systems, Maude, Rewriting logic, Model Checking

1 Introduction

Rule-based systems have been studied for decades and traditionally rules have been used in theoretical computer science, databases, logic programming, and in particular, in artificial intelligence (AI), to describe expert systems, robot behavior, and behaviour of business. They have found significant application in practice and there has been a move of rule-based systems into intelligent agents and vice versa. Specifically, there has recently been considerable interest in Semantic Web and rule-based approaches to various aspects of agent technology. The integration of the Semantic Web and intelligent agents research has been realized [35], and intelligent agents are considered as a promising approach towards realizing the Semantic Web vision [21]. The concept of agents, in the setting of this paper is used to refer to autonomous reasoning agents, where agents are capable of reasoning about their behaviour (using a knowledge base and inference rules) and interactions (capable of communicating with each other). An intelligent agent is called rule-based if its behaviour and/or its knowledge is represented using rules.

The main emphasis of the existing research on Semantic Web rule-based systems is how can ontologies be utilised for modelling and enhancing level of interoperability and usability of applications. However, that is not sufficient to make Semantic Web rule-based systems a key feature technology that has been moving into safety-critical domains including healthcare [9, 26], where the lives of patients may be at stake, and where technology is increasingly being used to help ensure compliance with clinical guidelines. For example, in a non-time critical environment, where small delays due to response time are not an issue, a rule-based system may respond to queries without any concern or consideration of the time needed for reasoning. However, there are many cases where the time taken to do the reasoning is of critical importance. As an example, in a multi-agent rule-based system, an agent may be able to produce reasonably correct information (e.g., a health planner agent can infer a patient's current status based on the information it has received from other heart rate and/or blood pressure measurement agents) and send the information to another agent (e.g., patient's GP) to reach its goal, but if the overall reasoning and interaction take too long the result may be irrelevant, e.g., patient might already be in a very dangerous condition or even die before any action can be taken.

Therefore, while rule-based systems are rapidly becoming an important component of Semantic Web application, the resulting system behaviour and the resources required to realize them, namely, how to ensure the correctness of rule-based designs (will a rule-based system produce the correct output for all legal inputs), termination (will a rule-based system produce an output at all) and response time (how much computation will a rule-based system have to do before it generates an output) can be difficult to predict. These problems become even more challenging for distributed rule-based systems, where the system being designed or analysed consists of several communicating rule-based programs which exchange information via messages. A communicated fact may be added asynchronously to the state of a rule-based system while the system is running, potentially triggering a new strand of computation which executes in parallel with current processing. In order to provide response time guarantees for such systems, we must know how much time a rule-based system needs to perform the required reasoning. Furthermore, for a rule-based system running on resource-bounded devices e.g., PDAs, smartphones or other mobile devices, the number of messages exchanged may also be a critical factor.

This paper extends our previous work [32] and the main contributions of this paper are: first, to present an approach for the specification and verification of an ontology driven system that supports automated verification of time and communication requirements in distributed Semantic Web rule-based agents. We consider distributed problem-solving in systems of communicating rule-based agents, and ask how much time (measured as the number of rule firings) and how many message exchanges it takes the system to find a solution. We use standard model checking techniques to verify interesting properties of such systems, and show how the Maude LTL model checker [15] can be used to verify properties including response-time guarantees of the form: *if the system receives a query, then a response will be produced within n time steps*. Second, to allow larger systems to be verified, the proposed approach makes use of abstract specifications to model the behaviour of some of the agents in the system, and exploits information about the reasoning strategy adopted by the agents. Abstract specifications are given as Linear Temporal Logic (LTL) formulas which describe the external behaviour of the agents (the response time behaviour of the agent), allowing their temporal behaviour to be compactly modelled. We explain how our abstraction approach gives both correct and complete results. Third, to illustrate the scalability of our approach we reimplemented an example scenario introduced in [2] and provide a more detailed complementary analysis of previously presented results [4], and presenting results for a more complex multi-agent home health care monitoring alarm system adapted from [30].

The remainder of the paper is structured as follows. In Section 2 we provide an overview of ontology and how agents are modelled using ontology-driven rules, followed by basics of model checking technique and the Maude LTL model checker. In Section 3 we present a scalable compositional modelling and verification framework of distributed agents. In Section 4 we briefly describe a prototyping tool Tovrba for translating ontology based specification of the agents into Maude. In Section 5 we present Maude encoding. In Section 6 we model a home health care monitoring system and present some experimental results using Tovrba, the scalability of the new approach is also illustrated using a distributed reasoning problem which can be easily parameterised to increase or decrease the problem size. We discuss related work in Section 8 and conclude in Section 9.

2 Preliminaries

2.1 Ontology-driven Horn clause rules

Ontologies and rules play a central role in the design and development of Semantic Web applications. An ontology is an explicit formal specification of a conceptualization which defines certain terms of a domain and the relationships among them [20]. The Web ontology language OWL is a semantic markup language for ontologies that provides a formal syntax and semantics for them. The W3C declared two different standardizations for OWL: OWL 1 and OWL 2 [28]. Both the description logic based OWL 1 and OWL 2 are decidable fragments of First Order Logic (FOL); however, the expressive power of OWL 1 is strictly limited to certain tree structure-like axioms [19]. For instance, a simple rule: `livesIn(?x, ?y), locatedIn(?y,?z)` → `hasCountry(?x,?z)` can not be modeled using OWL 1 axioms. Although OWL 2 can express this *country* rule indirectly, many rules are still not possible to model using OWL 2 axioms. Function-free Horn clause rules can remove such restrictions while being decidable but they are restricted to universal quantification and no negation. A combination of OWL 2 with rules offers a more expressive formalism for building Semantic Web applications. Several proposals have been made to combine rules with ontologies. We use one of them, the SWRL that extends OWL DL by adding new axioms, namely Horn clause rules. Although SWRL was a proposed extension for OWL 1, it can be used as a rule extension for OWL 2 [18]. We combine a set of SWRL rules with the set of OWL 2

RL axioms and facts to build our ontology. Since OWL 2 RL is based on DLP, the set of axioms and facts of an OWL 2 RL ontology can be translated to Horn clause rules [19]. Translations of some of the OWL 2 RL axioms and facts into rules are given in Table 1. In the second column, complete DL statements are given which are constructed by the corresponding OWL 2 RL axioms and facts to illustrate the translation. The translation of SWRL rules is straightforward because they are already in the Horn clause rule format.

OWL 2 Axioms and Facts	**DL Syntax**	**Horn clause rule**
`ClassAssertions`	a:C	C(a)
`PropertyAssertion`	$\langle a,b\rangle : P$	$P(a,b)$
`SubClassOf`	$C \sqsubseteq D$	$C(x) \rightarrow D(x)$
`EquivalentClasses`	$C \equiv D$	$C(x) \rightarrow D(x)$ $D(x) \rightarrow C(x)$
`EquivalentProperties`	$P \equiv Q$	$Q(x,y) \rightarrow P(x,y)$ $P(x,y) \rightarrow Q(x,y)$
`ObjectInverseOf`	$P \equiv Q^-$	$P(x,y) \rightarrow Q(y,x)$ $Q(y,x) \rightarrow P(x,y)$
`TransitiveObjectProperty`	$P^+ \sqsubseteq P$	$P(x,y), P(y,z) \rightarrow P(x,z)$
`SymmetricObjectProperty`	$P \equiv P^-$	$P(x,y) \rightarrow P(y,x)$
`Object/DataUnionOf`	$C_1 \sqcup C_2 \sqsubseteq D$	$C_1(x) \rightarrow D(x)$ $C_2(x) \rightarrow D(x)$
`Object/DataIntersectionOf`	$C \sqsubseteq D_1 \sqcap D_2$	$C(x) \rightarrow D_1(x)$ $C(x) \rightarrow D_2(x)$
`Object/DataSomeValuesFrom`	$\exists P.C \sqsubseteq D$	$P(x,y), C(y) \rightarrow D(x)$
`Object/DataAllValuesFrom`	$C \sqsubseteq \forall P.D$	$C(x), P(x,y) \rightarrow D(y)$
`Object/DataPropertyDomain`	$\top \sqsubseteq \forall P^-.C$	$P(y,x) \rightarrow C(y)$
`Object/DataPropertyRange`	$\top \sqsubseteq \forall P.C$	$P(x,y) \rightarrow C(y)$

Table 1: Translation of OWL 2 RL axioms and facts into Horn clause rules

2.2 Model checking using Maude

The model based verification approach uses model checking techniques, which are based on the semantics of the specification language. Applying model checking to a design comprises three components. First, a detailed description M (model) of the system has to be given using the description language of the model checker. Second, a property φ of the system has to be given by means of some property specification language, e.g., linear time logic (LTL) or computation tree logic (CTL). The expressive power of LTL and CTL is not comparable. While there are properties that can be expressed both in LTL and CTL, there are also properties exist that can be expressed in LTL but cannot be expressed in CTL and vice-versa [11, pp. 30–31]. Third, once the model M and the system property φ are given, a model checker will check whether or not $M \models \varphi$. The third phase is completely automatic. Thus the model checking problem can be stated simply as given a formula φ of some logical language and a model M, to determine whether or not φ is valid in the model M. In Maude [15], a rewriting theory $\mathcal{R} = (\Sigma, E, R)$, consists of a signature Σ, a set E of equations, and a set R of rules. The static part of a system is specified in an equational sub-logic of rewriting logic (membership equational logic) by means of equations E. The system dynamics (concurrent transitions or inferences) is specified by means of rules R that rewrite terms, representing parts of the system, into other terms. The rules in R are applied *modulo* the equations in E. Maude computes normal form of a term by applying equations from left to right iteratively, then an applicable rewrite rule is arbitrarily chosen and applied from left to right. Thus, data types are defined algebraically by equations and the dynamic behaviour of a system is defined by rewrite rules which describe how a part of the state can change in one step. A rewrite theory is often non-deterministic and could exhibit many different behaviours. In Maude, a term is either a constant, a variable, or the application of an operator to a list of argument terms. A ground term is a term containing no variables, but only constants and operators. Like any other model checking tool, verification in Maude requires a system specification and a property specification. The system specification is provided by a rewrite theory, whereas the property specification is given by LTL formulas.

3 A modelling and verification framework of distributed agents

We adapt the model of distributed agents presented in [2]. A distributed reasoning system consists of n_{Ag} (≥ 1) individual reasoners or *agents*. Each agent is identified by a value in $\{1, 2, \ldots, n_{Ag}\}$ and we use variables i and j over $\{1, 2, \ldots, n_{Ag}\}$ to refer to agents. An agent in the system is either concrete or abstract. Each concrete agent has a program, consisting of Horn clause rules, and a working memory, which contains facts (ground atomic formulas) representing the initial state of the system. The logical model presented in [2] is based on propositional language, however the restriction to propositional rules is not a very drastic assumption: if the rules do not contain functional symbols and we can assume a fixed finite set of constant symbols, then any set of first-order Horn clauses and facts can be encoded as propositional formulas. In this framework, concrete agents in a system also use different conflict resolution strategies. The behaviour of each abstract agent is represented in terms of a set of temporal epistemic formulas. That is, abstract specifications are given as LTL formulas which describe the external behaviour or the response time behaviour of some of the agents in the system. The overall rationale for choosing this abstract agent notion is discussed below in Section 3.1. The agents (concrete and abstract) execute synchronously. We assume that each agent executes in a separate process and that agents communicate via message passing. We further assume that each agent can communicate with multiple agents in the system at the same time. In the following sections, we describe in more detail how we model the concrete and abstract agents.

3.1 Managing complexity through strategy and abstraction

We would like to be able to verify properties of systems consisting of arbitrary numbers of complex communicating reasoners. However, our experience in [2, 3] has indicated that verifying such large, complex reasoning systems is infeasible with current model checking techniques. 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. The 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 the state space size, model checking performance is heavily dependent on the branching factor of states in the reachable state space as well as on 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.[4] 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). The model checking performance based on depth-first search can also vary dramatically from good to worst. In both the cases, verification of true formulas take longer than verification of false formulas since a model checker will find a counterexample faster than it takes to explore the whole model.

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. Abstract specifications are given as Linear Temporal Logic (LTL) formulas which describe the external behaviour of some of the agents, allowing their temporal behaviour to be compactly modelled. Conversely, reasoning strategies allow the detailed specification of the ordering of steps in the agent's reasoning process. 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.

3.2 Ontology-driven rules

The use of first-order rules increases the expressiveness of the framework in [2], and makes it easier to model complex real world scenarios. To formally represent a domain

[4] Even with on-the-fly model-checking [22], the model checker has to explore the state space at least until the solution depth.

model we use OWL 2 RL ontology augmented with SWRL rules, which is ultimately translated into a set of Horn-clause rules to design the desired multi-agent system following the concept presented in Section 2.1. Section 4 provides more detailed discussion of the translation process. However, the verification framework is standalone, and it is not necessary that rules will only be derived from ontologies, a system designer can model and write a set of rules to construct the systems using any other approaches. The use of ontology-driven rules simply provides a more natural way to think about and model real world rules and exploit this benefit. In addition, existing tools, including Protégé [1], support the design of OWL 2 RL and SWRL based ontologies, making it easier to model rule-based agents using semantic rules.

3.3 Description of concrete agents

The two main components of rule-based agents are the knowledge base (KB) which contains a set of first-order Horn-clause rules and the working memory (WM) which contains a set of facts that constitute the current (local) state of the system. The state of an agent also contains a communication counter, which is discussed below. Another component of a rule-based system is the inference engine which reasons over rules when the application is executed.

```
Rule::='<' Priority ':'Atoms '→' Atom'>'
Atoms::=Atom {, Atom}*
Atom::=standardAtom | commmunicationAtom
standardAtom::=Predicate(Term)
              | Predicate(Term, Term)
communicationAtom::=
     'Ask(' i ',' j ',' standardAtom ')'
   | 'Tell(' i ',' j ',' standardAtom ')'
Priority::=N≥0
N≥0 ::= 0|1|2|...
i::=1|2|...|nAg
j::=1|2|...|nAg
Predicate::= Person|hasCarer|Raining|...
Term::=Constant|Variable
Constant::='Ann|'Bob|'30|'P001|...
Variable::=?x|?temp|?name|...
```

Listing 1.1: Abstract syntax for concrete agent's rules

The inference engine may have some reasoning strategies to handle cases when multiple rule instances are eligible to fire. The agents use the refractory rule firing technique, i.e., each rule instance is fired only once. In Listing 1.1, we specify the abstract syntax for concrete agents' rules using a BNF. In this notation, the terminals are quoted, the non-terminals are not quoted, alternatives are separated by vertical bars, and components that can occur zero or more times are enclosed braces followed by a superscript asterisk symbol $(\{\ldots\}^*)$. In other words, the rules of a concrete agent have the plain text format: $< n : P_1, P_2, \ldots, P_n \rightarrow P >$, where n is a constant that represents the annotated priority of the rule and the P_i's and P are first-order atoms. If an agent i has this rule, the antecedents $P_1, P_2, \ldots, P_n$ match with the facts in the agent's working memory and the consequent P is not in the agent's working memory in a given state s, then the agent can fire the matching rule instance which adds the consequent to the agent's working memory in the successor state s'.

Model of communication We assume a simple query-response scheme based on asynchronous message passing for agent communication. Each agent's rules may contain two distinguished communication atoms: $Ask(i, j, P)$, 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; for example,
$< n : P_1, P_2, \ldots, P_n \rightarrow Ask(i, j, P) >$,
whereas agent j may have an Ask or a $Tell$ with arguments (i, j, P) in the antecedent of the rule; for example, $< n : Tell(i, j, P) \rightarrow P >$ is a well-formed rule (we call it trust 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. When a rule has either an Ask or a $Tell$ as its consequent, we call it a communication rule. All other rules are known as deduction rules. These include rules with Asks and $Tell$s in the antecedent as well as rules containing neither an Ask nor a $Tell$.

We assume that the state, for each agent i, contains a communication counter, which starts with value 0 and incremented by 1 each time while interacting (sending/receiving a message) with other agents. After the counter reaches its limit, say $n_C(i)$, agent i cannot perform any more communication actions. The exchange of information between agents work like this: if an $Ask(i, j, P)$ (or a $Tell(i, j, P)$) is in agent i's working memory in a given state, $Ask(i, j, P)$ (or $Tell(i, j, P)$) is not in the working memory of agent j, and agent j has not exceeded its communication bound then in the successor state, $Ask(i, j, P)$ (or $Tell(i, j, P)$) can be added to agent j's working memory, and its communication counter incremented.

Possible actions of an agent The semantics of the agents' language is based on transition systems and follow the approach of [2]. We view the process of producing new facts from existing facts as a sequence of states of an agent, starting from an initial state, and producing the next state by one of the following actions:

- **Rule:** firing a matching rule instance in the current sate;
- **Comm:** if agent i has an $Ask(i, j, P)$ (or a $Tell(i, j, P)$) in its current state, then agent j can copy it to its next state provided j's communication counter has not exceeded $n_C(j)$ value;
- **Idle:** which leaves its configuration unchanged.

That is, each transition (result of an action) corresponds to a single execution step and takes an agent from one state to another. States consist of the rules, facts, and communication counter of the agent. A *step* of the whole system is composed of the actions of each agent, in parallel. We measure time requirements for a problem as the number of such system steps. The key idea underlying the logical approach presented in [2] of rule-based systems is to define a formal logic that axiomatizes the set of transition systems, and it is then used to state various properties of the systems.

Reasoning strategies We assume that each concrete agent 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 framework (and the TOVRBA tool presented in Section 4) supports a set of standard conflict resolution strategies often used in rule-based systems including, Rule ordering, Depth, Breadth, Simplicity, and Complexity [13, 16, 36]. Different agents in the system may use different types of reasoning strategies. To allow the implementation of reasoning strategies, each atom of a rule is associated with a time stamp which records the cycle at which the atom was added to the working memory. In order to achieve this, the internal configurations of the rules in the Maude specification (cf. Section 5) follow the syntax given below:

$$< n : [t_1 : P_1], [t_2 : P_2], \dots, [t_n : P_n] \rightarrow [t : P] >$$

where the t_i's and t represent time stamps of atoms. When a rule instance of the above rule is fired, its consequent atom (ground instance of P) will be added to the working memory with time stamp $t = t' + 1$, i.e., t will be replaced by $t' + 1$, where t' is the current cycle time of the system.

3.4 Abstract agents

An abstract agent consists of a working memory and a behavioural specification. The behaviour of abstract agents is specified using a subset of LTL formulas extended with belief operators. The general form of the formulas used to represent the external behaviour of an abstract agent i is given in Listing 1.2.

$$\begin{array}{rcl}
\rho & ::= & X^{\leq n}\varphi_1 \mid G(\varphi_2 \rightarrow X^{\leq n}\ \varphi_3) \\
\varphi_1 & ::= & B_i\ Ask(i,j,P) \mid B_i\ Tell(i,j,P) \\
 & & \mid B_i\ Ask(j,i,P) \mid B_i\ Tell(j,i,P) \\
 & & \mid B_i\ P \\
\varphi_2 & ::= & B_i\ Ask(j,i,P) \mid B_i\ Tell(j,i,P) \\
\varphi_3 & ::= & B_i\ Tell(i,j,P) \mid B_i\ Tell(i,k,P) \\
 & & \mid B_i\ Ask(i,j,P) \mid B_i\ Ask(i,k,P) \\
n & ::= & N_{\geq 1} \\
N_{\geq 1} & ::= & 1 \mid 2 \mid 3 \dots
\end{array}$$

Listing 1.2: Temporal epistemic formulas for abstract agents

In the formulas X is the 'next step' temporal operator, $X^{\leq n}$ is a sequence of n (or less) 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', i.e., the contents of its working memory. Formulas of the form $X^{\leq n}\varphi_1$ describe agents which produce a certain message or input to the system within n time steps. These formulas (partly) describe proactive behaviour of an agent. For example, the formula $X^{\leq n} B_i\ \mathit{Tell}(i, j, P)$, which describes abstract behaviour of agent i produces a $\mathit{Tell}(i, j, P)$ within n time steps. That is i tells about P to j proactively by generating a $\mathit{Tell}(i, j, P)$ message in the interval $[1, n]$ thinking that it might be useful for j. In other words, i tells j about P without being asked. A formula φ_1 of the form $B_i\ Ask(i, j, P)$ or $B_i\ \mathit{Tell}(i, j, P)$ results in communication with the other agent as follows: when the beliefs appear (as an Ask or a Tell) in the abstract agent i's working memory, they are also copied to agent j's working memory at the next step. A formula φ_1 of the form $B_i\ P$ representing a belief involving an atom P (other than Ask and Tell), which may also appear in the abstract agent i's working memory within n time steps. This is not critical to how abstract agents interact with communication; however it describes abstract agent i's own behaviour.

The $G(\varphi_2 \rightarrow\ X^{\leq n}\ \varphi_3)$ formulas describe agents which are always guaranteed to reply to a request for information within n time steps. We interpret the formula $G(B_i\ Ask(j, i, P) \rightarrow\ X^{\leq n}\ B_i\ \mathit{Tell}(i, j, P))$ as follows: if t is the time stamp when abstract agent i came to believe $Ask(j, i, P)$ (i.e., $Ask(j, i, P)$ appears in the agent i's working memory), then the atom $\mathit{Tell}(i, j, P)$ must appear in the working memory of agent i within $t + n$ steps. The atom $\mathit{Tell}(i, j, P)$ is then copied to agent j's working memory at the next step. The other possible combinations of Ask and Tell in places of φ_2 and φ_3 in the $G(\varphi_2 \rightarrow\ X^{\leq n}\ \varphi_3)$ formulas can be interpreted in a similar way.

The language described above for the abstract agents is independent of the language of the concrete agents. Note, however, that we do not need the full language of LTL (for example, the Future (F) or Until (U) operator) in order to specify these abstract agents. This is because, a formula such as, e.g., $F\ B_i\ Ask(j, i, P)$ which states that the atom $Ask(j, i, P)$ must be appeared in the agent i's working memory at some time in the future, represents a form of temporal indeterminacy, which is not very helpful in our context.

3.5 Specifying systems at different levels of abstraction

In our framework, we assume that an agent in the system is either completely concrete or completely abstract. The representation of agents in the system are divided into two classes based on their behavioural specification. The system designer may have complete control over the internal behaviour of some agents in the system. The concrete agents class contains those agents. The remaining agents belong to the abstract agents class. In this step the designer identifies which agents he needs to design for what classes. The designer also determines the number of agents he needs to place in each class and their possible interactions. An agent can interact with one or more agents in the system, but not necessarily every agent interacts with every other agent in the system. The designer can consider the following different possible levels of system information in order to design and verify system properties.

1. The system designer may have detailed design information about the internal behaviour of some agents in the system including the initial facts in their working memories, their rules and the reasoning strategy. The remaining agents in the system are modelled using temporal epistemic formulas.

2. The system designer may have information of all the agents in the system including the initial facts in their working memories, their rules but no information at all about their reasoning strategy. This design gives the worst case model.
3. The system designer may have detailed information of all the agents in the system including the initial facts in their working memories, their rules and the reasoning strategy.

Both approaches (strategy and abstraction) have been combined in a prototyping tool TOVRBA for rule-based multi-agent systems which allows the designer to specify information about agents' interaction, behaviour, and execution strategy at different levels of abstraction. The TOVRBA tool generates an encoding of the system for the Maude LTL model checker, allowing properties of the system to be verified.

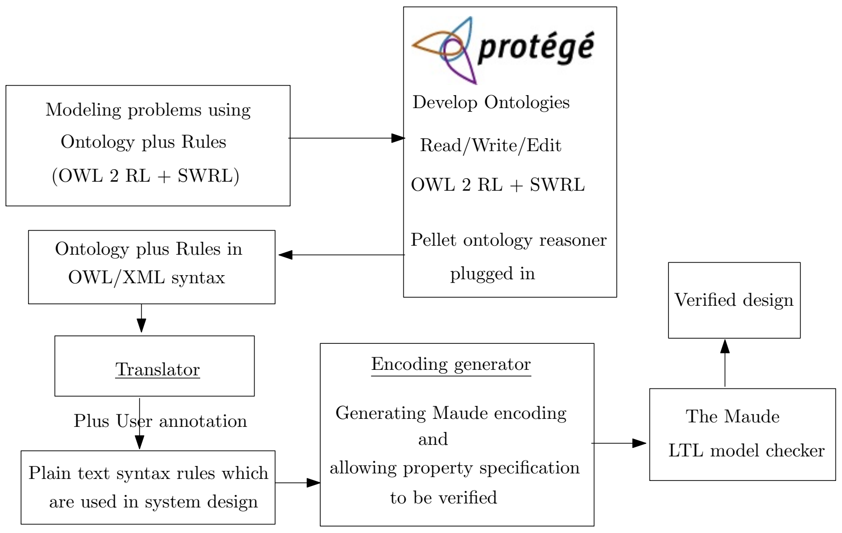

protégé
Modeling problems using Ontology plus Rules (OWL 2 RL + SWRL)
Develop Ontologies
Read/Write/Edit
OWL 2 RL + SWRL
Pellet ontology reasoner plugged in
Ontology plus Rules in OWL/XML syntax
Translator
Plus User annotation
Plain text syntax rules which are used in system design
Encoding generator
Generating Maude encoding and allowing property specification to be verified
The Maude LTL model checker
Verified design

Fig. 1: The TOVRBA tool architecture

3.6 Discussion of the abstraction approach

Our modelling approach presented above 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 [10, 12], which use a mapping between an abstract transition system and a concrete program. Depending on this mapping, verification results may be correct but not complete. By correct or conservative abstraction usually mean that if a formula is true in the abstract system, then it is true in the concrete system (but if a formula is false in the abstract system, it may not be false in the concrete system). 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. Complete or exact abstraction means that a formula is true in the abstract system if and only if it is true in the concrete system. Agents can be modelled as abstract if their response time guarantees have already been verified or the system designer is prepared to assume them.

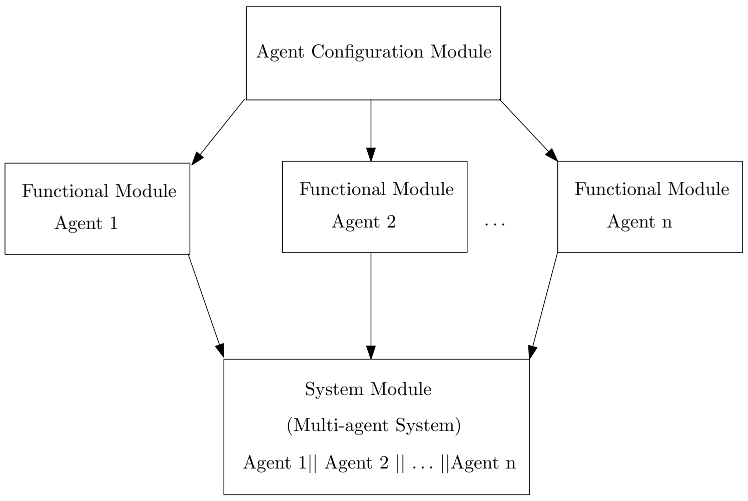

Agent Configuration Module
Functional Module Agent 1
Functional Module Agent 2
...
Functional Module Agent n
System Module
(Multi-agent System)
Agent 1|| Agent 2 || ... ||Agent n

Fig. 2: MAS implementation structure in Maude

4 A prototyping tool Tovrba

We use the Protégé [1] ontology editor and knowledge-base framework to build the ontologies augmented with SWRL rules while modelling a domain. The SWRL editor is integrated with Protégé and permits the interactive editing of SWRL rules. In order to encode an ontology-driven rule-based system using a Maude [15] specification and formally verify its interesting properties using LTL model checking, we first need to translate the ontology in the OWL/XML format to a set of simple plain text Horn clause rules. We developed a translator that takes as input an OWL 2 RL ontology in the OWL/XML format (an output file of the Protégé editor) and translates it to a set of plain text Horn clause rules. First, we take an OWL 2 RL ontology as an input and then invoke a DL reasoner to compute a complete class hierarchy. Then, we parse the inferred ontology that generates a set of OWL 2 RL axioms and facts. We use the OWL API [23] to parse the ontology and extract the set of axioms and facts. The design of the OWL API is directly based on the OWL 2 Structural Specifications and it treats an ontology as a set of axioms and facts which are read using the visitor design pattern. The DLP-based translation rules (cf. Section 2.1) are then recursively applied to generate equivalent plain text Horn clause rules for each axiom and fact. We also extract the set of SWRL rules using the OWL API which are already in the Horn clause rule format. First, atoms with corresponding arguments associated with the head and the body of a rule are identified and we then generate a plain text Horn clause rule for each SWRL rule using these atoms. The translated Horn clause rules of an ontology are then used to create agents of a multi-agent rule-based system using the Maude specifications. We then automatically verify interesting properties of the system using the Maude LTL model checker. The high-level architecture of the Tovrba tool is shown in Figure 1.

5 Maude encoding

We chose Maude [15] rewriting system and its 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, the variables that appear in a rule can be represented directly in the Maude encoding, without having to generate all ground instances resulting from possible variable substitutions.

```
fmod ACM is
 protecting NAT .
 protecting BOOL .
 protecting QID .
 sorts Constant Atom sAtom cAtom Term Rule Agenda WM .
 sorts TimeA TimeWM RepT RepTime Config .
 subsort Atom < WM .
 subsorts aAtom cAtom < Atom .
 subsort Rule < Agenda .
 subsort Qid < Constant .
 subsort TimeA < TimeWM .
 subsorts Constant < Term .
 subsort RepT < RepTime .
 ops com exec : -> Phase [ctor] .
 op [_ : _] : Nat Atom -> TimeA .
 op _ _ : WM WM -> WM [comm assoc] .
 op _ _ : TimeWM TimeWM -> TimeWM[comm assoc] .
 op _ _ : Agenda Agenda -> Agenda[comm assoc] .
 op Ask : Nat Nat sAtom -> cAtom .
 op Tell : Nat Nat sAtom -> cAtom .
   .
   .
   .
endfm
```

Listing 1.3: Sorts declaration and their relationships

We take advantage of Maude's modular structuring mechanisms to implement our systems design. We use a generic functional module and a set of functional and system modules to represent the system. The overall picture of our implementation is shown in Figure 2. Throughout this entire paper we will use verbatim texts to represent specification of the agents into Maude. Therefore, an agent i corresponds to `i` in Maude specification. Similarly, `Ask(i,j,P)` will have the same meaning as $Ask(i, j, P)$ and so on.

5.1 Agent configuration module

Each agent in a MAS has a configuration (local state) and the composition of all these configurations (local states) make the configuration (global state) of the MAS. The types necessary to implement the local state of an agent (working memory, program, reasoning strategy, message counters, time step etc.) are declared in a generic agent configuration functional module called `ACM`. The structure of the `ACM` is given in Listing 1.3. A number of Maude library modules such as `NAT`, `BOOL`, and `QID` have been imported into the `ACM` functional module. The modules `NAT` and `BOOL` are used to define natural and Boolean values, respectively, whereas the module `QID` is used to define the set of constant symbols (constant terms of the rule-based system). The set of variable symbols (variable terms of the rule-based system) are simply Maude variables of sort `QID`. Both variables and constants are subsorts of sort `Term`. Similarly, an atom is declared as an operator whose arguments are of sort `Term`, and returns an element of sort `Atom`. The sort `Atom` is declared as a subsort of the sort `WM` (working memory) etc. The data types presented in Listing 1.3 are manipulated using a set of equations. The equations are used for various purposes: for example, to check, whether or not a given atom (used to represent fact/predicate) is already in the agent's working memory, whether or not a rule instance is already in the agenda etc.

5.2 Implementation of agent modules

We model each concrete(and abstract) agent using a functional module `Concrete Agent-i` (and `AbstractAgen t-i`), which imports the `ACM` module defined above. The local configuration of an agent i is represented as a tuple:

`Si[A|RL|TM|M|RT|RT'|t|msg|syn]iS`

where `Si` and `iS` indicate start and end of a state of agent i. The variables `A` and `RL` are of sort `Agenda`, `TM` is of sort `TimeWM`, `M` is of sort `WM`, `RT` and `RT'` are of sort `RepTime`. Moreover, `t`, `msg`, and `syn` are of sort `Nat`. The variables `t`, `msg`, and `syn` have been used to represent respectively the time step, message counter, and a flag for synchronisation. Note that the structure of local configurations for both concrete and abstract agents are the same. This is just to maintain consistency of the shape of each agent's configuration. However, for example, the sort `RepTime` is of no use for concrete agents and its value is always empty for them.

5.3 Implementation of the MAS module

Computation steps of multi-agent systems are represented by transitions, which take systems from one configuration to subsequent ones. Each agent in the system has its own local state and the composition of all these local states comprises the configuration (global state) of the multi-agent system. In every configuration (global state), all agents proceed simultaneously. Each agent changes its next local configuration, possibly depending on the current local configurations of the other agents in the system. However, there can be an alternative interleaved execution model, where at most one agent is allowed to act at any one time. It depends on the modelled system which execution model (interleaved or synchronous) is more realistic. If we count time steps required by a system of agents to derive something, interleaved model gives rather pessimistic results because only one agent can 'think' at any single step and the rest are waiting. This makes sense if the agents run on the same processor. However if, as in most of our examples, agents are running on different processors and can 'think' in parallel, a synchronous model is more realistic.

```
mod MAS is
 protecting ConcreteAgent-i .
 protecting AbstractAgent-j .
   .
   .
   .
 sort masConfig .
 sort Phase .
 ops com exec : -> Phase .
 var phase : Phase .
 op _||_ : Config Config->Config[comm assoc] .
 op<_,_>: Config Phase->masConfig [ctor] .
 op comm : masConfig -> masConfig [ctor] .
   .
   .
   .
endm
```

Listing 1.4: Structure of MAS module

The `MAS` module imports all the agent modules and contains both functions and rewrite rules which are used to implement the dynamic behaviour of the system. The structure of the MAS module is given in Listing 1.4. The parallel composition of agent configurations in the system is achieved using the `_||_` operator. In the `MAS` module we declare a sort `masConfig` to represent the global configuration of the system. We then define an operator `<_,_>` whose first argument is the composition of all the local configurations of the system and the second argument is a phase, and it returns an element of sort `masConfig`. The `masConfig` moves through communication and execution phases. The communication phase simply says that if there is something to be communicated then do it, and then return to the execution phase.

The inference engine of concrete agents and the partial behaviour of abstract agents are implemented using a set of rules: `Generate`, `Choice`, `Apply`, `Idle`, and `Communication`. The `Generate` rule causes each agent to generate its conflict set. The `Choice` rule causes each agent to apply its reasoning strategy, the `Apply` rule causes each agent to execute the rule instances selected for execution, the `Idle` rule executes only when there are no rule instances to be executed (the application of the `Idle` rule advances the cycle time of the agent i, leaving everything else unchanged), and communication among agents is achieved using the `Communication` rule. When agents communicate with each other, one agent copies the communicated fact from another agent's working memory. Copying is only allowed if the fact to be copied is not already in the working memory of the agent intending to copy and it has not exceeded it's communication counter limit. For the sake of brevity, we do not describe the encoding in any further details here, we refer the interested reader to [31].

5.4 Verifying system properties

Model checking in Maude involves a Maude specification of a system together with a property of interest. A property is a LTL formula interpreted as a property of computations of the system (linear sequences of states generated by application of rewrite rules). A simple path from a given initial state s, to a state satisfying a property φ is a list of rules together with a state s' satisfying φ such that applying the rules starting with s leads to s'. One way to find a simple path is to model check the assertion that from s no state can be reached satisfying φ: modelCheck(s, $\sim$ F φ). If there is a reachable state satisfying φ, a counterexample will be returned. The counterexample contains the list of rules applied. Given a system module, say `MAS`, and an initial state, say s of sort `masConfig`, we can model check different LTL properties beginning at this initial state by doing the following:

- defining a new module, `ModelCheck-MAS`, that includes the module `MAS` and Maude's built-in module `MODEL-CHECKER` module as sub modules;
- giving a subsort declaration, `masConfig < State`, where `State` is a sort in the module `MODEL-CHECKER`;
- defining the syntax of the (target) state predicates we wish to use by means of constants and operators of sort `Prop`, a subsort of the sort `Formula` (i.e., LTL formulas) in the module `MODEL-CHECKER`;
- defining the semantics of the state predicates by means of equations.

The following `ModelCheck-MAS` system module defined in Listing 1.5 shows how we can define state predicates whose semantics are defined by appropriate equations.

```
mod ModelCheck-MAS is
 including MAS .
 including MODEL-CHECKER .
 subsort masConfig < State .
 op success : -> Prop .
 var C : Config .
 var phase : Phase .
 eq < Si[Ai:Agenda|RLi:Agenda|TMi:TimeWM|P Mi:WM|RTi:Rep
 TWM|RTi':RepTWM|t:Nat|msgi:Nat|syni:Nat]iS || C:Config,
  phase:Phase > |= success = true .
 eq C |= success = false [owise] .
 op init : -> masConfig .
 eq init = < S1[_|_|_|_|_|_|0|0|1]1S ||...||
             Si[_|_|_|_|_|_|0|0|1]iS ||...||
             Sn[_|_|_|_|_|_|0|0|1]nS,com > .

endm
```

Listing 1.5: Structure of ModelCheck-MAS module

In the state predicate semantics defined in Listing 1.5, the `masConfig` says that agent i's working memory contains a ground atom `P`. The remaining information of the configuration is specified using Maude's on-the-fly variable declaration. Note however that, the initial state must contain information using ground terms only. In the `ModelCheck-MAS` module the initial system state is represented using `init`, where all the `'_'` placeholders used in the configuration represent ground terms. Once the semantics of each of the state predicates has been defined, given an initial state `init`, we can model check any LTL formula, say φ, involving such predicates. We do so by executing in Maude, the command `reduce modelCheck(init,` φ`)`, where φ could be, for example, `[] success,<> success, <> ~success` etc.(`[]` stands for the global LTL operator G and <> stands for the future LTL operator F). Two things can then happen: if the property holds, then we get the result true; if it does not, we get a counterexample.

5.5 Analysis of the Maude implementation

When implementing reasoning strategies which involve time stamps of atoms, it is convenient to be able to associate a time stamp to each pattern. To achieve this, we have declared the sort `TimeWM` in the above encoding. However, in the encoding we maintained both the sorts `TimeWM` and `WM` simultaneously. In this section, we explain why. Let us suppose that each agent uses `TimeWM` as its only working memory. When agents generate their conflict sets, they check whether consequents of rule instances are already present in their working memory. If so, then these rule instances will not be added to their agendas. Similarly, when agent fires a rule instance or receives a message from another agent, it will make sure these atoms are not present in its working memory. For example, suppose an atom `P` with time stamp `t1` is already added to the working memory of an agent i. That is `[ t1 : P ]` is already present in `TimeWM`. Sometimes later, say at time `t2 (> t1)`, agent i needs to check whether `[ t2 : P ]` is already present in its working memory. It is apparent that the elements `[ t1 : P ]` and `[ t2 : P ]` of `TimeWM` are distinct because `t1` $\neq$ `t2`. However, the atom `P` is common to both of them. Therefore, to ensure that working memory does not contain duplicate atom it is necessary to ensure that the second part `P` of `[ t2 : P ]` is not already present in the working memory. This can be accomplished in one of two ways. One way is to compare the second part of `[ t2 : P ]` with the second part of each element `[ tk : P ]` of `TimeWM`. In order to implement this approach, some Maude conditional equations are required. However, the execution of additional conditional equations slows down the computation. Another way is to maintain a duplicate working memory `WM` which contains all the atoms of the form `P`. Whenever an element `[ t : P ]` is added to `TimeWM`, the corresponding atom `P` will be added to `WM`. In other words `TimeWM` and `WM` is updated simultaneously. Thus it is only necessary to check whether the second part `P` of `[ t2 : P ]` is already present in `WM` or not. Therefore, although maintaining only one working memory is enough, we use duplicate working memory for efficiency purposes.

6 Case study 1: Home health care monitoring alarm system

In this section, to illustrate the application of the framework we consider the following scenario of a home health care monitoring alarm system adapted from [30]. We built a home health care ontology using OWL 2 RL and SWRL from the scenario using Protégé [1]. A fragment of the ontology is depicted in Fig 3.

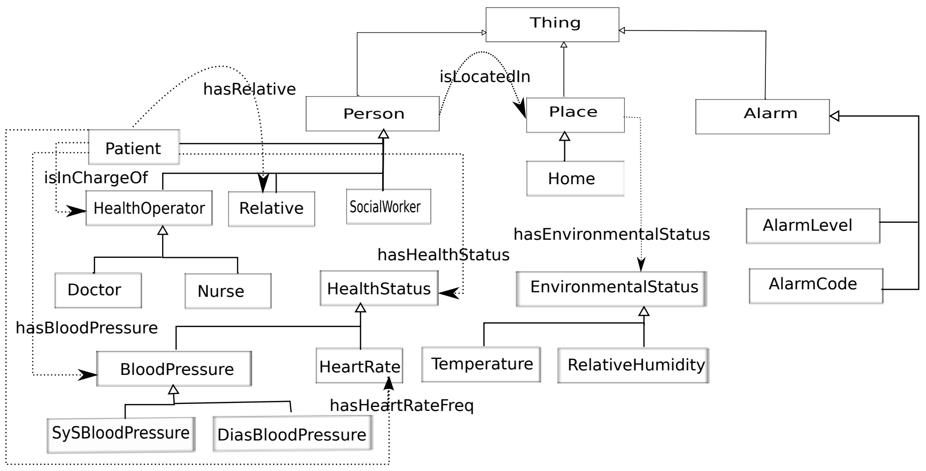


Fig. 3: Home health monitoring ontology

The dotted lines represent object/data properties between classes and solid lines represent "subclass" relations. A snapshot of an individual of the class "Patient" is given in Fig 5 (b) which clearly shows associated object and data properties with "Tracy". Static behaviour of the system is captured using OWL 2 RL and dynamic behaviour of the system is captured using SWRL rules. Some SWRL rules are given in Fig 5 (a). The prototyping tool TOVRBA translates the ontology into a set of Horn clause rules. The translated Horn clause rules of the ontology are then used to create agents of a multi-agent rule-based system using the Maude specification. The system consists of several concrete and abstract agents. The concrete agents in the system include a number of home healthPCs, pc_is, and a central Health Planner, p. Each pc_i agent in the system is connected with two body sensor agents such as a Blood pressure monitoring agent, b_i, and a Heart rate monitoring agent, h_i. The agents b_is and h_is are modelled as abstract agents. All the home healthPC agents pc_is can communicate with the agent p, which is located at the health centre. The agent p can also communicate with various other agents in the system including doctors, nurses, relatives of patients, and an emergency operator. The over-all picture of the system is depicted in Figure 4.

The abstract agents b_i and h_i measure the Blood pressure and Heart rate information of a patient and inform to the corresponding home healthPC, pc_i, as messages of the form:

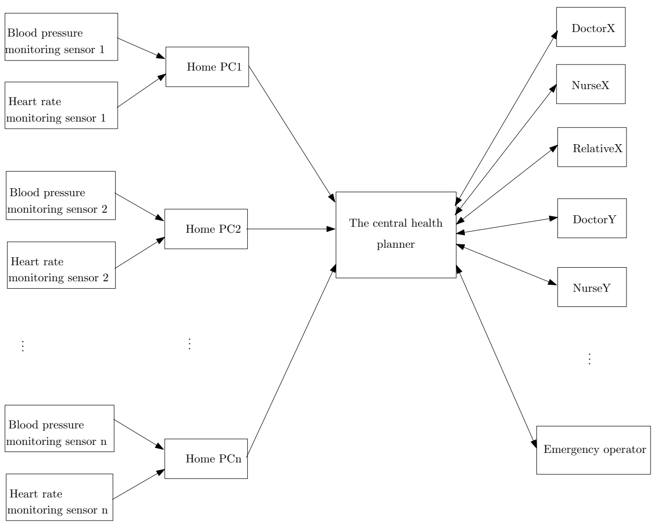
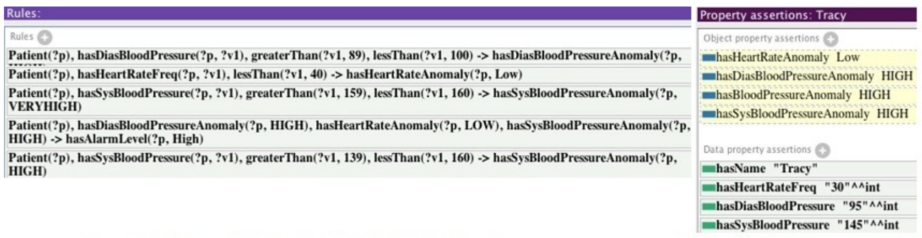

Blood pressure monitoring sensor 1
Heart rate monitoring sensor 1
Home PC1
Blood pressure monitoring sensor 2
Heart rate monitoring sensor 2
Home PC2
Blood pressure monitoring sensor n
Heart rate monitoring sensor n
Home PCn
The central health planner
DoctorX
NurseX
RelativeX
DoctorY
NurseY
Emergency operator

Fig. 4: Home health-care monitoring system

$Tell(b_i, pc_i, hasSysBloodPressure(?p, ?v2))$
$Tell(b_i, pc_i, hasDiasBloodPressure(?p, ?v1))$
$Tell(h_k, pc_i, hasHeartRateFreq(?p, ?v))$

Upon receiving the Blood pressure and Heart rate information from the body sensor agents, the agent pc_i derives an alarm level by firing a sequence of rules from its knowledge base, including the rules shown in Fig 5.


Rules:

Rules

Patient(?p), hasDiasBloodPressure(?p, ?v1), greaterThan(?v1, 89), lessThan(?v1, 100) -> hasDiasBloodPressureAnomaly(?p,

Patient(?p), hasHeartRateFreq(?p, ?v1), lessThan(?v1, 40) -> hasHeartRateAnomaly(?p, Low)

Patient(?p), hasSysBloodPressure(?p, ?v1), greaterThan(?v1, 159), lessThan(?v1, 160) -> hasSysBloodPressureAnomaly(?p, VERYHIGH)

Patient(?p), hasDiasBloodPressureAnomaly(?p, HIGH), hasHeartRateAnomaly(?p, LOW), hasSysBloodPressureAnomaly(?p, HIGH) -> hasAlarmLevel(?p, High)

Patient(?p), hasSysBloodPressure(?p, ?v1), greaterThan(?v1, 139), lessThan(?v1, 160) -> hasSysBloodPressureAnomaly(?p, HIGH)

Property assertions: Tracy

Object property assertions

hasHeartRateAnomaly	Low
hasDiasBloodPressureAnomaly	HIGH
hasBloodPressureAnomaly	HIGH
hasSysBloodPressureAnomaly	HIGH

Data property assertions

hasName	"Tracy"
hasHeartRateFreq	"30"^^int
hasDiasBloodPressure	"95"^^int
hasSysBloodPressure	"145"^^int

(a) Example SWRL rules for the Home Health Monitoring Ontology (b) An individual of the class Patient

Fig. 5: Example SWRL Rules and an individual of the patient

The level of alarms could be *Low*, *VeryLow*, *Medium* and *High* depending on the blood-pressure and heart-rate measurement values. The agent pc_i then sends the alarm level information to the agent p for the patient's health planning. In this system, the agents doctors, d_is, nurses, n_is, relatives of patients, r_is, and an emergency operator, e are modelled as abstract agents. These abstract agents can notify to the agent p about their availability status by sending messages which could be, e.g., *Available*,

$NotAvailable$, and $Busy$. The messages generated by these abstract agents are of the form:

$$\begin{array}{l} Tell(d_i, p, hasCareStatus(?c, ?status)) \\ Tell(n_i, p, hasCareStatus(?c, ?status)) \\ Tell(r_i, p, hasCareStatus(?c, ?status)) \end{array}$$

The agent p models alarm notification policies specifying whom should be alerted, how and when the notification is to be sent and if any acknowledgement is required. Alarm notification policy examples are given below:

Alarm Level	**Notification Policies**
VeryLow	message to relative, no ack
Low	message to doctor, no ack and message to relative, no ack
Medium	message to doctor or nurse, ack and message to relative, no ack
High	message to emergency operator, ack and message to relative, ack

The agent p alerts a contact person (doctor, nurse, or relative of a patient) based on their availability status and for certain cases the agent p may require an acknowledgement. The availability status of a doctor, nurse, or relative of a patient may change from $Available$ to $Busy$ or $NotAvailable$ when they are contacted by the agent p. In this case, the agent p waits for a fixed time interval and then based on the acknowledgement received it might contact other agents for a service. For instance, when a $Medium$ level alarm occurs, the agent p first alerts a doctor, d_i. If the received acknowledgement from the agent d_i within a fixed time interval is $Busy$ or $NotAvailable$, then the agent p alerts a nurse, n_i, if she also sends an $Busy$ or $NotAvailable$ message within a fixed time interval, then the agent p alerts an emergency operator. At the same time, the agent p alerts the relative of the patient, but an acknowledgement is not required.

The Blood pressure and Heart rate sensor agents in the system generate information about the measurement values at different times in the interval $[1, 5]$. For example, the agent b_i generates blood pressure information for a patient with patient's name $Tracy$ and systolic blood pressure 130mmHg using the following formula:

$$X^{\leq 5} \; B_{b_i} \; Tell(b_i, pc_i, hasSysBloodPressure('Tracy,' 130))$$

In this experiment, the priorities (from higher to the lower) among rules of the central Health Planner are assigned corresponding to the alarm levels $High$, $Medium$, Low, and $VeryLow$, respectively. The experimental results reported in Table 2, for the 1 patient scenario, the system generates $Medium$ alarms, for the 2 patients scenario, the system generates $Medium$ alarms for one patient and $High$ alarm for the other patient, and for the 3 patients scenario, the system generates $Medium$ alarms for two patients and $High$ alarm for the other patient. For ease of illustration, we modelled one doctor, one nurse, and one relative corresponding to each patient in the system. In the one patient scenario, two concrete agents are modelled using 16 and 36 rules respectively, three abstract agents are modelled using one LTL formula each, and other two abstract agents are modelled using two LTL formulas each. In the two patients scenario, three concrete agents are modelled using 16, 16, and 72 rules respectively, four abstract agents are modelled using one LTL formula each, and other seven abstract agents are modelled using two LTL formulas each. And in the three patients scenario, four concrete agents are modelled using 16, 16, 16, and 108 rules respectively, four abstract agents are modelled using one LTL formula each, nine abstract agents are modelled using two LTL formulas each, and one abstract agent is modelled using three LTL formulas. Maude encoding can be found online[5]. We verify the following properties of the system:

$Prop1.\ \ G(B_{pc_i}\ \ Tell(h_i, pc_i, hasHeartRateFreq('Tracy,' 30))$
$\wedge\ \ B_{pc_i}\ \ Tell(b_i, pc_i, hasSysBloodPressure('Tracy,' 130))$
$\wedge\ \ B_{pc_i}\ \ Tell(b_i, pc_i, hasDiasBloodPressure('Tracy,' 85))$
$\rightarrow X^n\ \ B_{pc_i}\ \ hasAlarmLevel('Tracy,' MEDIUM) \wedge msg_{pc_i}^{=m})$

the above property specifying the fact that healthPC classifies the alarm level as $'MEDIUM$ in n time steps while message counter value of the healthPC is m, when the values of blood pressure and heart rate are 130mmHg, 85mmHg and 30bps, respectively.

$Prop2.\ \ G(B_{pc_i}\ \ hasAlarmLevel('Tracy,' MEDIUM)$
$\rightarrow X^n\ \ B_p\ \ hasAlarmLevel('Tracy,' MEDIUM))$

the above property says that whenever patient's alarm level is classified (e.g., in this case it is $Medium$) the patient's home healthPC will interact and informs this to the planner p and the planner receives classified message in n time steps.

$Prop3.\ \ G(B_p\ \ hasAlarmLevel('Tracy,' MEDIUM)$
$\wedge\ \ B_p\ \ Tell(d_i, p, hasCareStatus('John,' Busy))$
$\wedge\ \ B_p\ \ Tell(n_i, p, hasCareStatus('Fiona,' Busy))$
$\rightarrow X^n\ \ B_e\ \ hasAlarmLevel('Tracy,' MEDIUM))$

the above property says that whenever patient's alarm level is $Medium$ and the agent p has received acknowledgements from the doctor and nurse as busy, then the agent p communicates with the emergency operator and the emergency operator e receives the message in n time steps.

The above properties are verified as true when the value of n and m are 7 and 3 in $Prop1$. The value of n is 2 in $Prop2$, and 3 in $Prop3$. The model checker spends 72 seconds for the 1 patient scenario, 165 seconds for the 2 patient scenario, and 842 seconds for the 3 patient scenario. However when we assign a value to n which is less than 7 in $Prop1$, less than 2 in $Prop2$, and less than 3 in $Prop3$ the properties are verified as false and the model checker returns counterexamples. Similarly, when we assign a value to m which is less than 3, $Prop1$ is verified as false. This also ensures the correctness of the encoding in that model checker does not return true for arbitrary values of n and m. Note that verification of true formulas take longer than verification of false formulas since a model checker will find a counterexample faster than it takes to explore the whole model. For example, when the model checker returns counterexamples it spends 0.04 seconds for the 1 patient scenario, 0.04 seconds for the 2 patient scenario, and 0.2 seconds for the 3 patient scenario. It should also be noted that the value of n depends on the experimental setup. For example, the value of n is 3 when verifying $Prop3$ for the 3 patient scenario and the planner has to contact emergency operator for one patient with $Medium$ alarm (because it has received acknowledgements from the doctor and

[5] https://www.dropbox.com/s/qjzv3ro9jqra4bs/MAS-MaudeEncoding.zip?dl=0

nurse as busy) and for another patient with $Medium$ alarm it receives positive acknowledgement from the doctor. However, the value of n is 4 when verifying $Prop3$ for the 3 patient scenario and the planner has to contact emergency operator for both the patients with $Medium$ alarm (because it has received acknowledgements from the doctor and nurse as busy for both the patients). The results are summarised in Table 2.

#Patients	#Concrete agents	# Abstract agents	$n\&m$	CPU Time (Sec.)
1	2	6	7&3 ($Prop\ 1.$) 2 ($Prop\ 2.$) 3 ($Prop\ 3.$)	72
2	3	11	7&3 ($Prop\ 1.$) 2 ($Prop\ 2.$) 3 ($Prop\ 3.$)	165
3	4	16	7&3 ($Prop\ 1.$) 2 ($Prop\ 2.$) 3 ($Prop\ 3.$)	842
3	4	16	7&3 ($Prop\ 1.$) 2 ($Prop\ 2.$) 4 ($Prop\ 3.$)	846

Table 2: Timing results for the health planner example

7 Case study 2: A synthetic distributed reasoning problem

To illustrate the scalability of our approach we reimplemented an example scenario introduced in [2] and preliminary results were reported in [4]. In this scenario, a system of communicating reasoners attempt to solve a distributed reasoning problem where the set of rules and facts that describes the 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:

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

In [2], variations on this synthetic 'binary tree' problem have been used, with A_is being the leaves and the goal formula being the root of the tree, as examples (see Figure 6). As we have already mentioned that the use of ontology-driven rules is to exploit an ontology and the SWRL rules to design a rule-based multi-agent system, which facilitates to capture and design critical elements of a real-world application. This synthetic distributed reasoning problem, which is not based on ontologies, is considered here because it can be easily parametrised by the number of leaf facts to increase or decrease the problem size.

Table 3: State space and CPU time produced by Mocha without using strategy abstraction

# Agents	# Leaves	Distribution	# Reach. states	# Reach. states (sym_search)	# Max. MDDs	# Max. MDDs (sym_search)	CPU time	CPU time (sym_search)
1	8	-	26	-	22	-	0.4	-
2	8	(4,4)	41943	336156	3108	3874	4	5
2	8	(odd,even)	55278	145511	3447	4636	7	8
1	16	-	784	-	173	-	1	-
2	16	(8,8)	8.6667e+08	2.34705e+10	131179	321423	469	3429
2	16	(odd,even)	7.52994e+08	3.64244e+09	189419	286196	613	2267
1	32	-	458330	-	1141	-	3	-
1	64	-	2.10066e+11	-	4655	-	251	-
1	128	-	4.41279e+22	-	38897	-	6472	-

Table 4: State space and CPU time produced by Maude using strategy abstraction

# Agents	# Leaves	# Steps	#States	#Msgs	CPU Time (in seconds)
1	8	7	24	-	0.1
1	16	15	48	-	0.2
1	32	31	96	-	0.5
1	64	63	192	-	0.7
1	128	127	384	-	1
1	512	511	1536	-	97
1	1024	1023	3072	-	903
1	2048	2047	6144	-	13252
2	128	115	790	2	7
3	128	103	1560	4	18

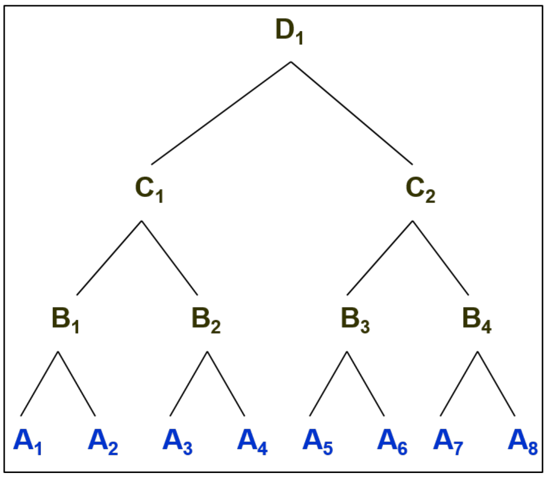


Fig. 6: Binary tree example

7.1 Analysis of experimental results

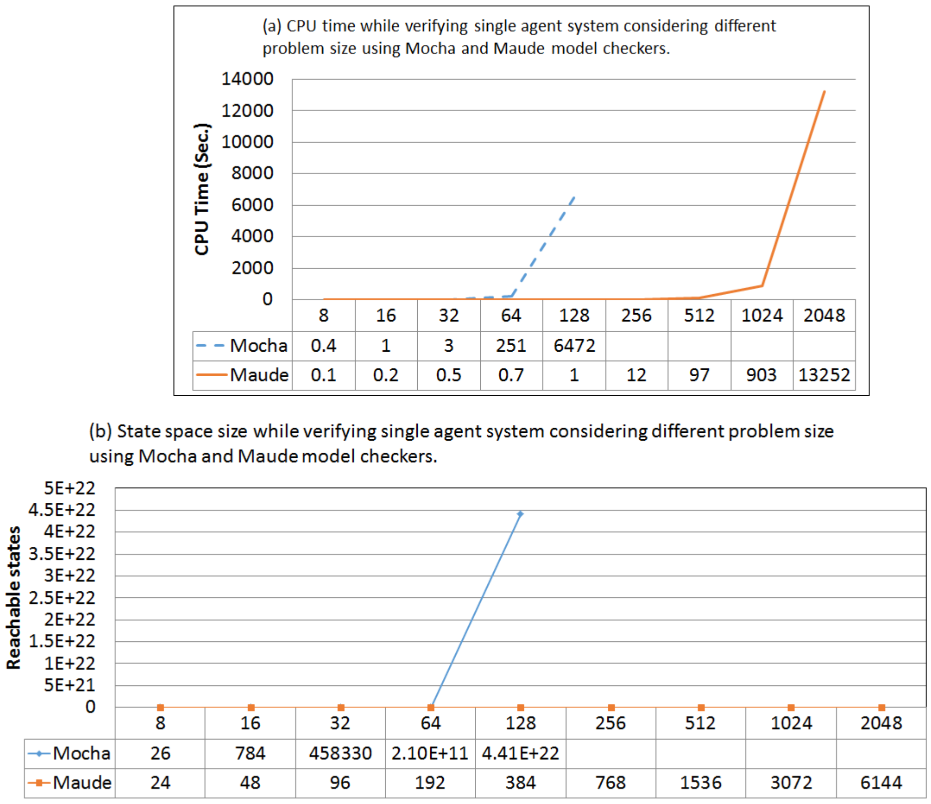
In [2], the results of various experiments of the binary tree problems using the Mocha model-checker [6] are reported. In the simplest case of a single agent, the largest problem that could be verified using Mocha had 128 leaf facts, as shown in Table 3. However, using our TOVRBA tool we are able to verify a system with 2048 leaf facts. This was modelled as a single concrete agent, with varying numbers of facts and rules. The experimental results are summarised in Table 4 ($\#\text{Agents} = 1$). In the case of multi-agent systems, the exchange of information between agents was modelled using **Comm** operation, which requires special communication rules. In [2], using Mocha we were able to verify a multi-agent system consisting of two agents with 16 leaf facts. An invariant property of the form $AG\neg(B_1\ \varphi \vee B_2\ \varphi)$ (where φ represents the the root node) was verified when the odd position node facts were assigned to one agent and the even position node facts were assigned to the other agent in the system. In our re-implementation, communication between agents is achieved using Ask and $Tell$ actions. The results presented in [2] and those for our TOVRBA 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 new approach. Maude encoding can be found online[6].

7.2 Discussion

The choice between symbolic and explicit-state model checking may depend on the system being verified. It has been argued that symbolic model checking performs better for synchronous systems, whereas explicit-state model checking is better for asynchronous systems [24][27, pp. 13]. However, Eisner and Peled [14] have reported that symbolic

[6] https://www.dropbox.com/s/qjzv3ro9jqra4bs/MAS-MaudeEncoding.zip?dl=0

model checking performs better even for asynchronous systems. In the experiments reported in this paper, we have used both symbolic and explicit-state model checking approaches. The results presented above using symbolic model checker Mocha, studied the worst case model indicated in Section 3.5. We have also experienced that Maude does not perform better than Mocha when we consider this worst case model. We could have encoded strategy based approach using Mocha to compare if it performs significantly better, 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. Figure 7 depicts experimental performance comparison using strategy and non-strategy based encodings, which indicates that much larger systems can be verified using our new approach.



	8	16	32	64	128	256	512	1024	2048
Mocha	0.4	1	3	251	6472				
Maude	0.1	0.2	0.5	0.7	1	12	97	903	13252

	8	16	32	64	128	256	512	1024	2048
Mocha	26	784	458330	2.10E+11	4.41E+22				
Maude	24	48	96	192	384	768	1536	3072	6144

Fig. 7: Experimental performance comparison using strategy and non-strategy based encodings

8 Related work

The idea of integrating ontologies and multi-agent systems has been realized in numerous research [25, 21]. Gateau [17] proposed a smart IoT middleware for comport management integrating ontologies and multi agent systems using JaCaMo [8] for Multi-Agent Programming. Ontology played a vital role to select a best action in multi-agent system whenever an event occurs. There has also been considerable work on rule-based agents and model checking multi-agent systems. In [35], Subercaze and Maret present a semantic agent model that allows SWRL programming of agents. A Java interpreter has been developed that communicates with the Knowledge-Base using the Protégé-OWL API. The prototype tool takes advantages of the Java-based domain modeling tool JADE that allows agent registration, service discovery and messages passing. The framework supports FIPA-ACL for agent communication. In [29], Mousavi et al. present an ontology-driven reasoning system based on BDI agent model [34]. In contrast to Jadex (that utilizes an XML format to represent agents' plans, beliefs and goals), in their framework, an ontology (in an OWL format) has been used to represent agents' believes, plans and events. The Java-based tool JADE was used to implement the agents, and the Protégé OWL was used to create the ontology. To illustrate the use of the framework, a simple Mobile Workforce Brokering Systems (a multi-agent system that automates the process of allocating tasks to Mobile Workforces) was modelled for simulation. In [5] 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 [5] uses Datalog facts represent information about references, and some simple rules ensure transitivity of the reference relation. While in the above a number of ontology-driven modeling and reasoning approaches [29], [35] have been developed for multi-agent systems, to our knowledge tools for automated formal verification for such systems are lacking. In [33], we have used the technique presented in this paper to model and verify resource-bounded context-aware systems, however, all the agents used in the case study were modelled as concrete agents. In the literature, there have been many other approaches to alleviate the state space explosion problem, including verification approaches based on compositional reasoning [7]. In compositional reasoning, a property φ to be verified is decomposed into sub-properties that describe the behaviour of small components of the system. The sub-properties are verified for the corresponding components. Then the system satisfies φ if all the sub-properties are satisfied locally and their conjunction implies φ. In contrast, our approach to verification using abstraction does not decompose φ into sub-properties. The property φ is verified in the whole system. However, we construct the system using a hierarchical composition in which the LTL properties can be previously verified properties of non-abstract versions of an abstract agent or set of abstract agents.

9 Conclusions and future work

In this paper, we proposed an approach to modelling, specifying and verifying response time guarantees of ontology-driven multi-agent rule-based systems. To design ontology, we use OWL 2 RL language because it is more expressive than the RDFS and it is suitable for the design and development of rule-based systems. An OWL 2 RL ontology can be translated into a set of Horn clause rules based on DLP [19]. Furthermore, we express more complex rule-based concepts using SWRL which allows us to write rules using OWL concepts. We show how the Maude LTL model checker can be used to verify desired system properties including response-time guarantees of the form: *if the system receives a query, then a response will be produced within n time steps*. We described results of experiments on a simple healthcare monitoring system, we also presented strategy-based efficient encoding of the rule-based multi-agent systems for LTL compared to our previously presented encoding for CTL. In future work, we plan to evaluate our approach on more real-life examples of Semantic Web and rule-based systems, and enhance our framework for designing and verifying situation-aware ambient intelligence systems.

ORCID Abdur Rakib `https://orcid.org/0000-0001-5430-450X`

Bibliography

[1] The Protégé ontology editor and knowledge-base framework (Version 4.1). http://protege.stanford.edu/ (July 2011)

[2] 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. LNCS, vol. 5348, pp. 1–14. Springer, Berlin/Heidelberg (2009)

[3] Alechina, N., Logan, B., Nga, N.H., Rakib, A.: Verifying time, memory and communication bounds in systems of reasoning agents. Synthese 169(2), 385–403 (April 2009)

[4] Alechina, N., Logan, B., Nguyen, H.N., Rakib, A.: Automated verification of resource requirements in multi-agent systems using abstraction. In: van der Meyden, R., Smaus, J.G. (eds.) 6th International Workshop, MoChArt 2010, Atlanta, GA, USA, July 11, 2010, Revised Selected and Invited Papers. LNAI, vol. 6572, pp. 69–84. Springer (2010)

[5] Alpuente, M., Feliú, M.A., Joubert, C., Villanueva, A.: Defining datalog in rewriting logic. In: Logic-Based Program Synthesis and Transformation, 19th International Symposium, LOPSTR 2009, Coimbra, Portugal, September 2009, Revised Selected Papers. LNCS, vol. 6037, pp. 188–204. Springer (2010)

[6] Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: Modularity in model checking. In: Proceedings of the 10th International Conference on Computer Aided Verification. pp. 521–525. CAV '98 (1998)

[7] Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional reasoning in model checking. In: Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures. LNCS, vol. 1536, pp. 81–102. Springer (1998)

[8] Boissier, O., Bordini, R.H., Hbner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming 78(6), 747 – 761 (2013), `http://www.sciencedirect.com/science/article/pii/S016764231100181X`

[9] Cao, F., Archer, N., Poehlman, S.: An agent-based knowledge management framework for electronic health record interoperability. Journal of Emerging Technologies in Web Intelligence 1(2), 119–128 (2009)

[10] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (Sep 1994)

[11] Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking (1999)

[12] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77 (1977)

[13] Culbert, C.: CLIPS reference manual. NASA (2007)

[14] Eisner, C., Peled, D.: Comparing symbolic and explicit model checking of a software system. In: Proceedings of the 9th International SPIN Workshop on Model Checking of Software. pp. 230–239. Springer-Verlag, Berlin, Heidelberg (2002), http://dl.acm.org/citation.cfm?id=645881.672229

[15] Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Proceedings of the 4th International Workshop on Rewriting Logic and its Applications(WRLA'02). Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187. Elsevier (2004)

[16] Friedman-Hill, E.J.: Jess, the rule engine for the java platform. Sandia national laboratories (2008)

[17] Gâteau, B.: A smart iot middleware for comfort management based on multi-agent-system. In: Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics. pp. 43:1–43:10. WIMS '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3227609.3227683

[18] Glimm, B., Horridge, M., Parsia, B., Patel-Schneider, P.F.: A syntax for rules in OWL 2. In: Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED 2009). vol. 529. CEUR (2009)

[19] Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proceedings of the 12th international conference on World Wide Web. pp. 48–57. ACM Press (2003)

[20] Gruber, T.: A translation approach to protable ontology specifications. Knowledge Acquisition 5, 199–220 (1993)

[21] Hendler, J.: Agents and the Semantic Web. IEEE Intelligent Systems 16, 30–37 (2001)

[22] Holzmann, G.: On-the-fly model checking. ACM Comput. Surv. 28(4) (Dec 1996)

[23] Horridge, M., Bechhofer, S.: The OWL API: A java API for working with OWL 2 Ontologies. In: 6th OWL Experienced and Directions Workshop (OWLED) (October 2009)

[24] Hu, A.J., York, G., Dill, D.L.: New techniques for efficient verification with implicitly conjoined bdds. In: Proceedings of the 31st Annual Design Automation Conference. pp. 276–282. DAC '94 (1994)

[25] Kravari, K., Kontopoulos, E., Bassiliades, N.: Emerald: A multi-agent system for knowledge-based reasoning interoperability in the semantic web. In: Konstantopoulos, S., Perantonis, S., Karkaletsis, V., Spyropoulos, C.D., Vouros, G. (eds.) Artificial Intelligence: Theories, Models and Applications. pp. 173–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[26] Lezcano, L., Sicilia, M., Rodríguez-Solano, C.: Integrating reasoning and clinical archetypes using OWL ontologies and SWRL rules. Journal of Biomedical Informatics 44, 343–353 (2011)

[27] Magazzeni, D.: Explicit Model Checking Techniques Applied to Control and Planning Problems. Ph.D. thesis, Dipartimento di Informatica, Università di L'Aquila, Università di L'Aquila (2009)

[28] Motik, B., Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language: Profiles, W3C Recommendation. http://www.w3.org/TR/owl2-profiles/ (October 2009)

[29] Mousavi, A., Nordin, M.J., Othman, Z.A.: An ontology driven, procedural reasoning system-like agent model,for multi-agent based mobile workforce brokering systems. Journal of Computer Science. 6, 557–565 (2010)
[30] Paganelli, F., Giuli, D.: An ontology-based context model for home health monitoring and alerting in chronic patient care networks. In: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops - Volume 02. pp. 838–845. AINAW '07 (2007)
[31] Rakib, A.: Verifying requirements for resource-bounded agents. Ph.D. thesis, The University of Nottingham. (2011)
[32] Rakib, A., Faruqui, R.U., MacCaull, W.: Verifying resource requirements for ontology-driven rule-based agents. In: Proceedings of the 7th International Conference on Foundations of Information and Knowledge Systems. FoIKS'12 (2012)
[33] Rakib, A., Ul Haque, H.M.: Modeling and verifying context-aware non-monotonic reasoning agents. In: Proceedings of the 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign. pp. 61–69. MEMOCODE '15 (2015)
[34] Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Proceedings of the First International Conference on Multi-agent Systems. pp. 312–319. The MIT Press (1995)
[35] Subercaze, J., Maret, P.: SAM - Semantic agent model for swrl rule-based agents. In: Proceedings of the International Conference on Agents and Artificial Intelligence. pp. 245–248. INSTICC Press (2010)
[36] Tzafestas, S.G.: Knowledge-Based System Diagnosis, Supervision, and Control. Plenum Publishing Co. (1988)