Model Checking Agent Programs by Using the Program Interpreter

Sung-Shik T.Q. Jongmans, Koen V. Hindriks, and M. Birna van Riemsdijk

Delft University of Technology

Abstract. Model checking agent programs is a challenge and it is still a question which approaches can suitably be applied to effectively model check such programs. We present a new approach to explicit-state, on-the-fly model checking for agent programs. In this approach we use the agent program interpreter for generating the state space. A model checker is built on top of this interpreter by implementing efficient transformations of temporal properties to Büchi automata and an efficient bookkeeping mechanism that maintains track of states that have been visited. The proposed approach is generic and can be applied to different agent programming frameworks. We evaluate this approach to model checking by comparing it empirically with an approach based on the Maude model checker, and one based on the Agent Infrastructure Layer (AIL) intermediate language in combination with JPF. It turns out that although our approach does not use state-space reduction techniques, it shows significantly improved performance over these approaches. To the best of our knowledge, no such comparisons of approaches to model checking agent programs have been done before.

1 Introduction

Various approaches have been used for model checking agent systems (see, e.g., [1–8]). In this paper, we focus on *explicit-state on-the-fly model checking for agent programming languages*. Current state-of-the-art approaches for model checking agent programs are based on the use of *existing model checkers*. In particular, in [8] and [1] agent programs written in Mable and AgentSpeak(F), respectively, are translated to Promela and verified with SPIN [9]. In the Agent Infrastructure Layer (AIL) project [10] a Java-based framework to which various APLs can be translated is used, in combination with the Java model checker Java Path Finder (JPF) [11]; the model checker is called AJPF. The definition of this translation needs to be specified only once for each AIL interpreted language. Moreover, in [7], an implementation of (a simplified version of) 3APL is presented in the Maude term rewriting language [12]. This enables model checking of 3APL programs with the Maude model checker (MMC) [13]. A possible advantage of such approaches is that built-in optimizations and state space reduction techniques of the existing model checker may be reused for the verification of agent programs.

In this paper, we propose a new approach in which a model checker is built from scratch on top of the interpreter of an agent programming language. Although any model checker needs to rely on an implementation of the semantics

of agent programs, our approach differs from others in the sense that it relies on an explicit but abstract interface to an agent program interpreter and evaluation of agent specific conditions are delegated to a standard interpreter for the language whereas temporal properties are handled by well-known techniques for LTL model checking (see also the architecture in Figure 1 below).[1] We have chosen the agent programming language GOAL [14] as our target language. One reason for choosing GOAL is that the operational semantics of GOAL has been implented in Maude, enabling the use of the MMC for GOAL, and a translation has been defined to AIL. This facilitates comparison between our approach and other approaches. To the best of our knowledge, no such comparisons of approaches to model checking agent programs have been done before. In this paper, we present an empirical evaluation of these approaches. It turns out that even though our approach does not use state-space reduction techniques, it shows significantly improved performance over these other approaches.

The contribution of this paper is thus twofold: we provide a new approach to model checking agent programs, and a comparison that provides insight into aspects that influence performance when using existing model checkers to model check agent programs.

The rest of the paper is organized as follows. Section 2 introduces some preliminaries. In Section 3, we introduce a new approach to model checking that is based on using the interpreter for an agent programming language itself during the verification of an agent program. We have implemented an interpreter-based model checker for the language GOAL, and briefly discuss the associated language for specifying properties. Section 4 presents a number of experiments and the results of a comparison between available approaches for model checking GOAL agents. Section 5 discusses our findings. The paper is concluded in Section 6.

2 Preliminaries

In this section, we briefly explain model checking, the GOAL language, and its property specification language.

Given a model of a system, model checking tests automatically whether this model satisfies a given property (see, for example, [15]). Properties are often specified in a temporal logic such as LTL [16], as we do in this work. A model of a program (consisting of all its possible computations) satisfies an LTL property φ if all computations satisfy the property. The model checking algorithm searches for a counterexample, i.e. a computation on which $\neg\varphi$ is true; if such a computation cannot be found, the model satisfies the property φ.

A GOAL agent decides which action to perform next based on its beliefs and goals. The beliefs (collectively called the *belief base*) typically represent the

[1] The model checker JPF used in an alternative model checking approach discussed in this paper is not built on top of the standard JVM in this sense but relies on a dedicated JVM developed for JPF. Most other model checking approaches require a translation to a specific language supported by the model checker (such as Maude). Our approach does not require such a translation.

current state of the agent's environment. The GOAL interpreter also offers the possibility to specify knowledge, which is general knowledge about the environment that is typically static. In the interpreter for GOAL, both knowledge base and belief base are Prolog programs. A decision to act will usually also depend on the goals of the agent. Goals of an agent are stored in a *goal base*. The goal base consists of conjunctions of Prolog atoms. Together, the beliefs and goals make up an agent's *mental state*. To make decisions, a GOAL agent uses so-called *action rules* which consist of a mental state condition used to inspect the agent's beliefs and goals, and an action that may be executed if the mental state condition holds. Actions include constructs for changing the agents' beliefs as well as goals; as we focus here on single agents communication primitives are not allowed. Action rules can be combined into so-called modules to provide additional structure to an agent program. To be precise, mental state conditions ϕ are built from *mental atoms* $\mathbf{B}\psi$ and $\mathbf{G}\psi$ as follows:

$$\begin{array}{l}\chi ::= \text{first-order atoms}\\ \psi ::= \chi \mid \neg\chi \mid \chi \wedge \chi\\ \phi ::= \mathbf{B}\psi \mid \mathbf{G}\psi \mid \neg\phi \mid \phi \wedge \phi\end{array}$$

Informally, $\mathbf{B}\psi$ is true if ψ follows from the belief base of the agent, and $\mathbf{G}\psi$ is true if ψ follows from the goal base. For example, we have $\mathbf{G}(p)$ in a mental state with goal base $\Gamma = \{p \wedge q\}$; due to space limitations, we cannot provide all the details here but refer the reader to [17] and remark that $\mathbf{G}$ refers to the primitive goal operator discussed in detail in [17]. A GOAL computation $t = m_0, a_0, m_1, a_1, \cdots$ is an infinite sequence of mental states m_i and actions a_i such that execution of a_i in m_i brings about m_{i+1}, and m_0 is the initial mental state. The meaning of a GOAL program is defined as the set of all its possible computations. More details about the program constructs the GOAL language supports and are supported by the model checker as well can be found in [17].

Although the model checking approach presented in this paper is able to handle programs with all features mentioned above, not all of these features could be used in the comparative experiments presented in this paper since various of these features are not supported by the translation of GOAL to AIL, nor by the implementation of the GOAL semantics in Maude. We recognize that in principle it is possible to extend the model checking approaches based on AIL and Maude to include these features. In this paper, however, we focus in particular on the performance of these approaches for the core subset of GOAL that is supported by all three of approaches discussed here. The basic assumption here is that if for a subset of the GOAL language we can already show significant performance differences, then it is unlikely that extensions to the full GOAL language will perform much better.

A simple GOAL program for solving a blocks world [18] tower building problem is given in Table 1. Blocks are represented using the predicate `block(X)`, the fact that a block `X` is stacked on another block `Y` is represented as `on(X,Y)`, and the fact that there is no block on top of a block `X` is represented as `clear(X)`. This program assumes a single tower of blocks labelled as `aa,ab,ac` etc. and stacked

in this order where aa is the bottom block. The agent moves all blocks to the table, i.e., when block ab is moved to the table, the tower has been unstacked. For this reason, the agent has the goal of having ab on the table.[2]

```
main: agent {
  beliefs {
      block(aa). block(ab). block(ac). block(ad). block(ae).
      on(aa,table). on(ab,aa). on(ac,ab). on(ad,ac). clear(ad).
      on(ae,table). clear(ae).
  }
  goals { on(ab,table). }
  program {
      if goal(on(ab,table)), bel(on(X,Y)), bel(clear(X))
      then moveXfromYtoTable(X,Y) .
  }
  action-spec {
      moveXfromYtoTable(X,Y) {
         pre { block(X) }
         post { not(on(X,Y)), on(X,table), clear(Y) }
      }
  }
}
```

Table 1: Simple GOAL program for the blocks world

The language used here to specify *properties of a* GOAL *program* is LTL, where the propositional atoms are mental atoms defined above.

3 An Interpreter-Based Model Checker

Figure 1 provides a graphical representation of an architecture for an interpreter-based model checker (IMC). The IMC architecture consists of two main components. The first component translates the negation of an LTL formula from the property language to a Büchi automaton, thus representing the property state space. (Recall that mental atoms are treated as propositional atoms in this component.) The formula translator that has been implemented is based on the LTL2AUT algorithm of [19]. The second component evaluates the property by means of a search of the product state space. The product state space is the product of the property state space and the program state space. The component implements the generalized nested depth-first search algorithm of [20], an *on-the-fly* exploration algorithm. This means that only parts of the product state space that are needed are actually generated. The program state space is obtained by means of the agent program interpreter which explains why we have labeled our approach interpreter-based.

We have created a model checker for the agent language GOAL by plugging in the interpreter of GOAL and using the mental atoms of the GOAL language as

[2] In GOAL, a conjunctive goal may be used to express that a set of blocks should be on the table, but the AIL translation does not allow the use of conjunctive goals.

atoms in the property language. This means that the mental state condition evaluation is delegated to the interpreter for the agent language itself and handled by the query evaluator that is part of the agent interpreter whereas temporal properties are represented by a Büchi automaton (see also Figure 1). In a similar way IMCs for other agent languages can be obtained by plugging in an interpreter for those languages and instantiating the atoms of the property language accordingly. For example, as Jason uses the construct ? for inspecting an agent's beliefs and ! for inspecting an agent's event base, these operators could be used instead of the **B** and **G** operators that are part of GOAL (or similar mappings as those proposed in [1] as long as the interpreter provides support to evaluate such conditions). So, even though the main IMC components have been built from scratch, this effort is not dedicated to a single agent programming language.

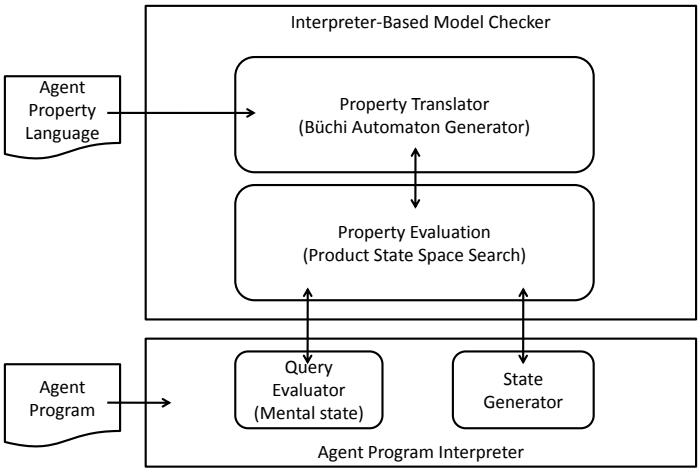


Fig. 1: Architecture of an Interpreter-Based Model Checker

To be able to support various agent languages a well-defined interface is needed from the IMC to the interpreter of a specific agent language. The interface for any IMC needs to provide support for two types of requests from the exploration component: requests for supplying successors states (given the current state and the agent program that is being checked), and requests for evaluating mental atoms in the mental state that is currently being examined. Both requests are handled by invoking existing methods of the interpreter. One advantage of the IMC approach is that the full support offered by the interpreter can be reused. Existing model checkers for APLs sometimes limit the expressiveness of the property language: the logic of [3], for example, only allows ground atomic

formulas inside mental literals. In contrast, because the query mechanism of the GOAL interpreter is used, *any* mental atoms that this interpreter can handle can be used as "atoms" in the property language, including conditions that contain free variables, the use of conjunctions and negations in mental atoms, and the use of knowledge rules. Another benefit of using an interface as described is that it only requires support for generating successor states and handling queries, and the approach abstracts from more specific differences between agent languages related to the precise set of built-in actions that is supported. The point is that although specific and concrete actions are needed in agent languages to compute successor states, the model checking approach presented here only needs to know the successor states but not the manner in which these were generated. The latter is delegated to the agent interpreter itself.

In addition, it is always possible to refine the IMC interpreter and add specific optimizations to make the interpreter more efficient. In our current implementation for GOAL we have implemented a translation of mental states to binary representations in order to more efficiently use memory resources. This representation of GOAL mental states increases the performance of the model checker. It is more costly to perform frequently used operations such as checking for equality of states and computing hash codes for states when the mental state representation that the interpreter manipulates is used instead of the binary representation. These operations need to be performed to check whether states have already been visited and to check for cycles in the search. We briefly explain the details of this representation for GOAL. Informally, every bit (having a unique index) in the binary representation of a mental state corresponds to a belief or a goal: 1-bits indicate that the corresponding belief or goal is part of the mental state, whereas 0-bits indicate the opposite. To translate binary representations of mental states back and forth, we need a bookkeeping mechanism that associates each indexed bit with a unique belief or goal. Since it is infeasible to compute beforehand which beliefs and goals an agent might have in a computation, assigning indices to beliefs and goals is done during model checking. This is achieved by *dynamically* identifying beliefs and goals that have not occurred so far, and assigning indices incrementally, starting from 0. This means that *at runtime* any beliefs or goals that have not occurred so far during execution are added to a list of beliefs and goals that have occurred so far and associated with an integer index (the position in the bit string); there thus is no need to know beforehand which beliefs or goals will occur. The benefit is that instead of using the explicit representation of beliefs and goals we can now use bit strings instead to represent mental states in the state space that needs to be searched. Using bit strings provides two benefits: it reduces memory (space) requirements but most importantly it reduces time needed to compare states (in order to check whether a state has already been visited).[3]

[3] Note that as knowledge (rules) are assumed to be static we do not need to represent these as part of the state space because they never change. Most, if not all, agent programming languages similar to GOAL assume rules to be static.

Due to space limitations we are not able to provide all the details here. In essence, the bit string representation of a mental state is a conversion of a mental state to a "canonical representation" of that state. The main point here is that if e.g. a belief or goal formula occurs in a mental state the presence of that formula can be checked at a unique index of the bit string. This means that whenever the state space exploration algorithm needs to verify that a mental state has already been visited it does not need to transform visited states into such a canonical representation first (e.g. by sorting the formulas in that state) in order to compare it with the new state to be checked. This involves a significant reduction of time needed to check the visited lists during state space exploration (which yields a time reduction approximately in the order of $N \times |m| \times log(|m|)$ where N denotes the length of the visited list and $|m|$ the average size of a mental state for each time that a visited check needs to be performed). To give an indication of the space reduction, we briefly discuss the more specific issue related to storing ground belief atoms only. Suppose that for explicit representation of such atoms we would use a string representation. A reasonable measure of size needed in that case would be the length of that representation, i.e. string length (in terms of bytes). For the sake of argument, let us assume that we can approximate space requirements by the average length of such strings and on average we would need L bytes to store a belief (this is an underestimate of what is really needed). Moreover, suppose we have N different belief atoms which may or may not occur in a belief base. This yields 2^N possible belief bases. The average number of beliefs in these belief bases is $\Sigma_0^N k \times \binom{N}{k}$ divided by 2^N, which equals $N/2$. Explicit state space representation (using only belief bases) would thus require $N \times 2^{N-1} \times L$ bytes, or $N \times 2^N \times 4 \times L$ bits. Instead the bit string representation would require $N \times L$ bytes to represent the list of belief atoms and $N \times 2^N$ bits to represent the state space. This is a conservative estimate of the space reduction, which shows that minimally space requirements are reduced with a factor $4 \times L$.

4 Experiments and Results

As we view our approach as only one alternative to others, it is important to evaluate our approach and compare it with existing alternatives. The key measure for comparison is performance as model checking of reasonably complex agent programs is only possible if performance is adequate. Our implementation of an IMC for GOAL enables a comparison between three approaches. The IMC for GOAL can be compared with AJPF (the AIL project model checker) and the MMC (based on Maude) that both provide a tool to model check GOAL agents. In order to gain a better understanding of the differences between these approaches we present a number of experiments and corresponding results.

4.1 Experimental Evaluation

Before introducing the experiments that we performed, we first discuss a number of issues related to evaluating the results of our experiments. One issue concerns

the input that is provided to the model checkers to obtain a fair comparison. A second issue concerns what it means to say that a model checker *scales well*. We use regression analysis to obtain resource consumption functions (time, space) that fit the data. A third issue concerns the interpretation of the data obtained from the experiments.

Semantic Equality of Agents The experiments are designed to enable a fair comparison as much as possible. The most important condition for a fair comparison is that semantically equivalent GOAL programs are provided as input to the three different model checkers. Unfortunately, as AJPF and MMC implement different subsets of GOAL, it turned out to be rather complex task to design experiments. For example, in AJPF, goals must be (negations of) single terms. As a result, the experimental agents used in the experiments are simple and are artificial in various respects. IMC is based on GOAL's interpreter and as such does not pose any restrictions, which illustrates another advantage of the interpreter-based approach introduced here; although there is no principled reason to assume other approaches cannot be extended to cover additional language features, the approach presented here provides an alternative that is able to support checking agent programs that use almost any feature supported by the agent interpreter (see also the discussion of the IMC interface above).

Apart from language support considerations there is a second reason for keeping the experiments simple. We first need to evaluate the performance of the model checkers for simple cases before providing more complex programs to check. This has also motivated us to use deterministic agents, i.e. agents that have only one possible computation, in our experiments. The relevant literature provides indications that it is already hard to model check simple programs due to performance reasons and these findings are confirmed by our experiments. We found that even simple non-deterministic agents were beyond the capabilities of MMC, and to a lesser extent this also is the case for AJPF. Another reason for our choices in this regard are that by only considering deterministic agents, it is easier to draw conclusions about resource consumption during the model checking of agents.

Scalability In order to be able to model check reasonably sized agent programs a model checker needs to be scalable. To avoid confusion, it is important to more precisely define when a model checker is said to scale well. Though improving scalability has been an important factor in research on software verification, to the best of our knowledge, no standards or metrics regarding scalability have been proposed for model checkers. In order to clarify this notion we introduce the following definition.

Definition 1 (Scalability and Unscalability). *A model checker is said to be* scalable *or* scale well *with regard to certain conditions, if the relation between those conditions and resource consumption of the model checker can be described by one of the following functions:*

$$\begin{array}{ll} \text{Logarithmic} & : y = b \cdot \log(x) + a \\ \text{Polynomial } (d < 1) & : y = b \cdot x^d \\ \text{Linear} & : y = b \cdot x + a \end{array}$$

A model checker is said to be not scalable *or* scale poor *with regard to certain conditions, if the relation between those conditions and resource consumption of the model checker can be described by one of the following functions:*

$$\begin{array}{ll} \text{Polynomial } (d > 1) & : y = b \cdot x^d \\ \text{Exponential} & : y = b \cdot r^x \end{array}$$

In these functions, x and y represents, respectively, the conditions and the resource consumption, a is called the intercept*, b is called the* coefficient*, d is called the* degree *of the polynomial, and r is called the* radix*..*

The intuition behind Definition 1 is that a model checker scales well if the relation between conditions and consumption is at most linear. The reason for regarding polynomial relations in degree $d > 1$ as not scalable is that x is, in general, very large for real-world model checking problems. In such cases, a quadratic relation versus a cubic relation can already make the difference between (in)tractability of a problem. Note that our definition differs from complexity theory, where all problems that can be solved in polynomial time are deemed tractable.

Regression analysis To determine the type of relation between experimental conditions and resource consumption, we will apply regression analysis to analyze the experimental data. That is, we will fit the functions given in Definition 1 to the measurements using least-squares regression, and assess how good the fits found are by comparing their R^2 values. The fitted function that yields the highest (i.e. closest to 1) R^2 is deemed the relation between conditions and consumption. The resource consumption of the model checkers is obtained by measuring two *dependent variables*: verification time and memory consumption.

The conditions under investigation, in statistics called *independent variables*, are the size of the belief base and the size of the state space. The size of the state space is defined as the total number of mental states that can be encountered on all computations of the GOAL agent. The size of a belief base is defined as the number of elements it contains (i.e. ground atoms). In the experiments reported on below, we simply used the model checkers to report the number of states visited.

To study the different effects of both independent variables, the experiments are organized as follows. In the first experiment, the size of the belief base is varied, while the size of the state space is kept constant. In the second experiment, the size of the belief base is kept constant, while the size of the state space is varied. Finally, in the third experiment, both the size of the belief base and state space are varied.

4.2 Experiments

We now present the three experiments that we have performed (together with the experimental results).

Experiment 1 (Size of Belief Base) In this first experiment, we investigate the scalability of the model checkers in the size of the belief base; the size of the state space is kept constant.

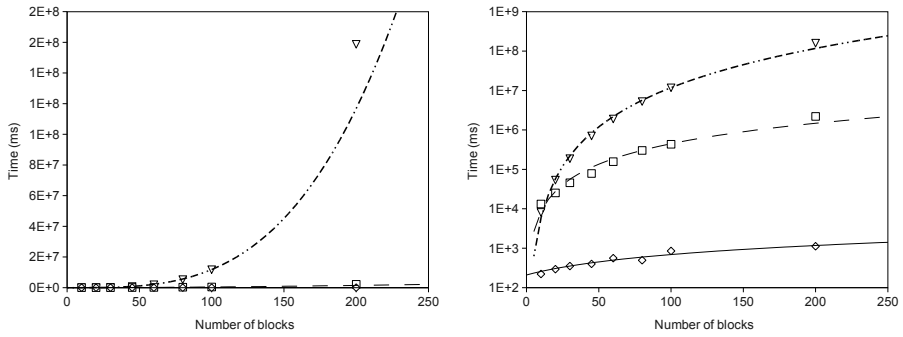
For this experiment, we have used variants of the agent program of Table 1 with $n \in \{10, 20, 30, 45, 60, 80, 100, 200\}$ blocks, rather than $n = 5$ as used in the agent program. Four of the n blocks are initially stacked on each other, whereas the remaining $n - 4$ blocks are on the table. For all n, the stacked blocks are aa, ab, ac, and ad: aa is on the table, ab is on aa, ac is on ab, and ad is on ac. In the target configuration, all the blocks are on the table. The property φ under investigation is whether the agent eventually brings about the target configuration. For all values of n, the program state space contains only four mental states. In contrast, the belief base grows as n increases: it becomes filled with *redundant* beliefs. That is, removing these beliefs would not affect the behaviour of the agent.

The verification times are displayed in Fig. 2a, and the calculated relations are given in the first column of Table 3a. These results suggest that IMC scales well to larger belief bases, in contrast to AJPF and MMC. Though the verification times of the latter two both grow polynomially in the size of the belief base, the degrees of the fitted functions (shown in Table 3a between brackets) show that the increase in verification time of MMC is more than cubic, whereas AJPF remains under quadratic. This difference can also be observed in Fig. 2a, and supports our decision to classify polynomial relations in degree $d > 1$ as not scalable. For $n = 200$, the absolute difference in performance is the largest: IMC took 1 second, AJPF took 40 minutes, while MMC took 44 hours.

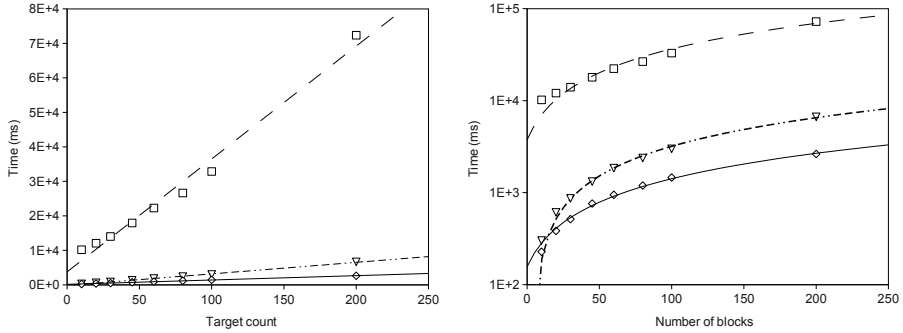
The memory consumption is displayed in Fig. 3a, and the calculated relations are given in the first column of Table 3b. One might notice that although the degrees of the fitted polynomial functions for IMC and AJPF are the same, the memory requirements of the former are much lower. The reason for this is that the coefficient b (see Definition 1) for IMC is roughly 2, whereas for AJPF it is roughly 22. According to Definition 1, all three model checkers scale well to larger belief bases with respect to memory consumption. Nevertheless, AJPF is substantially more memory demanding than IMC and MMC.

Experiment 2 (Size of State Space) In this second experiment, we investigate the scalability of the model checkers in the size of the state space; the size of the belief base is kept constant. To reduce the effect of the size of the belief base on the experimental results (the previous experiment showed that such an effect is definitely present), it should be as small as possible. As these requirements (small belief base, growing state space) are not easily satisfied by a Blocks World scenario, we use the following setting.

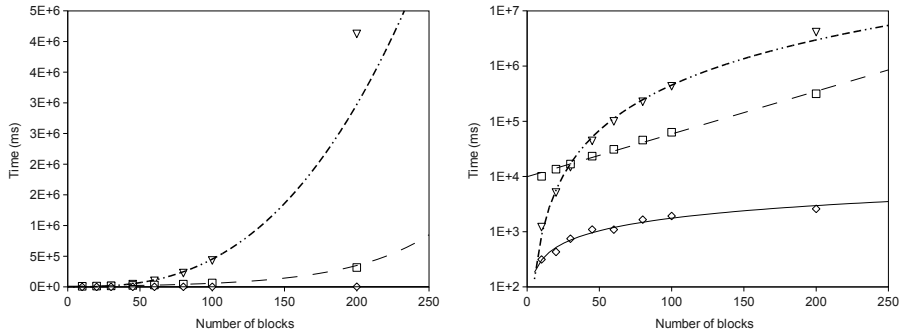
In this experiment, the agent is a simple counter: it starts at 0, and counts until infinity. There are various ways to implement this behaviour, and we chose an implementation in which: the belief base is used as little as possible (for reasons outlined above), and goals are issued very frequently. The reason for issuing many goals is that goal creation is a relatively slow operation in the cur-



(a) Experiment 1.



(b) Experiment 2.



(c) Experiment 3.

Fig. 2: Verification times of IMC (continuous line, measurements as ⋄), AJPF (dashed line, measurements as □), and MMC (dashed-dotted line, measurements as ▽) in Experiments 1, 2, and 3. Plotted lines are best-fit regression lines. Left figures have a linear scale on the y-axis, whereas the scale on the y-axis of right figures is logarithmic.

rent GOAL interpreter. Thus, by issuing many goals, not only do we challenge the interpreter, but the interpreter-based model checker as well. Note that because the agent can count until infinity, the state space of the agent is not finite. To ensure that the model checking procedure is decidable, the property φ must be verifiable in a finite number of interpretation cycles. One such property is that the agent eventually believes that its current number is some natural number n. Hence, the size of the state space is controlled by the value of n in φ; in this experiment, we chose $n \in \{10, 20, 30, 45, 60, 80, 100, 200\}$.

The verification times are displayed in Fig. 2b, and the calculated relations are given in the second column of Table 3a. Though we expected MMC to be the slowest of the three (based on its performance in Experiment 1), AJPF is in fact ten times slower: the slope of the linear function fitted on the AJPF measurements is roughly 330, whereas the slope of MMC's linear fit is only 33. For $n = 200$, the difference is the largest: IMC took 3 seconds, AJPF took 72 seconds, and MMC took 7 seconds.

The memory consumption is displayed in Fig. 3b, and the calculated relations are given in the second column of Table 3b. Similar to Experiment 1, all reported relations are scalable according to Definition 1, but again, AJPF demands substantially more memory. Also, the relations for IMC and MMC imply that MMC's memory consumption grows faster in the size of the state space than that of IMC. Hence, it is to be expected that for some $n > 200$, IMC will consume less memory than MMC. This is not obvious from Fig. 3b.

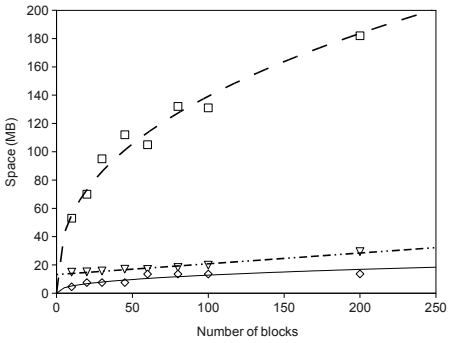
Experiment 3 (Size of State Space and Belief Base) In the third experiment, we investigate the scalability of the model checkers with respect to both the size of the belief base and the state space.

To satisfy the desired experimental conditions (growing belief base and state-space), we adapt the agent of Experiment 2 in such a way that it remembers all counted numbers. As a consequence, the size of the belief base will increase linearly in the size of the state space. These beliefs are, like the superfluous blocks in Experiment 1, redundant. The property under investigation is the same as in Experiment 2 for the same values of n such that all three model checkers terminate eventually.

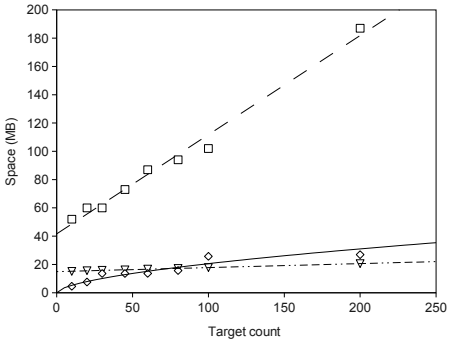
The verification times are displayed in Fig. 2c, and the calculated relations are given in the third column of Table 3a. IMC is again the fastest of the three model checkers, and still shows to scale well. In contrast, scalability of AJPF and MMC drops from linear to exponential and polynomial in degree 2.7, respectively. For $n = 200$, the absolute difference in performance is the largest: IMC took 3 seconds, AJPF took 5 minutes, while MMC took over an hours.

The memory consumption is displayed in Fig. 3c, and the calculated relations are given in the third column of Table 3b. The memory demands are similar to those in Experiments 2: AJPF performs, though depending linearly on the experimental conditions (thus, scalable in terms of Definition 1), the least well of the three model checkers, whereas IMC and MMC perform roughly equal. It is interesting to see that the intersection point of IMC and MMC for some $n > 200$,

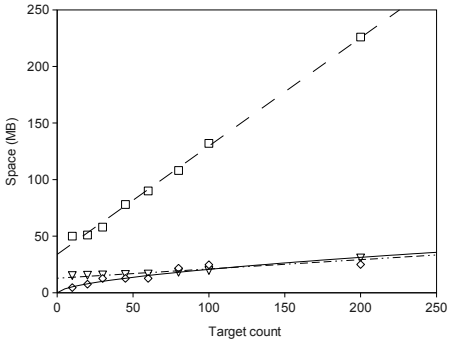
mentioned when treating the memory consumption of the model checkers in Experiment 2, is almost within the range of the values of n in Experiment 3.



(a) Experiment 1.



(b) Experiment 2.



(c) Experiment 3.

Fig. 3: Memory consumption of IMC (continuous line, measurements as $\diamond$), AJPF (dashed line, measurements as $\square$), and MMC (dashed-dotted line, measurements as $\triangledown$) in Experiments 1, 2, and 3. Plotted lines are best-fit regression lines.

Summary Table 3a summarizes the results with respect to verification time. IMC is the only model checker that scaled well in all three experiments. The other two model checkers clearly have problems when the size of the belief base increases. Table 3b summarizes the results with respect to memory consumption. With only polynomial relations in degree $d < 1$, IMC has performed best with regard to memory consumption. Though the measurement for AJPF and MMC imply scalability as well, Fig. 3 shows that there still are clear differences between IMC and MMC on the one hand, and AJPF on the other.

Table 2: Relations between resource consumption and experimental conditions.

	Experiment 1		Experiment 2		Experiment 3	
	Relation	R^2	Relation	R^2	Relation	R^2
IMC	**Linear**	0.9329	**Linear**	0.9974	**Poly** (0.75)	0.9722
AJPF	Poly (1.7)	0.9706	**Linear**	0.9829	Exponential	0.9925
MMC	Poly (3.3)	0.9958	**Linear**	0.9985	Poly (2.7)	0.9936

(a) Verification time. Boldface shows scalability.

	Experiment 1		Experiment 2		Experiment 3	
	Relation	R^2	Relation	R^2	Relation	R^2
IMC	**Poly** (0.40)	0.8266	**Poly** (0.59)	0.9109	**Poly** (0.60)	0.9113
AJPF	**Poly** (0.40)	0.9663	**Linear**	0.9860	**Linear**	0.9966
MMC	**Linear**	0.9664	**Linear**	0.9982	**Linear**	0.9491

(b) Memory consumption. Boldface shows scalability.

Comparison with Normal Execution In order to distinguish between overhead caused by the model checker and possible inefficiency caused by the underlying execution mechanism, we have measured resource consumption during execution (rather than verification) of the agents by the three underlying platforms as well. This can be done because the programs are deterministic, and therefore the trace that is model checked coincides with the trace generated by executing the programs.

In case of the program of Experiment 1, we observed that both AJPF and MMC introduce substantial overhead with respect to run-time: the largest difference between execution and verification times, measured for $n = 200$, amounted to roughly 37 minutes for AJPF, and over 41 hours for MMC. In contrast, the difference measured for IMC was only 50 milliseconds. With respect to memory consumption, less extreme differences were measured: for $n = 200$, IMC, AJPF and MMC required, respectively, 10 MB, 125 MB, and 2 MB less than when model checking.

In case of the program of Experiment 2, we observed that the overhead of MMC during verification is a lot smaller than in Experiment 1: the largest difference, measured for $n = 200$, is only 2 seconds (compared to 41 hours in Experiment 1). For AJPF, the difference is again large: roughly 70 minutes. For IMC, the largest difference is still negligible: approximately half a second. With respect to memory consumption, larger differences were measured than for Experiment 1: for $n = 200$, executing the agent took IMC, AJPF, and MMC, respectively, 24MB, 172MB, and 6MB less than when model checking the agent.

In case of the program of Experiment 3, we observed that the overhead of AJPF with respect to run-time is, as in the previous experiments, substantial. For MMC, in contrast to Experiment 2, the overhead is significant as well. For $n = 200$, the difference between execution and verification times for IMC, AJPF,

and MMC are 1 second, over 5 minutes, and 24 minutes, respectively. With respect to memory consumption, again larger differences were measured: for $n = 200$, executing the agent took IMC, AJPF, and MMC, respectively, 22 MB, 203 MB, and 16 MB less than when model checking the agent.

4.3 Wumpus Scenario

In order to illustrate that IMC is able to model check larger agent programs which give rise to much larger state spaces, we present results about model checking an agent for the well-known Wumpus World scenario [21]. The primary motivation for presenting this domain is to show IMC is able to handle more realistic scenarios. We are not able to present results for this domain for AJPF or MMC. The main reason is that even for small instances of this domain the state space already is significantly bigger than those used in the previous experiments, and by extrapolating the results obtained above it is unlikely to obtain results for either AJPF or Maude within reasonable time.

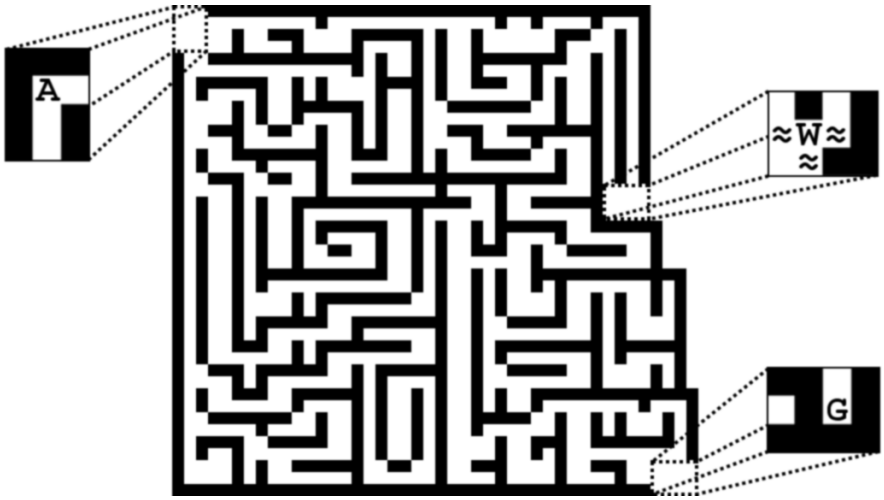


Fig. 4: Wumpus World

In the Wumpus World, a single agent is located in a cave that contains pits which need to be avoided, walls that prevent movement, and a beast called the Wumpus that as pits will kill the agent if it steps onto it. The cave is a grid world and locations can be identified by x and y coordinates. Figure 4 illustrates the environment; A denotes the agent, G represents gold, and W represents the Wumpus (the grids around the Wumpus are marked so the agent can smell it is next to the Wumpus; similarly pits are marked by 'breezes'). The goal of the agent is to locate gold that resides somewhere (at a position initially unknown to the agent, the environment is partially observable) and after getting the gold

leaving the cave. We can model check this environment as it is a single agent and deterministic environment which ensures we can always determine a unique set of successor states for each action that is performed by the agent; for more details see [21].

The first agent that we verified moves through the cave completely non-deterministically. It bumps into a wall if a wall blocks its way, and returns to a previously visited position if it encounters a stench (indicating that the Wumpus is close by). We checked a property that specifies that, given this Wumpus-avoidance-policy of the agent, it can never be at the same position as the Wumpus: $\Box\neg$**B**`position(33,-11)`. As the position of the Wumpus is fixed (at `(33,-11)`), satisfaction of this property means that at least the agent will never die. The model checker reports after exploring 57355 states in 6:30 minutes and using 97 MB of memory that the property is true. Another property that we would like the agent to satisfy is that it eventually obtains the gold: $\Diamond$**B**`has(gold)`. Unfortunately, the model checker reports a (non-minimal) counterexample after exploring 89 states in 2 seconds and using 42 MB of memory: at some point, the agent enters a loop of turning left, moving forward three steps, turning left twice, moving forward three steps again, and turning left again. We can, however, establish that there exists at least one computation on which the agent does obtain the gold by verifying the property $\Box\neg$**B**`has(gold)`. The model checker reports a counterexample in 2:19 minutes and uses 36 MB of memory; this counterexample corresponds to the computation on which the agent obtains the gold.

The second agent that we verified is a lot smarter than the first: it maintains a mental map of the cave by remembering the positions it visited, including a "trail of breadcrumbs" to find its way outside efficiently, and systematically explores unknown grounds. This implementation removes the non-determinism from the agent (making the state space smaller), but because a lot of information must be stored, the belief base is much larger: it grows linearly as the agent explores the cave, which has over 1600 different positions. The Wumpus-avoidance-policy of this agent is the same as that of the first, and the model checker re-confirms its effectiveness after exploration of 8225 states in 55 seconds using 37 MB of memory. As a result the agent satisfies $\Diamond$**B**`has(gold)`: verification required exploration of 3877 states, took 48 seconds and required 30 MB of memory.

5 Discussion

The experimental results clearly show that IMC outperforms the other two model checkers. Also, the results show that especially MMC is unable to deal with the simple toy examples that were under investigation, particularly with regard to verification time; to a lesser extent, this is also true for AJPF. With regard to AJPF, we believe that for a large part, the overhead of JPF is responsible for the slow verification (as well as for higher memory consumption), because executing (rather than verifying) the agent is substantially faster: the agent of Experiment 3 easily counts to 200 within a few seconds, whereas verifying with

AJPF whether this agent actually can count to 200 takes over 5 minutes. Similar differences were observed in all three experiments. With regard to MMC, the slow verification is partly ascribed to the rate at which the Maude GOAL interpreter can generate the state space: the agent of Experiment 3 already takes 45 minutes to count to 200 (when executed). Note, however, that the overhead of Maude's built-in model checker can be substantial as well: verifying whether the agent can actually count to 200 takes 24 more minutes.

Earlier, we mentioned that model checking non-deterministic agents is infeasible with AJPF and MMC: we carried out an additional experiment with IMC featuring a non-deterministic agent to illustrate this. Consider a Blocks World agent as in Experiment 1 with an initial belief base containing 200 blocks divided over 2 towers of 100 blocks each, and a property specifying that the target configuration (all blocks on the table) is reached. This non-deterministic agent has a state space of size 10,000. Nevertheless, verification with IMC takes only 2150 seconds, i.e. 35 minutes. In contrast, AJPF and MMC required already more time (40 minutes and 44 hours, respectively) to complete verification for $n = 200$ in Experiment 1: a comparable setting with only 4 states instead of 10,000. Given that the size of the belief base is constant, and assuming that AJPF and MMC scale linearly in the size of the state space (as suggested by the results of Experiment 2), it would take AJPF 100,000 minutes, i.e. 70 days, and MMC 110,000 hours, i.e. 12.5 *years*, to terminate.

Another observation concerns the size of the belief base: it turns out this has a large impact on the verification times associated with AJPF and MMC. It follows that the performance of these model checkers is not only dependent on the size of the state space, but also on the way that beliefs are dealt with. We speculate that two important aspects need to be optimized to increase performance. First, the mechanism for querying the belief base should be implemented as efficiently as possible. For MMC, this seems a problem as normal execution of the agents already took a long time in Experiments 1 and 3 (in Experiment 2, belief base queries were only performed on a belief base with at most two beliefs). Second, the model checker should not introduce overhead on this querying, which seems to be the case for AJPF.

6 Conclusion

We have distinguished three different approaches to model checking agent programs: two approaches that reuse existing model checkers in quite different ways, including for example [1] and the AIL approach [10], and the interpreter-based approach that we introduced in this paper. An architecture for this new approach has been introduced and we discussed how the approach can be applied to various agent programming languages. An implementation for the GOAL agent language has also been provided. One advantage of the interpreter-based approach is that in practice it supports a more expressive property language than that supported by currently existing alternatives.

In order to compare these approaches we performed various experiments to gain insight into the performance of these approaches. As far as we know, such performance comparisons have not been made before and providing such results is one of the contributions of this paper. Our main finding is that the interpreter-based approach outperforms the other two approaches. In addition, the computed relations with respect to resource consumption show that model checking non-deterministic agents (or, in general, any agent with a state space that is orders of magnitudes larger than, for example, the simple Blocks World examples we used) is currently beyond the capabilities of AJPF and MMC. In contrast, IMC handles those state spaces with relative ease.

We plan on further developing the interpreter-based GOAL model checker, in particular by extending it with state space reduction techniques. As we have full control over de code of the model checker, we expect that implementing such techniques is, from a programming point of view, less complex than when such language-specific optimizations would need be incorporated in an existing model checker.

References

1. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking agentspeak. In: Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems, ACM (2003) 409–416
2. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifiable multi-agent programs. Programming Multi-Agent Systems **3067** (2004) 72–89
3. Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated verificiation of multi-agent programs. In: 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society (2008) 69–78
4. Dennis, L.A., Fisher, M.: Programming verifiable heterogeneous agent systems. Programming Multi-Agent Systems **5442** (2009) 40–55
5. Kacprzak, M., Nabialek, W., Niewiadomski, A., Penczek, W., Pólrola, A., Szreter, M., Wozna, B., Zbrzezny, A.: Verics 2007 - a model checker for knowledge and real-time. Fundamenta Informaticae **85** (2008) 313–328
6. Lomuscio, A., Raimondi, F.: Mcmas: a model checker for multi-agent systems. Tools and Algorithms for the Construction and Analysis of Systems **3920** (2006) 450–454
7. van Riemsdijk, M.B., de Boer, F.S., Dastani, M., Meyer, J.J.C.: Prototyping 3apl in the maude term rewriting language. Computational Logic in Multi-Agent Systems **4371** (2007) 95–114
8. Wooldridge, M., Fisher, M., Huget, M.P., Parsons, S.: Model checking multi-agent systems with mable. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems, ACM (2002) 952–959
9. Holzmann, G.J.: The SPIN model checker. Addison-Wesley (2003)
10. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A common semantic basis for bdi languages. Programming Multi-Agent Systems **4908** (2008) 124–139
11. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10** (2004) 203–232

12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science **285** (2002) 187–243
13. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude ltl model checker. In: Proceedings of the 4th International Workshop on Rewriting Logic and its Applications, Elsevier Science (2002) 162–187
14. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.C.: Agent programming with declarative goals. Intelligent Agents VII **1986** (2001) 248–257
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. The MIT Press (1999)
16. Emerson, E.: Temporal and modal logic. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics. Elsevier, Amsterdam (1990) 996–1072
17. Hindriks, K.V.: Programming rational agents in goal. Multi-Agent Programming (2009) 119–157
18. Slaney, J., Thiébaux, S.: Blocks world revisited. Artificial Intelligence **125** (2001) 119–153
19. Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear temporal logic. Computer Aided Verification **1633** (1999) 681–692
20. Tauriainen, H.: Nested emptiness search for generalized buchi automata. In: Proceedings of the 4th International Conference on Application of Concurrency to System Design. (2004) 165–174
21. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice-Hall, Englewood Cliffs, NJ (2003)