Distributed BDD-based BMC for the Verification of Multi-Agent Systems

Andrew V. Jones   Alessio Lomuscio

Department of Computing
Imperial College London, UK

{andrew.jones,a.lomuscio}@imperial.ac.uk

ABSTRACT

We present a method of distributed model checking of multi-agent systems specified by a branching-time temporal-epistemic logic. We introduce a serial algorithm, central to the distributed approach, for combining binary decision diagrams with bounded model checking. The algorithm is based on a notion of "seed states" to allow for state-space partitioning. Exploring individual partitions displays benefits arising from the verification of partial state-spaces. When verifying both an industrial model and a scalable benchmark scenario the serial bounded technique was found to be effective. Results for the distributed technique demonstrate that it out-performs the sequential approach for falsifiable formulae. Experimental data indicates that increasing the number of hosts improves verification efficiency.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Verification

Keywords

Verification of MAS, Distributed Model Checking, Bounded Model Checking, Binary Decision Diagrams

1. INTRODUCTION

Multi-agent systems (MAS) are distributed systems in which agents, based on sophisticated communicating processes, exhibit autonomous behaviour. Recent research in the verification of multi-agent systems has highlighted their use in ensuring correct functionality in scenarios ranging from e-Business and web services to security protocols.

Symbolic model checking [19] is a powerful technique for the verification of reactive systems. Traditionally, such approaches use binary decision diagrams (BDDs) to represent the model. However, even BDDs cannot overcome the state space explosion problem.

There has been significant research into applying symbolic model checking to the realm of multi-agent systems [16, 23]. Unfortunately, due to the highly autonomous and decoupled nature of multi-agent systems, the resulting BDD representing the reachable states can still exceed the feasible limits of what can be verified.

Bounded model checking (BMC) [5] attempts to alleviate this difficulty by considering only a truncated model up to a specific depth. Traditionally, bounded model checking looks at the possible falsification of a universally quantified formula, via a translation of the model and the negation of the property, to the Boolean satisfiability problem (SAT). BMC for MAS, via a translation to SAT, has been investigated in [22]; an experimental implementation is presented in [15].

In this paper we present a method of bounded model checking for the epistemic logic CTLK [22]. The approach uses BDDs to represent the reachable state space [11, 19], rather than a translation of the problem to SAT (Section 3). We demonstrate how to extend this technique to support agent verification in a distributed environment (Section 3.2). We implement these techniques into an existing model checker for multi-agent systems, `MCMAS` (Section 4). For a constructive evaluation of our methods, when compared with the existing implementation (Section 5.1), we use an industrial use-case (Section 5.2) and provide a scalable benchmark scenario (Section 5.3).

Although the serial and distributed techniques place limitations upon the grammar, both approaches are sound and complete.

Related Work. In 2001 Copty *et al.* [9] investigated the possibility of using BDDs rather than SAT when performing bounded model checking. They adapted Intel's conventional BDD-based model checker `Forecast` to perform bounded model checking. Their method attempts to check an invariant property through a reachability check of a target error set, representing the complement of the given property. Much like SAT-based BMC, their algorithm only checks up to a given depth and terminates with an incomplete result if the intersection is not satisfied.

The ideas presented by Copty *et al.* were further extended by Cabodi *et al.* [6]. They discuss the idea of performing not only forward bounded model checking, but also working backwards from the error set using the state pre-image function. Additionally, their algorithm allows for completeness by taking the fixed point of states into consideration.

The model checker `NuSMV` [7] attempts to provide a method for "early falsification", again of only invariant properties.

Cite as: Distributed BDD-based BMC for the Verification of Multi-Agent Systems, Andrew V. Jones and Alessio Lomuscio, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

The approach taken by this verifier is to check, at each successive depth, if the reachable states are a subset of the states in which the property holds.

Both these approaches place a severe limitation upon the various properties that can be checked. It should be immediately obvious that the full grammar of expressible properties in a temporal-epistemic logic cannot be expressed by simply providing the model checker with a single state and then attempting a reachability check. Restriction to simple invariant properties exhibits the same issues.

In addition to the work discussed above, Iyer *et al.* [12] propose a grid-based method for bounded model checking by finding various "candidate deep reachable states", which can be used as seeds from which to run parallel SAT solvers. They argue that, when starting SAT-based BMC at a deeper initial state, it is possible to explore further into the model, as well as to locate errors that may not be detected by existing methods. Their method uses partitioned BDDs and under-approximation to construct a partitioned state space, such that generating the seeds remains tractable, but this is achieved at the expense of completeness [13]. Seed states are written as conjunctive normal form clauses at regular intervals and are subsequently used to start multiple parallel SAT-based BMC instances.

2. PRELIMINARIES

2.1 Temporal-Epistemic Interpreted Systems

The Interpreted Systems formalism [10] is an established semantics for multi-agent systems.

Assume that $\mathcal{A} = \{1, \ldots, n\}$ represents the set of n agents in the system, as well as the environment e modelling where all of the agents "live".

Each agent $i \in \mathcal{A}$ has a set of local states $\mathcal{L}_i$ and a repertoire of actions Act_i that it can perform. The protocol function $\mathcal{P}_i : \mathcal{L}_i \rightarrow 2^{\mathrm{Act}_i}$ governs which actions can be performed by an agent in a given local state. Assume that the environment is modelled in a similar way (i.e., by associating the sets $\mathcal{L}_e$, Act_e and $\mathcal{P}_e$ with the same meaning).

The set of joint actions $\mathrm{Act} \subseteq \mathrm{Act}_1 \times \cdots \times \mathrm{Act}_n \times \mathrm{Act}_e$ represents actions that are performed "jointly" (i.e., synchronously - all agents and the environment perform their respective action at the same time). An evolution function $\tau_i : \mathcal{L}_i \times \mathrm{Act} \rightarrow \mathcal{L}_i$, $i \in \mathcal{A}$, specifies how agents evolve from one state to another, depending on the joint action performed by the system as a whole (τ_e, respectively, for the environment).

The set of all possible global states $\mathrm{G} \subseteq \mathcal{L}_1 \times \cdots \times \mathcal{L}_n \times \mathcal{L}_e$ is a subset of the Cartesian product of the local states for all agents in the system. A global state $(l_1, \ldots, l_n, l_e) \in \mathrm{G}$ represents an instantaneous configuration of the system.

The transition relation $\mathrm{T} \subseteq \mathrm{G} \times \mathrm{Act} \times \mathrm{G}$ defines the temporal evolution of the system. Two global states g and g', $(g, g') \in \mathrm{T}$ iff there exists a joint action $a_1, \ldots, a_n$, such that for all $i \in \mathcal{A}$, $a_i \in \mathcal{P}_i(l_i(g))$ and $\tau_i(l_i(g), a_1, \ldots, a_n) = l_i(g')$. We assume seriality of this relation (i.e., every global state has at least one successor).

Given an initial state $\iota \in \mathrm{G}$ the protocols for each agent and the global transition function generate a (potentially infinite) structure representing all of the possible computations of the system. A path $\pi = (\iota, g_1, \ldots)$ is an infinite sequence of global states such that $\forall_{k \geq 0}\ (g_k, g_{k+1}) \in \mathrm{T}$ (for finite paths, k is bounded accordingly). $\pi(k)$ is the k^{th} global state of the path π, whilst $\Pi(g)$ is the set of all paths starting at the given state ($g \in \mathrm{G}$).

The function $l_i : \mathrm{G} \rightarrow \mathcal{L}_i$ is a projection of an individual agent's local state from a given global state. The epistemic accessibility relation $\sim_i \subseteq \mathrm{G} \times \mathrm{G}$ represents that two global states are indistinguishable for that agent. Formally, $(g, g') \in \sim_i$ iff $l_i(g) = l_i(g')$.

A model of an interpreted system $\mathcal{M}_{\mathcal{IS}}$ is a tuple $(\mathrm{G}, \iota, \mathrm{T}, \sim_1, \ldots, \sim_n, \mathcal{V})$ where G is the set of reachable states accessible from ι via T, and $\mathcal{V}$ is a mapping of global states to the propositional variables that hold at that state $\mathcal{V} : \mathrm{G} \rightarrow 2^{\mathcal{PV}}$.

The models of interpreted systems can be used to reason about a branching-time temporal-epistemic logic. The logic CTLK [22] is an enrichment of Computational Tree Logic (CTL) [11], with modalities for knowledge. The language CTLK is defined in terms of a countable set of propositional variables $\mathcal{PV} = \{p, q, \ldots\}$, $i \in \mathcal{A}$, and using the following syntax:

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid EX\varphi \mid EG\varphi \mid E\left[\varphi U \psi\right] \mid \overline{K}_i\varphi$$

The epistemic modality $\overline{K}_i\varphi$ is read as "agent i considers it possible that φ". We define $EF\varphi$ as $E\left[\mathrm{true} U \varphi\right]$. The duals are as follows: $AX\varphi \overset{\mathrm{def}}{=} \neg EX \neg\varphi$, $AF\varphi \overset{\mathrm{def}}{=} \neg EG \neg\varphi$ and $AG\varphi \overset{\mathrm{def}}{=} \neg EF \neg\varphi$. $A\left[\varphi U \psi\right]$ has the obvious semantics. The dual of the epistemic modality for "possibility" is "knowledge"; $K_i\varphi$ is defined as $\neg\overline{K}_i\neg\varphi$, and is read as "agent i knows φ". The temporal operators are read as usual [11].

We can define two fragments of CTLK: an *existential* fragment ECTLK and a *universal* fragment ACTLK [22]. ECTLK places a restriction upon the syntax such that negation can only be applied to elements of $\mathcal{PV}$ (i.e., in the BNF above $\neg\varphi$ is replaced with $\neg p$). The universal fragment contains the negations of all the formulae in ECTLK (i.e., ACTLK $= \{\neg\varphi \mid \varphi \in \mathrm{ECTLK}\}$). It can be seen that ACTLK contains formulae of the kind $AX\varphi$, $AF\varphi$, $AG\varphi$ $A\left[\varphi U \psi\right]$ and $K_i\varphi$.

Given a model of an interpreted system $\mathcal{M}_{\mathcal{IS}}$, a global state g and two CTLK formulae φ and ψ, satisfaction of φ and ψ at a global state g in a model $\mathcal{M}_{\mathcal{IS}}$, written $\mathcal{M}_{\mathcal{IS}}, g \vDash \varphi$ (or, for brevity, $g \vDash \varphi$), is defined as follows:

$$\begin{array}{lll}
g \vDash p & \text{iff} & p \in \mathcal{V}(g) \\
g \vDash \neg\varphi & \text{iff} & g \nvDash \varphi \\
g \vDash \varphi \vee \psi & \text{iff} & (g \vDash \varphi) \text{ or } (g \vDash \psi) \\
g \vDash \varphi \wedge \psi & \text{iff} & (g \vDash \varphi) \text{ and } (g \vDash \psi) \\
g \vDash EX\varphi & \text{iff} & (\exists\pi = \Pi(g))\ \pi(1) \vDash \varphi \\
g \vDash EG\varphi & \text{iff} & (\exists\pi = \Pi(g))\ \forall_{m \geq 0}\ [\pi(m) \vDash \varphi] \\
g \vDash E\left[\varphi U \psi\right] & \text{iff} & (\exists\pi = \Pi(g))\ \exists_{m \geq 0} \\
 & & [\pi(m) \vDash \psi \text{ and } \forall_{0 \leq j < m}\ \pi(j) \vDash \varphi] \\
g \vDash \overline{K}_i\varphi & \text{iff} & \exists g' \in G, g \sim_i g' \text{ and } g' \vDash \varphi
\end{array}$$

A CTLK formula φ is valid in a model $\mathcal{M}_{\mathcal{IS}} = (\mathrm{G}, \iota, \mathrm{T}, \sim_1, \ldots, \sim_n, \mathcal{V})$ iff $\mathcal{M}_{\mathcal{IS}}, \iota \vDash \varphi$, i.e., φ is true in the initial state of a model.

2.2 MCMAS – A Model Checker for Multi-Agent Systems

MCMAS [16] is a symbolic model checker for multi-agent systems. It implements ordered binary decision diagram-based algorithms for the verification of temporal-epistemic formulae on interpreted systems. It is written in C++ and uses the CUDD [24] library that provides BDD data structures, asynchronous variable reordering and garbage collection.

MCMAS implements the standard fixed point methods [11] in the algorithm $\textsc{Sat}_{\textsc{ctlk}}$ [23]. This calculates the satisfiability set of states $[\![\varphi]\!]$, i.e., the set of reachable states in which φ holds. MCMAS's algorithm $\textsc{Sat}_{\textsc{K}}$ symbolically calculates the satisfiability set for formulae of the kind $K_i\varphi$.

To check the validity of a formula φ MCMAS constructs (and checks the satisfiability of) the formula $\iota \rightarrow \varphi$, where ι represents a propositional atom that holds only in the initial states of the model. If $[\![\iota \rightarrow \varphi]\!]$ is equivalent to the set of reachable states, then the formula φ holds at the initial states of the model. Hence it is valid.

3. BDD-BASED BMC

The method we propose, as outlined in Algorithm 1, extends the algorithms presented in [23]. It attempts to perform falsification of an ACTLK property (line 4) at every depth of incremental state space generation (line 7).

Algorithm 1 BDD-BMC(ψ : ACTLK Formula, $\mathcal{I}$: Initial State, Trans : Transition Relation) : Boolean

```
 1: $\varphi \leftarrow \neg\psi$ {$\varphi$ : ECTLK Formula}
 2: Reach $\leftarrow \mathcal{I}$ {Reach : BDD}
 3: while True do
 4:   if $[\![\iota \rightarrow \varphi]\!]$ = Reach then
 5:     return False
        {Counterexample to ACTLK formula found}
 6:   end if
 7:   Reach $\leftarrow$ Reach $\vee$ (Reach $\wedge$ Trans)
 8:   if Reach Unchanged then
 9:     break {Fixed point reached}
10:   end if
11: end while
12: return $[\![\iota \rightarrow \psi]\!]$ = Reach
```

The technique put forward differs from Cabodi and Copty ([6] and [9], respectively) in one major respect. While [6, 9] merely performed a set intersection between either the reach set or the frontier set of states with a target error state, the algorithm proposed performs a full satisfiability check on the current reachable state space at the current BMC depth.

The algorithm presented has two "exit" points: lines 5 and 12. Line 5 encodes that, upon discovery of a counterexample to the ACTLK formula, the construction of the reachable states is stopped. The second exit point (line 12) is only accessible via a `break` in the loop (line 9), the condition detecting a fixed point in the construction of the state space.

3.1 Symbolic Epistemic Possibility

Although one requirement of Kripke models is that the transition relation should be serial, the current fixed point methods for CTL (see [11]) are correct even when using non-serial transition relations. Currently MCMAS only supports the "box" modality $K_i\varphi$. To calculate the satisfiability set ($[\![\varphi]\!]$) for the entire grammar of ECTLK formulae we require an extension of MCMAS to provide a symbolic method for $\overline{K}_i\varphi$. One approach could be to use the dual of $\overline{K}_i$ and the existing $\textsc{Sat}_{\textsc{K}}$ method [23], although this would be inefficient. We extend the original algorithm $\textsc{Sat}_{\textsc{ctlk}}$ with a method for symbolic calculation of $\overline{K}_i\varphi$, shown in Algorithm 2. We refer to this extension as $\textsc{Sat}_{\textsc{ctl}\overline{\textsc{k}}}$.

Algorithm 2 $\textsc{Sat}_{\overline{\textsc{K}}}$($\varphi$: Formula, i : Agent) : *set of* State

```
X $\leftarrow \textsc{Sat}_{\textsc{ctl}\overline{\textsc{k}}}(\varphi)$
Y $\leftarrow \mathrm{pre}_{\textsc{k}}(\mathrm{X}, i)$
return Y
```

The function $\mathrm{pre}_{\textsc{k}}$ returns the set of all states that are epistemically accessible for the given agent i; i.e., the set of all global states in which the local state of agent i is invariant. It can be seen that the algorithm is correct, given the obvious parallels to $\textsc{Sat}_{\textsc{EX}}$ (see [11]).

3.2 Distributed BDD-based BMC

We take inspiration from the work of Iyer *et al.* [12, 13] to develop an extension to MCMAS and a Java framework to support distributed bounded model checking. Similarly to the original BDD-based BMC approaches, we focus only on invariant properties – those that have AG as the top most connective in the parse tree. We also restrict the grammar such that epistemic sub-formulae can only be defined with non-temporal formulae (i.e., conjunctions, disjunctions and negations of atoms) beneath the modality K_i. The algorithm works in three main stages:

1. *Fixed-Depth BDD-based BMC.* Initially our original algorithm is used to perform bounded model checking up to a fixed depth.

2. *Seed State Generation.* If the ACTLK property is not falsifiable up to the fixed depth, then every state ("seed") on the fringe of the current set of reachable states is saved to the file using the DDDMP package [3].

   The satisfiability set for the non-temporal sub-formulae beneath each K_i is also saved (e.g., $[\![p]\!]$ for $K_i\,p$).

3. *Distributed Parallel BDD-based BMC.* Finally, concurrent MCMAS BMC instances are started on different hosts for each seed state (iteratively, where the number of seeds is greater than the number of hosts).

To allow for completeness of epistemic sub-formulae, if the fixed point is reached for an individual seed state, that host will then load $[\![p]\!]$ (see above) from disk and perform one final satisfiability check with this enlarged reach set.

3.2.1 Correctness of Distributed BMC

When we restrict the grammar of expressible properties as stated above (i.e., invariant formulae with non-temporal epistemic sub-formulae), the approach of partial state space evaluation used in the method of distributed bounded model checking is both sound and complete. It is sound because only valid conclusions can be derived, i.e., a counterexample found from a seed state is a valid counterexample in the full model. It is complete because all valid conclusions can be derived, i.e., when a counterexample cannot be found from a seed state, one does not exist in the full model.

Proposition 1. *Seeded bounded model checking is* sound *for $AG\varphi$.*

Proof. Through the construction of the seed states every seed state is reachable from the initial state in the model. Finding a counterexample from this seed state means that there exists a path from that state to another in which φ does not hold (i.e., $EF\neg\varphi$ holds in the seed state). As such there exists a path in the full model that starts at the initial state, passes through the seed state and reaches the error state. Therefore, from the semantics of CTLK, we also have $EF\neg\varphi$ in the initial state (the union of the sub-model from the seed and the path to that seed is a valid sub-model). For epistemic sub-formulae this means that the required epistemically-related states are found from the current seed, or in $[\![p]\!]$ if the fixed point has been reached. □

Proposition 2. *Seeded bounded model checking is* complete *for $AG\varphi$.*

Proof. If the truncated model up to the depth at which the seed states were generated could not satisfy $EF\neg\varphi$, and neither could any of the partial state spaces starting from each individual seed, this means that there does not exist a reachable state in which φ does not hold. As such, from the semantics of CTLK, we do not have a path in any part of the model that satisfies $EF\neg\varphi$. The union of all seeded sub-models and the initial fixed-depth model is the model itself. Therefore, $AG\varphi$ is valid in the model. Due to the persistence of $[\![p]\!]$ we know that all possibly epistemically-related states have been included when performing $\text{Sat}_{\overline{K}}$. □

4. IMPLEMENTATION

This section includes a description of the modifications made to MCMAS release 0.9.7.1 to support both bounded and distributed model checking. A release of this branch is available from [2].

4.1 BDD-Based BMC

In the implementation we restrict MCMAS to allow only for the verification of properties specified in either ACTLK or ECTLK. After parsing the model we construct a list of tuples (φ, ψ), where φ is the originally specified formula. For ACTLK $\psi = \neg\varphi$; for ECTLK $\psi = \varphi$.

We implemented the method `check_formulae_BMC` that is called at every depth of state space exploration. The method loops over the aforementioned list and removes the tuples for which ψ can be satisfied (i.e., for an ACTLK φ a counterexample can be found). State space generation only continues to a deeper depth if there still exist formulae to be verified (i.e., the list is not empty).

4.2 Symbolic Epistemic Possibility

To create a BDD representing the local state for a given agent we construct an expression consisting of the representation of local states of every other agent through the conjunction of their variables. The CUDD function `ExistAbstract` [1] is then used to quantify existentially this expression from the BDD representing $[\![\varphi]\!]$. The resulting BDD represents *only* the local state for the agent whose knowledge we wish to check. This can be used to identify the global states for which the local state is invariant from the global states in which φ holds.

4.3 Distributed BDD-based BMC

Using Java we implemented a framework (included in our experimental release [2]) for distributing the verification process. It has two types of instance: "Master" and "Slave". "Master" represents the initial instance that performs fixed-depth BMC, after which this instance becomes a "coordination" node. "Slave" instances are those that perform *full* BMC (until a counterexample is found or a fixed point is reached) from a given seed state.

When a slave instance returns $false$ the master instance terminates the verification on all other slave instances; alternatively, when a slave returns *true* it is allocated another seed. This process continues until all seeds have been exhausted ($AG\varphi$ is true) or a counterexample is found ($AG\varphi$ is false).

We extended MCMAS's syntax checking phase to enforce the restriction upon the grammar, as specified in Section 3.2.

5. EVALUATION

This section presents an evaluation of the BDD-based BMC implementation, followed by the distributed extension. The following section describes the hardware and software configuration that we used for verification. Initially (§ 5.2) we benchmark BMC against the original, unmodified, implementation in a use-case scenario taken from industry – the "software development" model. In Section 5.3 we present a further scalable scenario—the "faulty train gate controller" model—that allows for a more critical evaluation of the system. To conclude, in Section 5.4 we evaluate the distributed implementation compared to the serial BMC approach.

5.1 Hardware and Software

The machines used for the following evaluation were dual core PCs, each with 4 GiB of memory and an Intel Core 2 Duo clocked at 3.00 GHz, with a 4096 KiB cache. The machines ran 32-bit Ubuntu Linux 8.04.2, with a vanilla 2.6.24-19-generic kernel and glibc 2.7. The modified MCMAS build was linked against release 2.4.1 of the CUDD library and version 2.0.3 of the DDDMP package. The seed states were saved to the networked file system mounted on a server with an XFS file system, using a 4 KiB block size (in a RAID configuration). The network between the machines and the file server ran at 1 Gb/s. All experiments were performed four times, with the results presented here being the average across all four runs.

5.1.1 BDD Library Settings

In a similar way to the work presented in [20, 21], and to provide fair benchmarks, we turned off CUDD's asynchronous variable reordering and garbage collection. This was done to evaluate the approach, rather than benchmarking a specific implementation.

For instance, if CUDD were to perform asynchronous reordering more frequently during state space generation, this could cause a sub-optimal variable reordering to be selected. Such an ordering could be preferential for the current reach set, but might be an adverse ordering for the reach set generated in the next state space generation iteration. CUDD only allows a certain time per-attempt to find an optimal reordering and, if one is not found, does not change the ordering.

We wanted to avoid assessing the benefits that such an implementation gains from the optimisations (such as automatic variable reordering) arising from its use of an auxiliary library.

5.2 The Software Development Model

In [17, 18] Lomuscio *et al.* present a model based on the composition of services revolving around a governing contract. Their model contains seven agents: "a principal software provider (PSP), a software provider (SP), a software client

Table 1: ACTLK “Software Development” Properties

φ_{SD1}	$AG\,(PSP\_Green \rightarrow K_{PSP}\,(AF\,(PSP\_End)))$
φ_{SD2}	$K_{PSP}\,(A\,[All\_Green\,U\,Software\_Delivered])$

Table 2: ECTLK “Software Development” Properties

φ_{SD3}	$E\,[Client\_Green\,U\,Software\_Delivered]$
φ_{SD4}	$E\,[(PSP\_Green \wedge ServiceProvider\_Green)\ U\ (Software\_Integrated \wedge EF\,(Software\_Tested))]$
φ_{SD5}	$E\,[Client\_Green\,U\ (E\,[(Client\_Green \wedge Software\_Delivered)\ U\ (\neg Client\_Green)])]$

(C), an insurance company (I), a testing agency (T), a hardware supplier (H) and a technical expert (E).”

Their idea is as follows: The client (C) desires for a piece of software to be developed and, subsequently, for the technical expert (E) to deploy this software upon hardware supplied by H. Two parties provide the software: the principle (PSP) and non-principle (SP) software providers. The PSP performs software integration of its software with SP's when a deliverable is made, which it then sends to the testing agency (T) for testing. If the software passes testing, it is given to the insurance company (I) for the provision of software insurance. The software is finally handed over to the E, who deploys it on the H.

The description of the model as found in [17, 18] defines a protocol allowing for changes to software and negotiation between contractual parties. It should be noted that, if any of the above parties errantly deviate from this protocol (e.g., C requires software changes that either PSP or SP do not agree with, or the software fails in testing too many times), the contract is violated.

Various properties for the software development model can be seen in Tables 1 and 2; the former shows properties in ACTLK, the latter in ECTLK. When an agent is said to be in a “green” state, this represents that agent being in compliance with the contract. For instance, PSP_Green represents that the PSP is compliant, whilst All_Green represents that all of the parties are compliant (i.e., the conjunction of i_Green holds for all parties). The other atoms take their intuitive meanings (e.g., $Software\_Integrated$ means that the software has been successfully integrated and $Software\_Delivered$ holds only in states for which the software has been delivered).

Specifications

ACTLK properties from Table 1:

φ_{SD1} – Whenever the PSP is in compliance (i.e., it is in a green state) it knows that the contract will eventually be successfully fulfilled.

φ_{SD2} – PSP knows that all other parties are in compliance until the software is delivered.

ECTLK properties from Table 2:

φ_{SD3} – There exists a path in which the Client (C) is always in compliance until he receives the software.

φ_{SD4} – In paths where the PSP and SP are always in compliance the software can eventually be integrated and tested.

Table 3: Software Development Results

	% BMC of Conventional			
Formula	Memory	Time	States	Depth
φ_{SD1}	1.604	0.221	0.007	0.000
φ_{SD2}	1.665	0.215	0.007	0.000
φ_{SD3}	32.305	10.198	16.725	79.167
φ_{SD4}	12.246	4.152	3.129	59.722
φ_{SD5}	2.655	0.389	0.420	8.333

φ_{SD5} – There exists a trace through the model where the client is always in compliance until the software is delivered and before the client enters a violation.

Within this model all of the ACTLK properties are falsifiable, whilst all of the ECTLK ones are satisfiable. The reader is referred to [17, 18] for explanations of the valuations of these formulae.

5.2.1 Evaluation

A comparative evaluation of our BMC implementation and the conventional model checking technique can be seen in Table 3. Each cell represents the value BMC required, as a percentage of exploring the whole model.

From the ACTLK results it can be seen that BMC appears to explore a negligible portion of the model. This arises from the fact that BMC can falsify the formulae in the initial state and so does not have to explore the model any further. This is caused by the transition relation being reflexive on the initial state and $\neg\varphi$ holding in this state. As such $EG\neg\varphi$ holds in the initial state, whilst the formula being verified (e.g., $A\,[\psi U \varphi]$) requires $AF\varphi$ to hold. This is a known advantage of the BMC technique.

The ECTLK results highlight the fact that, the deeper into the model we explore, the greater the memory required. For instance, φ_{SD3} shows that we require 32% of the memory used by the conventional technique to represent 80% of the model (68% of the memory is used to represent the final 20% of the model).

It can be seen that BDD-based BMC can out-perform standard BDD verification. Although these results demonstrate that the method can be effective, the formulae being checked are those for which we would expect BMC to perform best (i.e., falsifiable ACTLK/satisfiable ECTLK formulae).

5.3 A Scalable Multi-Agent System

Whilst the evaluation of a real benchmark from industry is of interest, to evaluate the technique further we analyse a scalable model. The following model has satisfiable ACTLK properties and it is also possible to vary the depth of the model to be explored before the "early termination" condition of the BMC algorithm is reached.

We adapted the benchmark scenario of the "Train Gate Controller", as presented by Alur *et al.* [4] and modified by Wooldridge *et al.* [25] and Kacprzak *et al.* [14]. The model involves two circular train tracks, each with a train travelling in a different direction. At a particular part of the track the trains must pass through a tunnel that can only accommodate a single train. At the point at which the tracks merge there is a controller, which controls signals for entry to the tunnel. If a train sees a green light it knows it is safe to enter the tunnel. The local states for each agent, as per the interpreted systems semantics, can be defined as follows: $\mathcal{L}_{\textsc{Train}_1} = \mathcal{L}_{\textsc{Train}_2} = \{Away, Wait, Tunnel\}$ and $\mathcal{L}_{\textsc{Controller}} = \{Red, Green\}$. The protocol is omitted.

Our interpretation of the model is unique in that we adapt the trains to display faults under certain circumstances. Using `MCMAS`'s bounded integer type we extend each train with a "service counter" (with a maximum value) and a "breaking depth". The service counter is incremented each time a train performs an action; when this counter reaches the maximum value the train is serviced, resetting the counter to zero. Once the service counter exceeds the breaking depth, the trains *may* perform a non-deterministic break action whilst in the tunnel. The controller is also changed such that it waits two evolutions between entering the *Red* state and changing back to the *Green* state.

We defined three types of trains: for the first, once the break action has been performed the train remains in the tunnel perpetually; when the second type performs a break action the train is delayed from leaving for that turn (but can perform an infinite number of break actions); the final (working) type has all of the break actions removed.

The specifications in Table 4 are false in a model containing type-1 or type-2 trains but are true in one with type-3 (working) trains. The formulae have been given with respect to a model containing two trains but they can easily be adapted to refer to more trains in a larger model.

We define the propositional atom "$\textsc{Train}_i$__IN__TUNNEL" ($i \in 1, 2$) to hold iff train i is currently inside the tunnel (i.e., the local state for that train is *Tunnel*).

$\varphi_{\textsc{tgc}1}$ states that $\textsc{Train}_1$ is infinitely often *not* in the tunnel. $\varphi_{\textsc{tgc}2}$ expresses a mutual exclusion property in the model: two trains never occupy the tunnel at the same time. $\varphi_{\textsc{tgc}3}$ represents that, whenever a train is in the tunnel, it knows that the other train is not. $\varphi_{\textsc{tgc}4}$ represents that trains are aware that they have exclusive access to the tunnel. Finally, $\varphi_{\textsc{tgc}5}$ indicates that trains are aware that there is a gap of at least one transition between the first train leaving and the next entering.

The formulae in Table 4 can be parameterised in a similar way to those presented by Kacprzak *et al.* [14]. For example, for a system composed of N trains, a parameterised $\varphi_{\textsc{tgc}3}$ can be seen in Figure 1. This takes the intuitive meaning: "when a train is in the tunnel it knows that no other train in the whole system is in the tunnel".

5.3.1 Evaluation

The memory requirement of bounded model checking with respect to the original technique is shown in Figure 2, whilst Figure 3 shows the same requirement for time. Both figures illustrate results for various complexities of formulae, whilst the breaking threshold of the train is directly proportional to the BMC depth (i.e., iterations of Algorithm 1) required to falsify the formulae. This depth affects the number of reachable states and the size of the BDD representing them.

Figure 2: Memory required to verify various formulae.

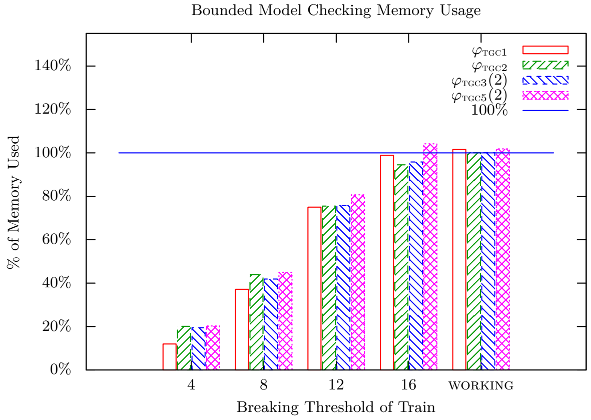
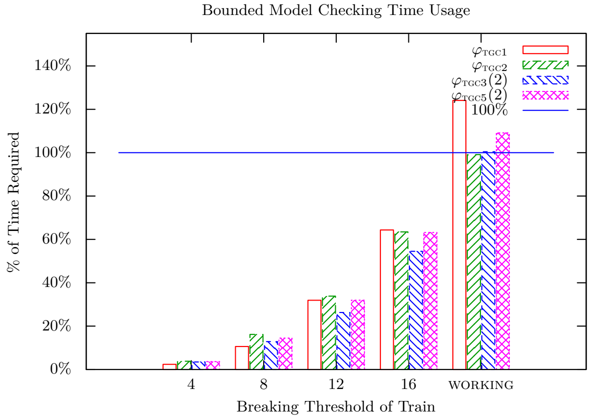


Figure 3: Time used to verify various formulae.



Both graphs demonstrate the advantage of the BDD-based BMC implementation over the current release. For instance, in Figure 2, at a breaking depth of 4, checking the formulae requires, on average, less than 20% of the memory used for full verification.

In Figure 2 it can be seen that there is a marginal overhead when checking $\varphi_{\textsc{tgc}5}$ at a deep breaking depth, but this overhead appears to be less in the working model. The cause of this is that the number of state space generation iterations required to find a counterexample at the deepest breaking bound is greater than that for reaching a fixed point in the working model.

5.3.2 Counterexamples

The approach adopted by traditional SAT-based BMC "finds counterexamples of minimal length" [5]. A comparison of the length of counterexamples generated between `MCMAS`'s BMC implementation and the original implementation can be seen in Table 5. These counterexamples were generated for various properties in a model with two type-2 trains, a maximum counter value of 20 and a breaking depth of 10.

For the first property it can be seen that, although the new

Table 4: Train Gate Controller Properties

φ_{TGC1}	$AG\,(AF\,(\neg\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel}))$
φ_{TGC2}	$AG\,(\neg\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel} \vee \neg\textsc{Train}_2\_\textsc{in}\_\textsc{tunnel})$
φ_{TGC3}	$AG\,(\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel} \rightarrow K_{\textsc{Train}_1}\,(\neg\textsc{Train}_2\_\textsc{in}\_\textsc{tunnel}))$
φ_{TGC4}	$AG\,(K_{\textsc{Train}_1}\,(\neg\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel} \vee \neg\textsc{Train}_2\_\textsc{in}\_\textsc{tunnel}))$
φ_{TGC5}	$AG\,(\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel} \rightarrow K_{\textsc{Train}_1}\,(AX\,(\neg\textsc{Train}_2\_\textsc{in}\_\textsc{tunnel})))$

Figure 1: An example of possible parameterisation

$$AG\left(\textsc{Train}_1\_\textsc{in}\_\textsc{tunnel} \rightarrow K_{\textsc{Train}_1}\left(\bigwedge_{i=2}^{N} \neg\textsc{Train}_i\_\textsc{in}\_\textsc{tunnel}\right)\right)$$

Table 5: Length of counterexamples generated by BMC and full verification.

	Formula				
Method	φ_{TGC1}	φ_{TGC2}	φ_{TGC3}	φ_{TGC4}	φ_{TGC5}
Regular	25	17	4	4	12
BMC	13	16	4	4	FAIL

approach generates a shorter counterexample, it is still not minimal. Length 10—the lowest number of joint actions that must occur before the train can perform a break action—is the shortest. `MCMAS` follows the procedures for counterexample generation as laid out in [8]. The results presented demonstrate that these procedures can be suboptimal.

When attempting to generate a counterexample for φ_{TGC5}, CUDD printed the string `Unexpected Error` and caused `MCMAS` to exit with a non-zero error code. We were able to make `MCMAS` generate a counterexample for this property when performing BMC, but this required manual intervention to cause `MCMAS` to explore the model to a deeper depth than required to falsify the property alone.

5.4 Evaluating Distributed BMC

Table 6 shows the results obtained from performing distributed bounded model checking on a model with 3 trains, a maximum service counter of 7 and a breaking depth of 4. The table displays ratios comparing resource utilisation of seeded BMC and BMC – a value greater (less) than 1 indicates a decrease (increase). Falsification of the parameterised version of φ_{TGC3} was attempted (Fig. 1). The initial fixed-depth BMC was performed to a depth of 4. We can see, that when the property can be falsified, only having to explore a partial state space is greatly favourable. Otherwise, significant over-computation is required to explore each seed to its respective fixed point.

Table 6: A comparison of seeded BMC vs. BMC for a single master and 3 slaves (seed depth of 4).

	Ratio		
Model	Memory	Time	States
FAULTY	1.712	4.561	1.712
WORKING	0.884	0.005	0.003

Table 7: Ratios comparing time for seeded BMC vs. BMC, for a varying number of slaves (seed depth of 3).

# Hosts	Ratio
2	0.016
4	0.031
6	0.044
8	0.057

The faulty train gate controller model is not ideal for benchmarking the distributed aspects, due to the fact that it is possible for the whole model to revert to the initial state, as specified in the ISPL (due to the cyclical nature of the model). This is illustrated by the ratio of states explored in the WORKING model – exploring each seed to its fixed point is the same as exploring the initial state to its fixed point. We explore $|G| \times |\text{seeds}|$ states, where $|G|$ is the total number of global reachable states in the model and $|\text{seeds}|$ is the total number of seeds. This is further highlighted in the memory ratio for the WORKING model – exploring all the seeds requires more memory. This arises from using a seed state as the initial state, which can then lead to a different variable ordering and, subsequently, a larger BDD for the reach set once a fixed point is reached.

Focusing on the model above, in which falsification is not possible, Table 7 shows that increasing the number of slave instances causes a decrease in the time required for the verification process. Unlike the previous results, the initial fixed-depth BMC was only performed to a depth of 3. This table illustrates that increasing the number of hosts available to the verification improves the efficiency of the method.

6. CONCLUSION

In this paper we have presented a method for performing bounded model checking using binary decision diagrams, as opposed to the conventional approach of a conversion of the problem to the Boolean satisfiability problem. Experiments investigating reasoning about the industrial use-case "software development" model and the benchmark scenario "faulty train gate controller" model show bounded model checking to be the preferential approach when automatic variable reordering is disabled. Results demonstrate that the distributed method is an effective technique for harnessing the resources available from multiple hosts in a networked environment. For falsifiable formulae verification efficiency is increased.

This work shows that the adaptation of existing BDD-based model checkers to perform bounded model checking, without significant re-engineering to support SAT, can be fruitful. Not all symbolic model checkers use libraries that provide automatic reordering but, for the ones that do, further research needs to be undertaken to find optimal heuristics to use when performing bounded model checking.

In addition, our future work aims to compare the implementation presented here with the existing SAT-based bounded model checker for multi-agent systems VerICS [15]. The method for distributing the verification process uses only the temporal transition relation to build up the submodel and, therefore, we intend to investigate how the epistemic relations for each agent could be used in a similar way.

Acknowledgements. The work presented in this paper was partially supported by the EPSRC through project E035655.

7. REFERENCES

[1] About CUDD: The U. Colorado BDD Package. `http://www.ece.cmu.edu/~ee760/760docs/cuddv1.pdf`.

[2] Distributed Bounded Model Checking Multi-Agent Systems. `http://code.google.com/p/dbmcmas/`.

[3] The DDDMP Package. `http://fmgroup.polito.it/quer/research/tool/tool.htm`.

[4] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: User Manual. In *cMocha (Version 1.0.1) Documentation*, 1998. `http://mtc.epfl.ch/software-tools/mocha/doc/c-doc/`.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[6] G. Cabodi, P. Camurati, and S. Quer. Can BDDs compete with SAT solvers on Bounded Model Checking? In *Proceedings of the 39th conference on Design automation (DAC '02)*, pages 117–122. ACM, 2002.

[7] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: An Open-Source Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *LNCS*, pages 359–364. Springer-Verlag, 2002.

[8] E. M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-Like Counterexamples in Model Checking. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 19–29. IEEE Computer Society, 2002.

[9] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *LNCS*, pages 436–453. Springer-Verlag, 2001.

[10] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[11] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004.

[12] S. Iyer, J. Jain, D. Sahoo, and E. A. Emerson. Under-approximation Heuristics for Grid-based Bounded Model Checking. *Electronic Notes in Theoretical Computer Science*, 135(2):31–46, 2006. Post-Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2005).

[13] S. K. Iyer, J. Jain, M. R. Prasad, D. Sahoo, and T. Sidle. Error Detection Using BMC in a Parallel Environment. In *Proceedings of the 13th International Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, volume 3725 of *LNCS*, pages 354–358. Springer-Verlag, 2005.

[14] M. Kacprzak, A. L. T. Lasica, W. Penczek, and M. Szreter. Verifying Multiagent systems via Unbounded Model Checking. In *Proceedings of the 3rd NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *LNCS*, pages 189–212. Springer-Verlag, 2004.

[15] M. Kacprzak, W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, B. Wozna, and A. Zbrzezny. VerICS 2007 - A Model Checker for Knowledge and Real-Time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008.

[16] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, volume 5643 of *LNCS*, pages 682–688. Springer-Verlag, 2009.

[17] A. Lomuscio, H. Qu, and M. Solanki. Towards Verifying Compliance in Agent-Based Web Service Compositions. In *Proceedings of The Seventh International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS '08)*, pages 265–272. ACM, 2008.

[18] A. Lomuscio, H. Qu, and M. Solanki. Towards Verifying Contract Regulated Service Composition. In *Proceedings of the 2008 IEEE International Conference on Web Services (ICWS '08)*, pages 254–261. IEEE Computer Society, 2008.

[19] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[20] P. K. Nalla. *Efficient Distributed Bounded Property Checking*. PhD thesis, Wilhelm-Schickard-Institut für Informatik, 2008.

[21] P. K. Nalla, R. J. Weiss, P. M. Peranandam, J. Ruf, T. Kropf, and W. Rosenstiel. Distributed Symbolic Bounded Property Checking. *Electronic Notes in Theoretical Computer Science*, 135(2):47–63, 2006.

[22] W. Penczek and A. Lomuscio. Verifying Epistemic Properties of Multi-agent Systems via Bounded Model Checking. *Fundamenta Informaticae*, 55(2):167–185, 2002.

[23] F. Raimondi and A. Lomuscio. Towards Symbolic Model Checking for Multi-agent Systems via OBDDs. In *Proceedings of the 3rd NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *LNCS*, pages 213–221. Springer-Verlag, 2004.

[24] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.4.1. `http://vlsi.colorado.edu/~fabio/CUDD/`, May 2005.

[25] W. van der Hoek and M. Wooldridge. Tractable Multiagent Planning for Epistemic Goals. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '02)*, pages 1167–1174. ACM, 2002.