A BDD-based BMC Approach for the Verification of Multi-Agent Systems

Andrew V. Jones and Alessio Lomuscio

Department of Computing
Imperial College London
London, UK
{andrew.jones05,a.lomuscio}@imperial.ac.uk

Abstract. We present a technique combining binary decision diagrams and bounded model checking for the verification of a temporal epistemic logic; a method for distributing the bounded verification process is also discussed. We include implementation details of these proposed approaches as an extension to `MCMAS`, a BDD-based model checker for multi-agent systems. The "Train Gate Controller" scenario is adapted to display faults and is used to provide a comparison with the original implementation. When variable reordering in the underlying library is disabled the serial bounded technique is shown to be effective; for satisfiable formulae the overhead imposed is negligible. The initial results for our 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.

1 Introduction

Multi-agent systems (MAS) are distributed systems in which agents, representing processes, exhibit autonomous behaviour. Recent research into the verification of multi-agent systems, using temporal epistemic logics, has highlighted their use in ensuring the correct functionality of various protocols, such as those dealing with authentication and security.

Symbolic model checking [14] is a powerful technique for the verification of reactive systems. Traditionally, such approaches use reduced ordered binary decision diagrams (ROBDDs) to represent the model. These, however, suffer adversely from the state space explosion problem.

Bounded model checking (BMC) [2] attempts to alleviate this difficulty by considering only a truncated model up to a specific depth. Most approaches to bounded model checking look 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 [15]; an experimental implementation is presented in [12].

Performing bounded model checking using BDDs rather than SAT has been investigated previously; a method was originally presented by Copty *et al.* [6] and later extended by Cabodi *et al.* [3]. The main aim of these investigations

was to determine if the benefits gained from performing BMC were due to the "underlying technology" used for model checking – BDDs vs. SAT – or whether the gains derived from the method of model checking – bounded vs. unbounded. Crucially, their approaches focused only on invariant properties in LTL – a limited subset of those expressible. Their relatively simplistic method attempted to falsify a property through a reachability check of a target error set, representing the complement of that property. Cabodi's later work discussed the idea of performing the procedure in reverse, working backwards from the error set using the state pre-image function.

The model checker `NuSMV` [4] attempts to provide a method for "early falsification", again of only invariant properties. 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 of 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 this paper we present a method of bounded model checking for the existential fragment of the epistemic logic CTLK [15], grounded in the interpreted systems formulation of multi-agent systems [7] (Section 2.1). Our approach uses ROBDDs to represent reachable state space [8,14], rather than a translation of the problem to SAT (Section 3). Unlike previous attempts, our method supports the full grammar of the existential fragment, rather than merely the invariant properties. We show that this is not only flexible, but can also be easily extended to support agent verification in a distributed environment (Section 3.2). An implementation of such techniques into an existing model checker for multi-agent systems, `MCMAS` (Section 2.2), is presented, as well as the provision of a scalable scenario (Section 4.1), which allows for a constructive evaluation of our methods when compared with the existing implementation (Section 4.2).

Related Work In addition to the work discussed above, Iyer *et al.* [9] 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 locate errors that may not be locatable by existing methods. Their method uses partitioned ROBDDs 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 [10]. 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 [7] allows for the definition of and reasoning about the behaviours exhibited by a multi-agent system.

Assume that $\mathcal{A} = \{1, \ldots, n\}$ represents the set of n agents in the system, as well as the existence of a special agent, the environment, e, that models 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, which it can perform. The protocol function, $\mathcal{P}_i : \mathcal{L}_i \to 2^{\mathrm{Act}_i}$, governs which actions can be performed by an agent in a given local state (this definition allows for non-determinism). Similarly, assume that the environment is modelled in the same way (i.e. the existence of $\mathcal{L}_e$, 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). Upon this, the environment and each agent has an evolution function, $\tau_i : \mathcal{L}_i \times \mathrm{Act} \to \mathcal{L}_i$, specifying how an agent evolves 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, G, can be represented as the Cartesian product of the local states for all agents in the system $\mathrm{G} \subseteq \mathcal{L}_1 \times \cdots \times \mathcal{L}_n \times \mathcal{L}_e$. A tuple, $(l_1, \ldots, l_n, l_e) \in \mathrm{G}$, represents an instantaneous configuration of the system. The function, $l_i : \mathrm{G} \to \mathcal{L}_i$, 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 $l_i(g) = l_i(g')$.

The set of joint actions can be used to define a transition relation, $\mathrm{T} \subseteq \mathrm{G} \times \mathrm{Act} \times \mathrm{G}$. 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 $\tau_i(l_i(g), a_1, \ldots, a_n) = l_i(g')$.

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 $\mathit{\Pi}(g)$ is the set of all paths starting at the given state ($g \in \mathrm{G}$).

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 $\mathcal{V} : \mathrm{G} \to 2^{\mathcal{PV}}$.

The models of interpreted systems can be used to reason about a branching time temporal epistemic logic. The logic CTLK [15] is an enrichment of Computational Tree Logic (CTL) [8], with modalities for knowledge, although we only consider one here. 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:

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

The epistemic modality $\overline{K}_i\varphi$ is read as "agent i considers it possible that φ". We define $EF\varphi$ as $E[\text{true}U\varphi]$, to mean "there exists a path upon which φ is eventually true". The duals are as follows: $AX\varphi \stackrel{\text{def}}{=} \neg EX\neg\varphi$, $AF\varphi \stackrel{\text{def}}{=} \neg EG\neg\varphi$ and $AG\varphi \stackrel{\text{def}}{=} \neg EF\neg\varphi$. $A[\varphi U\psi]$ 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".

We can define two fragments of CTLK: an *existential* fragment, ECTLK, and a *universal* fragment, ACTLK. 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 $= \{\neg\varphi \mid \varphi \in \text{ECTLK}\}$). We can easily see that ACTLK contains formulae of the kind $AX\varphi$, $AF\varphi$, $AG\varphi$ $A[\varphi U\psi]$ 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, the semantics of CTLK are defined inductively as follows ($\mathcal{M}_{\mathcal{IS}}$ has been omitted for brevity):

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

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 [13] is an existing 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 library, which provides BDD data structures, asynchronous variable reorderings and garbage collection.

MCMAS implements the standard fixed point methods [8] in the algorithm $\textsc{Sat}_{\text{CTLK}}$ [16]. This calculates the satisfiability set of states, $[\![\varphi]\!]$, i.e. the set of reachable states in which φ holds. MCMAS's algorithm $\textsc{Sat}_{\text{K}}$ 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 reachable states, then it is possible to deduce that the formula φ holds at the initial states of the model and is valid.

3 BDD-based BMC

Our method, as outlined in Algorithm 1, directly extends the algorithms presented in [16]. 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: φ ← ¬ψ {φ : ECLTK Formula}
2: Reach ← I {Reach : BDD}
3: while True do
4:    if [[ι → φ]] = Reach then
5:       return False {Counterexample to ACTLK formula found}
6:    end if
7:    Reach ← Reach ∨ (Reach ∧ Trans)
8:    if Reach Unchanged then
9:       break {Fixed point reached}
10:   end if
11: end while
12: return [[ι → ψ]] = Reach
```

Conceptually, whilst similar to the approaches of both Cabodi and Copty ([3] and [6], respectively), ours differs significantly in one major way. Both of the original BDD-based BMC methods merely performed a set intersection between either the reach set or the frontier set of states, with a target error state. In comparison, the algorithm we set forward here performs a full satisfiability check on the current reachable state space at a specific BMC depth.

The algorithm presented has two "exit" points: lines 5 and 12. The first of these is the case that the algorithm has found a counterexample to ψ. As soon as we find the counterexample to the ACTLK formula we are able to terminate the algorithm and cease further construction of the reachable state space. The second exit point (line 12) is only accessible via a `break` in the loop (line 9), the condition for which is reaching a fixed point in the state space, i.e. the set of next states generated is the same as the previous set of next states.

In the implementation we restrict `MCMAS` to only allow for the verification of properties specified in ACTLK when performing bounded model checking. Once the model has been parsed, we construct a `vector` of `pair`s of modal formulae, the first being the ACTLK formulae and the second being the negation of the first in ECTLK. We implemented the method `check_formulae_BMC`, which is called at every depth of state space exploration. The method loops over the vector of modal formula, and removes ones for which the ECTLK formula can be satisfied (i.e. a counterexample of the ACTLK property can be found). State space generation only continues to a deeper depth if there still exist formulae to be verified (i.e. the vector is not empty).

3.1 A Symbolic Method for Epistemic Possibility

Although one requirement of Kripke models is that the transition relation should be serial, the current fixed point methods for CTL (see [8]) are correct even when using non-serial transition relations. Currently, `MCMAS` only supports the "box" modality $K_i\varphi$. To be able 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 SAT_{K} method [16], although this would be inefficient. We extend the original algorithm SAT_{CTLK} with our method for symbolic calculation of $\overline{K}_i\varphi$, as can be seen in Algorithm 2. We refer to this extension as $\text{SAT}_{\text{CTL}\overline{\text{K}}}$.

Algorithm 2 $\text{SAT}_{\overline{\text{K}}}(\varphi : \text{FORMULA}, i : \text{AGENT}) : \textit{set of}\ \text{STATE}$

1: $\text{X} \leftarrow \text{SAT}_{\text{CTL}\overline{\text{K}}}(\varphi)$
2: $\text{Y} \leftarrow \text{pre}_{\text{K}}(\text{X}, i)$
3: **return** Y

The function pre_{K} returns the set of all states that are epistemically accessible for the given agent i; that is, the set of all global states in which the local state of agent i is invariant. We can easily see that the algorithm is correct, given the obvious parallels to SAT_{EX} (see [8]).

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

3.2 Distributed BDD-based BMC

We take inspiration from the work of Iyer *et al.* [9,10] to develop an extension to `MCMAS` and a Java framework to support distributed bounded model checking. In a similar way to both of 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. Our 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, termed a "seed", on the fringe of the current set of reachable states set is saved to the file using the `DDDMP` package[2].

[1] See `http://www.ece.cmu.edu/~ee760/760docs/cuddv1.pdf`
[2] `http://fmgroup.polito.it/quer/research/tool/tool.htm`

3. *Distributed Parallel BDD-based BMC.* Finally, concurrent `MCMAS` BMC instances are started on different hosts for each of these seed states.

The Java framework has two types of instance: "Master" and "Slave". "Master" represents the initial instance that performs fixed-depth BMC, after which this instance becomes a "co-ordination" 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 or a counterexample is found.

Correctness of Distributed Bounded Model Checking For invariant temporal-epistemic formulae the approach of partial state space evaluation used in our method of distributed bounded model checking is sound. For invariant temporal-only formulae the method is also complete.

Proposition 1. *Seeded bounded model checking is* sound *with respect to the full model when a counterexample for $AG(\varphi)$ is found from an individual seed state.*

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.

Proposition 2. *Seeded bounded model checking is* complete *for temporal formulae with respect to the full model when a counterexample for $AG(\varphi)$ cannot be found from any seed state.*

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 CTL, we do not have a path in any part of the model that satisfies $EF(\neg\varphi)$, so $AG(\varphi)$ is valid in the model.

4 Evaluation

4.1 A Scalable Multi-Agent System

To allow us to investigate effectively the efficiency of our BMC implementation we require a scalable model. We have adapted the scenario of the "Train Gate

Controller", as presented by Alur *et al.* [1] and modified by Wooldridge *et al.* [17] and Kacprzak *et al.* [11].

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 exists 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: $\mathcal{L}_{\text{TRAIN}_1} = \mathcal{L}_{\text{TRAIN}_2} = \{Away, Wait, Tunnel\}$ and $\mathcal{L}_{\text{CONTROLLER}} = \{Red, Green\}$. The protocol is omitted as it can easily be inferred.

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 every time a train performs an action; when this counter reaches the maximum value then 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 have defined 3 types of trains: the first, once the break action has been performed, is in the tunnel perpetually; when the second type performs a break action it simply delays the train leaving for that turn (but the train can perform an infinite number of breaks); the final, working, type has all of the break actions removed.

The specifications in Table 1 are falsifiable in a model containing type-1 or type-2 trains, but are satisfiable 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.

Table 1. Train Gate Controller Properties

φ_{TGC1}	$AG\,(AF\,(\neg\text{TRAIN}_1\text{\_IN\_TUNNEL}))$
φ_{TGC2}	$AG\,(\neg\text{TRAIN}_1\text{\_IN\_TUNNEL} \vee \neg\text{TRAIN}_2\text{\_IN\_TUNNEL})$
φ_{TGC3}	$AG\,(\text{TRAIN}_1\text{\_IN\_TUNNEL} \rightarrow K_{\text{TRAIN}_1}\,(\neg\text{TRAIN}_2\text{\_IN\_TUNNEL}))$
φ_{TGC4}	$AG\,(K_{\text{TRAIN}_1}\,(\neg\text{TRAIN}_1\text{\_IN\_TUNNEL} \vee \neg\text{TRAIN}_2\text{\_IN\_TUNNEL}))$
φ_{TGC5}	$AG\,(\text{TRAIN}_1\text{\_IN\_TUNNEL} \rightarrow K_{\text{TRAIN}_1}\,(AX\,(\neg\text{TRAIN}_2\text{\_IN\_TUNNEL})))$

We define the propositional atom "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 $Tunnel$).

The first formula states that TRAIN_1 is infinitely often *not* in the tunnel. The second formula expresses a mutual exclusion property in the model: two trains never occupy the tunnel at the same time. The next specification, φ_{TGC3}, represents that, whenever a train is in the tunnel, it knows that the other train is not. The next expression represents that trains are aware that they have exclusive access to the tunnel. The final property 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 1 can be parameterised in a similar way to the those presented by Kacprzak *et al.* [11]. For example, for a system composed of N trains, the property φ_{TGC3} can be parameterised as follows:

$$AG\left(\text{TRAIN}_1\text{\_IN\_TUNNEL} \rightarrow K_{\text{TRAIN}_1}\left(\bigwedge_{i=2}^{N} \neg\text{TRAIN}_i\text{\_IN\_TUNNEL}\right)\right)$$

This takes the intuitive meaning of: "when a train is in the tunnel, it knows that no other train in the whole system is in the tunnel".

4.2 Results

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. `MCMAS` was branched from version 0.9.7.1 and 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 a 813 GiB 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.

Benchmarks To provide fair benchmarks, we turned off CUDD's asynchronous variable reordering and garbage collection. Our justification for disabling these features is that we wished to evaluate our novel approach, rather than benchmarking a specific implementation.

For instance, if CUDD were to perform asynchronous reorderings 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 reorderings) arising from its use of an auxiliary library.

The memory requirement of bounded model checking, with respect to the original technique, is shown in Figure 1, whilst Figure 2 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 (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.

In Figure 1 we can see that there is a marginal overhead when checking φ_{TGC5} 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.

Fig. 1. Memory required to verify various formulae.

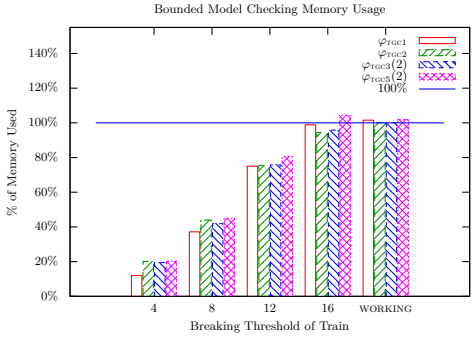
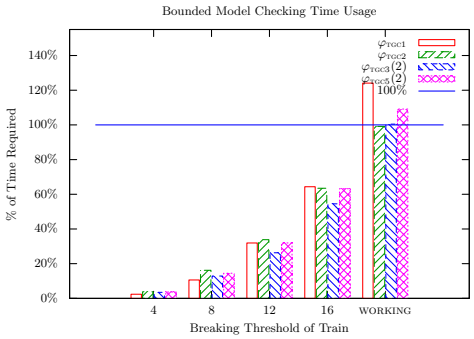


Fig. 2. Time used to verify various formulae.



Counterexamples The approach adopted by traditional SAT-based BMC "finds counterexamples of minimal length" [2]. A comparison of the length of counterexamples generated between `MCMAS`'s BMC implementation and the original implementation can be seen in Table 2. 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 we can see that, although the new 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 [5]. Our results 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.

Table 2. Length of counterexamples generated by BMC and full verification.

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

4.3 Evaluating Distributed Bounded Model Checking

Table 3 shows the possible decreases available when 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). Given our method is only complete with respect to the

temporal fragment, falsification of the parameterised version of φ_{TGC3}, replacing K_{TRAIN_1} with AX, was attempted. 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.

Focusing on the model above, in which falsification is not possible, Table 4 shows us 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.

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

Model	Ratio		
	Memory	Time	States
FAULTY	1.8255	3.8130	1.7297
WORKING	0.9500	0.0013	0.0008

Table 4. Ratios comparing time for seeded BMC vs. BMC, for a varying number of slaves (seed depth of 3).

# Hosts	Ratio
2	0.0033
4	0.0066
6	0.0098
8	0.0131

5 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 looking at reasoning about a faulty train gate controller model show bounded model checking to be the preferential approach when automatic variable reordering is disabled. 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.

Our future work aims to compare the implementation presented here with the existing SAT-based bounded model checker for multi-agent systems, VERICS [12]. We propose to extend our symbolic method for the modality $\overline{K}_i\varphi$ to support the operators $\overline{D}$ and $\overline{C}$, representing the dual of distributed and common knowledge, respectively. Our method for distributing the verification process is not complete for the temporal-epistemic fragment, due to the possible occurrence of epistemically-related states in the temporal past. Therefore, we intend to investigate how our process might be further extended to allow for completeness of epistemic formulae.

References

1. 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.*

2. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579/1999 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin / Heidelberg, 1999.
3. G. Cabodi, P. Camurati, and S. Quer. Can BDDs compete with SAT solvers on bounded model checking? In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 117–122, New York, NY, USA, 2002. ACM.
4. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 241–268. Springer Berlin / Heidelberg, 2002.
5. E. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. *Logic in Computer Science*, 0:19, 2002.
6. 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 *Computer Aided Verification*, volume 2102/2001 of *Lecture Notes in Computer Science*, pages 436–453. Springer Berlin / Heidelberg, 2001.
7. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
8. M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004.
9. 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. Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2005).
10. S. K. Iyer, J. Jain, M. R. Prasad, D. Sahoo, and T. Sidle. Error detection using BMC in a parallel environment. In *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 354–358. Springer Berlin / Heidelberg, 2005.
11. M. Kacprzak, A. Lomuscio, T. Lasica, W. Penczek, and M. Szreter. Verifying Multi-agent Systems via Unbounded Model Checking. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 189–212. Springer Berlin / Heidelberg, 2005.
12. 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. *Fundam. Inform.*, 85(1-4):313–328, 2008.
13. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *CAV*, pages 682–688, 2009.
14. K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.
15. W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundam. Inf.*, 55(2):167–185, 2002.
16. F. Raimondi and A. Lomuscio. Towards symbolic model checking for multi-agent systems via obdds. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 213–221. Springer Berlin / Heidelberg, 2005.
17. W. van der Hoek and M. Wooldridge. Tractable multiagent planning for epistemic goals. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 1167–1174, New York, NY, USA, 2002. ACM.