MEng Individual Project Report

A Method of Bounded Model Checking for a Temporal Epistemic Logic Based on Reduced Ordered Binary Decision Diagrams

Andrew Jones
⟨avj05@doc.ic.ac.uk⟩
Department of Computing
Imperial College London

Supervisor: Dr. Alessio R. Lomuscio ⟨alessio@doc.ic.ac.uk⟩
Second Marker: Prof. Marek Sergot ⟨mjs@doc.ic.ac.uk⟩

Project Archive: http://www.doc.ic.ac.uk/~avj05/mcmas.tgz

June 16, 2009

Abstract

Symbolic model checking 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 infamous state space explosion problem. Bounded model checking – a procedure for "bug hunting" – attempts to alleviate this difficulty by considering only a truncated model up to a specific depth. The possible falsification of a universally quantified formula is shown through a translation of the specification, and the model, to the boolean satisfiability problem (SAT).

We propose a method of bounded model checking for the existential fragment of the epistemic logic CTLK, grounded in the interpreted systems formulation of multi-agent systems. Our approach uses ROBDDs to represent reachable state space, rather than a translation of the problem to SAT. We show that this is not only flexible, but can also be easily extended to support agent verification in a distributed environment. An implementation of such techniques into an existing model checker for multi-agent systems, `MCMAS`, is presented, as well as the provision of a scalable scenario which allows for a constructive evaluation of our methods against the existing implementation.

Acknowledgements

First and foremost, I wish to thank my supervisor, Alessio Lomuscio, for his help and encouragement throughout the project. This project would not have been anywhere near as successful without his enthusiasm and willingness to listen to my problems – as well as reading, and replying to, the gargantuan emails I started to send towards the later stages.

I would also like to thank:

- Marek Sergot for reassuring me of this project's potential during the early stages and for marking this report.
- Michael Huth for being amenable to my visits to his office to ask, what must have seemed to him, very random questions – although they were all very related to this project.
- William Harrower, William Jones and Robin Doherty for their friendship and support through the entire degree – despite their abuse about my love for formal methods. As well as, somehow, at various stages of the degree, finding it in themselves to manage to live with me (I have no idea how Rob managed three years of it).

And, saving the most important until last – my parents, without whose unconditional love, support (and proof reading) this report would not have been possible.

Contents

Chapter 1

Introduction

"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."
H. Barendregt. The quest for correctness. 1996.

1.1 The Problem

Providing assurances about systems is not easy. All the while our daily lives are becoming more and more dependent upon computerised systems, but without any reassurance of the reliability of these devices. The systems with which humans generally interact with are classed as *reactive* systems because of their continual interaction with their environment. It is apparent that some of these systems may contain errors in their software but, in the context of a safety-critical control system for a nuclear power plant, or a plane's flight control system, it should be obvious that *any* kind of bug is unacceptable.

Systems verification looks at determining if a given system meets the required specification. Currently, most verification is a manual effort by humans, which is just as error prone as the design of the system itself. The current approaches to verification are based around exhaustive testing and simulation, but, as humans become even more dependent on these systems, and the systems themselves become increasingly more complex, bugs in these systems can easily be overlooked and missed.

Currently, there is a migration from a "testing" approach to a more thorough "formal methods" approach to this problem. The term *model checking* applies to a collection of *formal* techniques for the analysis and exhaustive state space exploration of these reactive systems.

Formal methods have lead to a rise in tools, such as model checkers, which attempt to *prove* the correctness of software. However, these methods either require an "abstracted" model of the system and, as such, are not entirely representative of the entire system, or they consume a lot of "resources", be these time or memory, to perform the task with which they are presented.

The infamous "state space explosion" problem arises from the attempted verification of systems, or software, which contain a large number of concurrent processes. The resulting interleavings can lead to an unfavourable amount of permutations of state orderings and these, in turn, lead to the "explosion".

Bounded model checking (BMC) (Section 2.3.5), on the other hand, is an attempt to reason about the full system without the requirement of ever exploring the entire the model.

1.1.1 An Illustrative Scenario

Consider an autonomous agent such as NASA's Mars Spirit and Opportunity, both of which landed on Mars in 2004. The rovers were programmed to traverse the Martian landscape collecting data and measurements from rocks on the surface. They were instructed to give priority to rocks which had "green patches", in which NASA scientists were particular interested.

The rovers had limited resources; the batteries could only charge from sunlight, and priority was given to transmitting data back to earth. They had interruptible activities but, during integration, a mistake was made in the code. For example, the following error was introduced: if the sun went down during data collection, the rover would blindly continue to scrape the rock and the data was lost.

This point is illustrated with a trace of such a system in Figure 1.1.

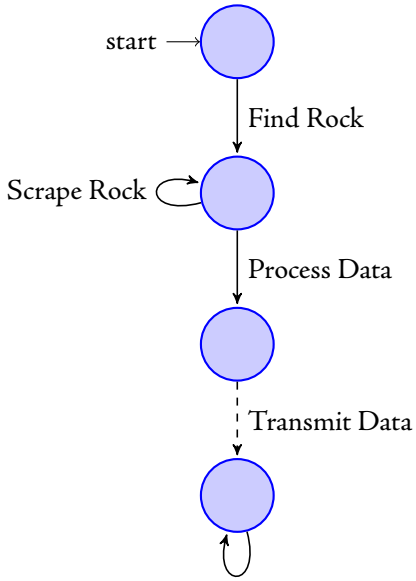

Figure 1.1: A *simple* model of a Mars rover

NASA uses model checking to verify its systems. Assume that they wish to verify that their rovers will *always* transmit the data - this means that there cannot exist a trace of an execution through the model for the agent which does not transmit the data. The approach taken by "Regular" model checking is to build up a set of *every* single possible reachable state in the model, and then see if the property does, or does not, hold.

In comparison, bounded model checking attempts to find a trace through the system in which the rover *never* transmits the data. This is performed as an *incremental check* prior to attempting to find any more accessible states at each iteration of state space generation.

We can see that, when using bounded model checking, the error trace (called a *counterexample*) can be located after reaching the second state. In this instance the whole procedure can return false and only has to explore 2 states, rather than checking 4 states as a conventional model checker would.

1.2 Motivation

Conventional model checking, and bounded model checking, are currently *complimentary* to each other. Most conventional "symbolic" model checkers have implementations that are based on a representation called "Binary Decision Diagrams" (BDDs, Section 2.3.3). In the simplest of terms, these are binary decision trees where isomorphic sub-trees are removed to reduce redundancy.

In comparison, bounded model checking is an approach to overcome the state space explosion problem by a "translation" of the property, and the model, to the boolean satisfiability problem (SAT, Section 2.3.5). Modern SAT solvers take a boolean formula and attempt to find an assignment to each variable contained in it, such that the whole formula evaluates to true.

Most model checkers are based on BDDs requiring a complete exploration of the state space (Section 2.3.1). There is not an obvious conversion from a model checker based on BDDs to being able to perform bounded model checking with SAT.

1.3 Contributions

The main contribution of this report is a *method* for bounded model checking based on BDDs (Section 4.1). This method, unlike other methods for BDD based BMC, is complete and exact; it does not return either false positives or negatives. An evaluation of this method includes an evaluation of using existing fixed point methods for calculating the satisfiability sets when using non-total transition relations. The specifics and implementation details of this method, and the evaluation, can be found in Chapter 4 and Chapter 5, respectively.

This work contains two different approaches to performing bounded model checking:

1. "Full" BMC – Section 4.1 – This performs iterative depth bounded model checking *inside* the model checker in an attempt to find the shortest counterexample to a given specification

2. "One-shot" BMC – Section 5.5 – Rather than performing the satisfiability checks at each incremental depth, we describe a method for performing a single satisfiability check at a given depth. This depth may be less than the depth to find the fixed point.

The *uniqueness* of our approach is that, while there do exist tools which perform bounded model checking on BDDs (Section 2.3.9), these are very *primitive* approaches. They can only verify *liveness* properties which are expressible in terms of atomic propositions. These methods detect a violation of the specification by finding a reachable state in which the proposition does not hold.

In comparison, our method allows for a fuller lexicon of expressions which can not only deal with quantifications of paths through the model, but can also verify *epistemic* properties – ones that express a notion of knowledge.

The final method contributed by this work is an approach of *distributed* bounded model checking based upon exploration of *partial* state spaces on different networked hosts (Section 4.3.4). Our method generates a set of candidate "seed" states, each of which can be used as the initial state on different hosts.

To support this distribution, we further restrict the universal fragment of CTLK to only allow *invariant* properties (Section 4.3.1). This restriction means that the method of distributed partial state space exploration is both sound and complete (Section 5.3).

To allow for the effective evaluation of such a method, we have presented a novel *scalable* scenario – "The Faulty Train Gate Controller", Section 4.4 – which can allow for an adjustable depth for a counterexample to be found. Increasing this bound towards the depth at which the fixed point in the state space is reached allows us to critically evaluate if the method pays an undue overhead. The model presented has a number of *meaningful* parameterised specifications (Section 4.4.3); this allows for the generation of formulae proportional to the number of components in the system.

Chapter 2

Background

2.1 Temporal Logics

It is possible to describe a finite state system (e.g. a reactive one) as a *Kripke structure* [33]:

$$\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$$

Where

- $\mathcal{S}$ is a finite set of states (or "worlds")
- $\mathcal{I}$ is the set of initial states ($\mathcal{I} \subseteq \mathcal{S}$)
- $\mathcal{R}$ is a transition relation between worlds ($\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$) – it is a total relation such that $\forall s \in \mathcal{S},\ \exists s' \in \mathcal{S}\ :\ (s, s') \in \mathcal{R}$.
- $\mathcal{L}$ is a function which labels states from $\mathcal{S}$ with atomic propositions

Temporal logics are used to specify properties about the behaviours of a system defined by Kripke structures. A behaviour of such a system can be obtained by repeatedly applying the transition relation $\mathcal{R}$ to the set of currently reachable states, starting with an initial state $s \in \mathcal{I}$.

Informally, a *trace*, or "run", of a system modelled by a Kripke structure is a sequence of states such that the first state is in the initial states, and each successive state is reachable as per the transition relation. Given that $\mathcal{R}$ is total, all of the traces of these systems are infinite.

This "infinite" behaviour of systems, modelled with Kripke Structures, led to Lamport's [34] classification of the requirements of these systems to fall into two categories:

- **Safety** – "something bad is unable to occur" – A system will satisfy the stated property, if all of the behaviours of the system do not satisfy this property.
- **Liveness** – "something good will eventually happen" – in this case the system must exhibit a specific behaviour to satisfy the property (e.g. returning to the initial state).

There are two main classifications of types of temporal logics:

- **Linear** – These logics allow for the specification of properties of execution sequences of systems (e.g. LTL).
- **Branching** – These logics allow for the specification of the choices available to the system during execution (e.g. CTL).

From this moment on, **AP** is used to represent the set of *atomic propositions*.

2.1.1 Linear Temporal Logic

LTL syntax

Definition 1. The Syntax of LTL formulae is as follows:[1]

- $\forall p \in \mathbf{AP}$, p is a formula
- If φ is a formula, then $\neg\varphi$ is a formula
- If φ and ψ are formulae, then $\varphi \vee \psi$ is a formula
- If φ is a formula, $\mathbf{X}\varphi$ is a formula
- If φ and ψ are formulae, then $\varphi\mathbf{U}\psi$ is a formula

From the above, we can see that the only temporal logic operators that are used in LTL are **X** (ne**X**t) and **U** (**U**ntil).

LTL syntax can also be given in Backus-Naur Form (BNF), where $p \in \mathbf{AP}$:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

The temporal operators **G** ("always", **G**lobally) and **F** ("eventually", **F**uture) can be further defined as:

$$\mathbf{F}\varphi \stackrel{\text{def}}{=} \text{true}\mathbf{U}\varphi$$

$$\mathbf{G}\varphi \stackrel{\text{def}}{=} \neg\mathbf{F}\neg\varphi$$

As a note to the reader **F** and **G** are sometimes written as $\Diamond$ and $\Box$ respectively.

LTL semantics

Definition 2. Semantics of LTL

Let $p \in \mathbf{AP}$, $\mathcal{M}$ be a Kripke structure $(\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, $s \in \mathcal{S}$, φ, ψ are LTL formulae. Satisfaction, $\vDash$, is defined as follows:

$$\begin{aligned}
\mathcal{M}, s \vDash p &\quad \text{iff} \quad p \in \mathcal{L}(s) \\
\mathcal{M}, s \vDash \neg\varphi &\quad \text{iff} \quad \mathcal{M}, s \nvDash \varphi \\
\mathcal{M}, s \vDash \varphi \vee \psi &\quad \text{iff} \quad (\mathcal{M}, s \vDash \varphi) \text{ or } (\mathcal{M}, s \vDash \psi) \\
\mathcal{M}, s \vDash \varphi \wedge \psi &\quad \text{iff} \quad (\mathcal{M}, s \vDash \varphi) \text{ and } (\mathcal{M}, s \vDash \psi) \\
\mathcal{M}, s \vDash \mathbf{X}\varphi &\quad \text{iff} \quad \mathcal{R}(s) \vDash \varphi \\
\mathcal{M}, s \vDash \varphi\mathbf{U}\psi &\quad \text{iff} \quad \exists j \geq 0 : \mathcal{R}^j(s) \vDash \psi \wedge \left(\forall 0 \leq k < j : \mathcal{R}^i(s) \vDash \varphi\right)
\end{aligned}$$

Where $\mathcal{R}$ is defined as per the Kripke semantics given previously, with the one addition that it maps the input state to a unique successor state for a certain *behaviour* through the model. It should be noted that $\mathcal{R}^0(s) = s$, and $\mathcal{R}^{n+1}(s) = \mathcal{R}^n(R(s))$.

2.1.2 Computational Tree Logic

Computational Tree Logic (**CTL**) was introduced by Clarke and Emerson in 1980 [15] - CTL is a branching time logic. It is able to express the existence of, and properties upon, runs of a system.

[1] Adapted from [32], see also [44, 10]

CTL Syntax

Definition 3. The syntax of CTL is defined as follows:

- $\forall p \in \mathbf{AP}$, p is a formula
- If φ is a formula, then $\neg\varphi$ is a formula
- If φ and ψ are formulae, then $\varphi \vee \psi$ is a formula
- If φ is a formula, $\mathbf{EX}\varphi$ is a formula
- If φ is a formula, $\mathbf{EG}\varphi$ is a formula
- If φ and ψ are formulae, then $\mathbf{E}\left[\varphi \mathbf{U} \psi\right]$ is a formula

Syntax of CTL in BNF:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{EX}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}\left[\varphi \mathbf{U} \varphi\right]$$

From the above definition, we can see that CTL has four temporal operators: $\mathbf{EX}\varphi$, $\mathbf{EG}\varphi$, $\mathbf{E}\left[\varphi \mathbf{U} \psi\right]$. From these, we can further define extra temporal operators:

- $\mathbf{AX}\varphi \stackrel{\text{def}}{=} \neg\mathbf{EX}\left(\neg\varphi\right)$
- $\mathbf{EF}\varphi \stackrel{\text{def}}{=} \mathbf{E}\left[\text{true}\mathbf{U}\varphi\right]$
- $\mathbf{AG}\varphi \stackrel{\text{def}}{=} \neg\mathbf{EF}\left(\neg\varphi\right)$
- $\mathbf{A}\left[\varphi \mathbf{U} \psi\right] \stackrel{\text{def}}{=} \neg\mathbf{E}\left[\neg\psi \mathbf{U} \neg\varphi \wedge \neg\psi\right] \wedge \neg\mathbf{EG}\neg\psi$
- $\mathbf{AF}\varphi \stackrel{\text{def}}{=} \mathbf{A}\left[\text{true}\mathbf{U}\varphi\right]$

Semantics of CTL

Definition 4. A *path* [16] in a Kripke structure $\mathcal{M}$ is an infinite series of states $\pi = s_0, s_1, \ldots$ such that $\forall i \geq 0, (s_i, s_{i+1}) \in \mathcal{R}$.

It should be noted that, in comparison to LTL, $\mathcal{R}$, where used here, may returns the set of all successor state, and not a particular successor for a particular behaviour. In CTL, **E** and **A** are *path quantifiers* – **E** represents the **e**xistence of a path, whilst **A** is a quantifier over **a**ll paths (they can be seen as synonymous to $\exists$ and $\forall$).

We can now inductively define the meaning of $\vDash$ (where $\mathcal{M}, s \vDash \varphi$ means φ holds in the state s in the model $\mathcal{M}$, and $\mathcal{M}, \pi \vDash \varphi$ means φ holds along a path π in a model $\mathcal{M}$).

Definition 5. Semantics of CTL

$$\begin{aligned}
\mathcal{M}, s \vDash p &\quad \text{iff} \quad p \in \mathcal{L}(s) \\
\mathcal{M}, s \vDash \neg\varphi &\quad \text{iff} \quad \mathcal{M}, s \nvDash \varphi \\
\mathcal{M}, s \vDash \varphi \vee \psi &\quad \text{iff} \quad (\mathcal{M}, s \vDash \varphi) \text{ or } (\mathcal{M}, s \vDash \psi) \\
\mathcal{M}, s \vDash \varphi \wedge \psi &\quad \text{iff} \quad (\mathcal{M}, s \vDash \varphi) \text{ and } (\mathcal{M}, s \vDash \psi) \\
\mathcal{M}, s \vDash \mathbf{EX}\varphi &\quad \text{iff} \quad \exists \pi = s_0, s_1, \ldots : \mathcal{M}, s_1 \vDash \varphi \\
\mathcal{M}, s \vDash \mathbf{EG}\varphi &\quad \text{iff} \quad \exists \pi = s_0, s_1, \ldots : \forall i \mathcal{M}, s_i \vDash \varphi \\
\mathcal{M}, s \vDash \mathbf{E}\left[\varphi \mathbf{U} \psi\right] &\quad \text{iff} \quad \exists \pi = s_0, s_1, \ldots : \exists k \geq 0 : \mathcal{M}, s_k \vDash \psi \text{ and } \forall 0 \geq j > k \; \mathcal{M}, s_j \vDash \varphi
\end{aligned}$$

In the above, the semantical differences between **E** and **A** can be seen by replacing $\exists\pi$ with $\forall\pi$.

2.2 Multi-Agent Systems

Trends in the fields of interconnected and distributed systems have lead to the introduction of the multi-agent systems paradigm. These are systems comprised of software programs which can "act as autonomous, rational agents" [61]. These so called "agents" are capable of independent actions, as well as communication and co-operation with other agents. The agent acts in an such a way that it can reach its design objectives without being given an explicit way of completing these goals. The idea of a *multi-agent system* is one of many agents *interacting* in a global *environment*. In these systems, agents are able to hold knowledge pertaining to, and express belief about, their environment.

2.2.1 "Agents"

Definition 6. Agents [59] are autonomous systems that

- Perceive the environment in which they are situated (via sensors)
- Act upon the environment (via effectors)
- Are designed with certain "performance" requirements
  - Maintain environment in a certain state
  - Achieve certain state of its environment

An *agent* [43] is:

- *Situated* in an environment
- Capable of *autonomous* action
- Capable of *social* interaction with peers
- Acting to *meet* their design objective

An agent has a set of *local* states $\mathcal{L}$ which represents the current "configuration" of the agent. This configuration might be an assignment to the local variables of the agent, or the values within a knowledge base of known facts. An agent has a set of actions $\mathcal{A}$, and a function which maps from the current state of the agent to the set of enabled actions (a "protocol") for that state $\mathcal{P} : \mathcal{L} \to 2^{\mathcal{A}}$. It is then possible to define an "evolution" function for an agent:

$$\mathcal{T} : \mathcal{L} \times \mathcal{A} \to \mathcal{L}$$

As well as having a set of *local* states, the agent also has an initial state $\mathcal{I}$ which the agent starts in.
This allows us to then define the idea of a run of *an agent*:

Definition 7. [59] A run of an agent is a set of states and actions $(e_0, a_0, e_1, a_1, \ldots)$ such that $e_0 = \mathcal{I}$ (the initial state), $a_0 \in \mathcal{P}(e_0)$, and $\forall i, a_i \in \mathcal{P}(e_i) : \mathcal{T}(e_i, a_i) = e_{i+1}$

2.2.2 Interpreted Systems

"An interpreted system is a semantic structure representing the temporal evolution of a system of agents." [52, 56] We are now assuming that we have a set of n agents i ($i = \{1, \ldots, n\}$) - the local states for an agent i are now represented as $\mathcal{L}_i$. The same is true for the actions ($\mathcal{A}_i$ = agent i's actions), the initial state ($\mathcal{I}_i$ = agent i's initial state) and the protocol ($\mathcal{P}_i$ = agent i's protocol).

The set of n agents act within an "environment" ($\mathcal{L}_E$) which can also be modelled with a set of states – this can be seen as a special agent which can capture any information which may not pertain to a specific agent.

Definition 8. Global States

The set $\mathcal{G}$ of *global states* of a system is:

$$\mathcal{G} \subseteq \mathcal{L}_i \times \cdots \times \mathcal{L}_n \times \mathcal{L}_E$$

A tuple $g = (l_1, \ldots, l_n, l_E) \in \mathcal{G}$ can be seen as a "snapshot" of the current system, where each of the $l_i \in \mathcal{L}_i$. If g is a global state then $l_i(g)$ represents the local state of agent i in the global state g.

If we make the assumption that an interpreted system is a synchronous one, that is, all of the agents within the system transition at the same time, then we can define the global transition function:

$$\tau : \mathcal{G} \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_n \to \mathcal{G}$$

Along with this, we also have an "evolution" function which determines the transitions for an individual agent between its local states. For an agent i, the evolution function t_i is as follows:

$$t_i : \mathcal{L}_i \times \mathcal{L}_E \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_n \times \mathcal{A}_E \to \mathcal{L}_i$$

Similarly, we have an evolution function for the environment's local states, t_E:

$$t_E : \mathcal{L}_E \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_n \times \mathcal{A}_E \to \mathcal{L}_E$$

The set of initial states $\mathcal{I}$, evolution functions t_i and the protocols $\mathcal{P}_i$, define the run of *an interpreted system*:

Definition 9. [52] A *run* of an interpreted system $\pi = (g_0, g_1, \ldots)$ is such that $g_0 \in I$, and for each pair $(g_j, g_{j+1}) \in \pi$ there exists a set of actions a enabled by the protocol $\mathcal{P}$ such that $t(g_j, a) = g_{j+1}$.

Let A be a set of agents $\{1, \ldots, n\}$ with respective local states, protocols and transition functions. The set **AP** is the countable set of propositional variables $\{p, q, \ldots\}$ and $\mathcal{V}$ is the valuation function for those variables $\mathcal{V} : \mathbf{AP} \to 2^{\mathcal{G}}$.

Definition 10. [52]

An *interpreted system* is a tuple:

$$\mathcal{IS} = (\mathcal{G}, \mathcal{I}, \Pi, \sim_1, \ldots, \sim_n, \mathcal{V})$$

Where:

- $\mathcal{G}$ is the set of reachable global states
- $\mathcal{I}$ is the set of initial states $\mathcal{I} \subseteq \mathcal{G}$
- Π is the set of all the possible runs of the system

The binary relation $\sim_i$, $i \in A$ is defined by:

$$g \sim_i g' \text{ iff } l_i(g) = l_i(g')$$

The relation $g \sim_i g'$ represents that the local state of the agent in the current global state is invariant between the two global states. That is, $l_i(g) = l_i(g')$ where the function l_i is the projection function for an agent's local state from the global state. If two states are invariant for an agent, then this means that those states are *indistinguishable* for that agent, and, as such, the agent is unable to distinguish which global state it is currently in.

An interpreted systems *model* [52] From the definition of an interpreted system $\mathcal{IS}$, we can create a model $\mathcal{M}_{\mathcal{IS}} = (\mathcal{G}, \mathcal{I}, \Pi, \sim_1, \ldots, \sim_n, \mathcal{V})$ where:

- $\mathcal{G}$ is the set of reachable global states
- $\mathcal{I} \subseteq \mathcal{G}$ is the set of initial states
- Π is the set of possible runs in the system
- $\sim_i$ is the binary relation for every agent i ($g \sim_i g'$ iff $l_i(g) = l_i(g')$). As before, this represents that the global state is indistinguishable for the agent i (i.e. the local state of the agent is invariant between the two global states).
- $\mathcal{V}$ is the valuation function for the propositional atoms

2.2.3 A logic of knowledge

Interpreted systems provide "computationally grounded semantics that has been used to model knowledge…" [56]. CTLK is an epistemic logic; it allows for the expression of properties which contain *a* notion of *knowledge*.

Syntax of CTLK

Definition 11. BNF definition of the CTLK language

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

Semantics of CTLK

The epistemic modality $\mathbf{K}$ is used to represent "knows" – in logical form, agent i knowing φ is written as $\mathbf{K}_i\varphi$. As such,

$$\mathcal{IS}, g \vDash \mathbf{K}_i\varphi \text{ iff } \forall g' \in \mathcal{G}, g \sim_i g' \text{ implies } \mathcal{IS}, g' \vDash \varphi$$

CTLK is enriched with a further two epistemic operators, for a set of agents $\Gamma \subseteq$ Agents in the System:

"Everybody Knows" [21] The modal operator $\mathbf{E}_\Gamma\varphi$ is exactly true when all members of the group Γ know φ, formally:

$$\mathcal{IS}, g \vDash \mathbf{E}_\Gamma\varphi \text{ iff } \forall i \in \Gamma,\ \mathcal{IS}, g \vDash \mathbf{K}_i\varphi$$

This modality shows that every agent (or, if Γ is a strict subset of all the agents, every agent in the set Γ) knows φ. It is sometimes referred to as "mutual knowledge".

"Common Knowledge" [21] The modal operator $\mathrm{C}_\Gamma\varphi$ is true if all agents in the group Γ know φ, and everyone in the group Γ knows that everyone in Γ knows φ, and so on and so forth. The following abbreviations are useful to help define common knowledge:

$$\begin{aligned} \mathbf{E}^0_\Gamma\varphi &= \varphi \\ \mathbf{E}^1_\Gamma\varphi &= \mathbf{E}_\Gamma\varphi \\ \mathbf{E}^{k+1}_\Gamma\varphi &= \mathbf{E}_\Gamma\mathbf{E}^k_\Gamma\varphi \end{aligned}$$

As a formal definition:

$$\mathcal{IS}, g \vDash \mathbf{C}_\Gamma\varphi \text{ iff } \forall k = 1, 2, \ldots\ \mathcal{IS}, g \vDash \mathbf{E}^k_\Gamma\varphi$$

The computation of common knowledge (calculated using a fixed point) is based upon the following equivalence [21]:

$$\mathbf{C}_\Gamma\varphi = \mathbf{E}_\Gamma(\varphi \wedge \mathbf{C}_\Gamma\varphi)$$

Definition 12. Semantics of CTLK [52]

For an interpreted system $\mathcal{IS}$, global state g, and a formula φ:

$$
\begin{aligned}
\mathcal{IS}, g \vDash p \quad &\text{iff} \quad p \in \mathcal{V}(s) \\
\mathcal{IS}, g \vDash \neg\varphi \quad &\text{iff} \quad g \nvDash \varphi \\
\mathcal{IS}, g \vDash \varphi \vee \psi \quad &\text{iff} \quad (\mathcal{IS}, g \vDash \varphi) \text{ or } (\mathcal{IS}, g \vDash \psi) \\
\mathcal{IS}, g \vDash \varphi \wedge \psi \quad &\text{iff} \quad (\mathcal{IS}, g \vDash \varphi) \text{ and } (\mathcal{IS}, g \vDash \psi) \\
\mathcal{IS}, g \vDash \mathbf{EX}\varphi \quad &\text{iff} \quad \exists\pi,\ \exists i\ :\ \pi_i = g \wedge \pi_{i+1} \vDash \varphi \\
\mathcal{IS}, g \vDash \mathbf{EG}\varphi \quad &\text{iff} \quad \exists\pi,\ \exists i\ :\ \pi_i = g \wedge \forall j \geq i\ \pi_j \vDash \varphi \\
\mathcal{IS}, g \vDash \mathbf{E}[\varphi \mathbf{U} \psi] \quad &\text{iff} \quad \exists\pi,\ \exists i\ :\ \pi_i = g \wedge \exists k \geq 0 : \pi_{i+1} \vDash \psi \wedge \forall j : i \leq j < (i+k)\ ,\ \pi_j \vDash \varphi \\
\mathcal{IS}, g \vDash \mathbf{K}_i\varphi \quad &\text{iff} \quad \forall g' \in \mathcal{G},\ g \sim_i g' \text{ implies } \mathcal{IS}, g' \vDash \varphi \\
\mathcal{IS}, g \vDash \mathbf{E}_\Gamma\varphi \quad &\text{iff} \quad \forall g' \in \mathcal{G},\ g \sim_\Gamma^{\mathbf{E}} g' \text{ implies } \mathcal{IS}, g' \vDash \varphi \\
\mathcal{IS}, g \vDash \mathbf{C}_\Gamma\varphi \quad &\text{iff} \quad \forall g' \in \mathcal{G},\ g \sim_\Gamma^{\mathbf{G}} g' \text{ implies } \mathcal{IS}, g' \vDash \varphi
\end{aligned}
$$

The relation $\sim_\Gamma^{\mathbf{E}}$ is the union of all $\sim_i$, such that $\sim_\Gamma^{\mathbf{E}} = \bigcup_{i \in \Gamma} \sim_i$. While $\sim_\Gamma^{\mathbf{G}}$ is the transitive closure of the $\sim_\Gamma^{\mathbf{E}}$.

π_i represents the global state at position i in a run π. The modalities of **AX**, **EF**, **AF**, **AG**, **AU** can be derived as in CTL.

2.3 Model Checking

Model checking [15, 13] is a method of formal verification, used to verify the correctness of a system. In a nutshell, the problem of model checking is simply: given a description of a finite-state system and property (expressed as a logical formula), does the system satisfy that property? If an error is located, the process will return a *counterexample* showing the steps in which the error state was reached.

> *"Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model"* [32]

> *"Model checking is an effective technique to expose potential design errors"* [3]

The rest of this section intends to concentrate upon CTL model checking.

2.3.1 Explicit Model Checking

The principle behind CTL model checking is, given a model $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ to "label" each state $s \in \mathcal{S}$ with all of the formulae that are valid in s. Then to check if a formula φ is valid in s:

$$\mathcal{M}, s \vDash \varphi \text{ iff s is "labelled" with } \varphi$$

To decide if the model $\mathcal{M}$ satisfies the formula φ, is as simple as checking if *all* the initial states are in the set of states which satisfy φ. That is:

$$\mathcal{M} \vDash \varphi \text{ iff } \mathcal{I} \subseteq \{s \in \mathcal{S} \mid \mathcal{M}, s \vDash \varphi\}$$

The notation $[\![\varphi]\!]$ is used to represent the set of states in a the model in which the formulae φ holds:

$$[\![\varphi]\!] = \{s \in \mathcal{S} \mid \mathcal{M}, s \vDash \varphi\}$$

The algorithms below have been adapted from [32, 24, 50]. For further information, the reader is advised to consult these texts.

Algorithm 1 CTL Model Checking [3]

1: **for all** $i \leq |\varphi|$ **do**
2: **for all** $\psi \in \textsc{Sub}(\varphi)$ with $|\psi| = i$ **do**
3: compute $\textsc{Sat}(\psi)$ from $\textsc{Sat}(\psi')$
4: **end for**
5: **end for**
6: **return** $\mathcal{I} \subseteq (\varphi)$

Definition 13. Sub-formulae of a CTL-formula [32]

Let $p \in \mathbf{AP}$, and φ, ψ be CTL formulae, then:

$$\begin{aligned}
\textsc{Sub}(p) &= p \\
\textsc{Sub}(\neg\varphi) &= \textsc{Sub}(\varphi) \cup \{\neg\varphi\} \\
\textsc{Sub}(\varphi \vee \psi) &= \textsc{Sub}(\varphi) \cup \textsc{Sub}(\psi) \cup \{\varphi \vee \psi\} \\
\textsc{Sub}(\mathbf{EX}\varphi) &= \textsc{Sub}(\varphi) \cup \{\mathbf{EX}\varphi\} \\
\textsc{Sub}(\mathbf{EG}\varphi) &= \textsc{Sub}(\varphi) \cup \{\mathbf{EG}\varphi\} \\
\textsc{Sub}(\mathbf{E}\,[\varphi\mathbf{U}\psi]) &= \textsc{Sub}(\varphi) \cup \textsc{Sub}(\psi) \cup \{\mathbf{E}\,[\varphi\mathbf{U}\psi]\}
\end{aligned}$$

Algorithm 2 $\textsc{Sat}(\varphi : \textsc{Formula})$: *set of* State

1: **if** $(\varphi = \textsc{true})$ **then**
2: **return** S
3: **else if** $(\varphi = \textsc{false})$ **then**
4: **return** $\emptyset$
5: **else if** $(\varphi \in \mathbf{AP})$ **then**
6: **return** $\{s \mid \varphi \in \mathcal{L}(s)\}$
7: **else if** $(\varphi = \neg\varphi_1)$ **then**
8: **return** $\mathcal{S} \setminus \textsc{Sat}(\varphi)$
9: **else if** $(\varphi = \mathbf{EX}\varphi_1)$ **then**
10: **return** $\textsc{Sat}_{\mathbf{EX}}(\varphi_1)$
11: **else if** $(\varphi = \mathbf{E}\,[\varphi_i\mathbf{U}\varphi_2])$ **then**
12: **return** $\textsc{Sat}_{\mathbf{EU}}(\varphi_1, \varphi_2)$
13: **else if** $(\varphi = \mathbf{EG}(\varphi_1))$ **then**
14: **return** $\textsc{Sat}_{\mathbf{EG}}(\varphi_1)$
15: **end if**

State pre-image functions

$$\begin{aligned}
\text{pre}_{\exists}(\mathrm{Y}) &= \left\{s \in \mathcal{S} \mid \exists s' : (s\mathcal{R}s' \text{ and } s' \in \mathrm{Y})\right\} \\
\text{pre}_{\forall}(\mathrm{Y}) &= \left\{s \in \mathcal{S} \mid \forall s' : (s\mathcal{R}s' \text{ implies } s' \in \mathrm{Y})\right\}
\end{aligned}$$

If Y is a set of states, $\text{pre}_{\exists}(Y)$ generates the set of states which *can* transition into Y, and $\text{pre}_{\forall}(Y)$ generates the set of states which *only* transition into Y.

Algorithm 3 $\textsc{Sat}_{\mathbf{EX}}(\varphi : \textsc{Formula}) : \textit{set of } \textsc{State}$

```
1: X ← Sat(φ)
2: Y ← pre∃(X)
3: return Y
```

Algorithm 4 $\textsc{Sat}_{\mathbf{EU}}(\varphi : \textsc{Formula}, \psi : \textsc{Formula}) : \textit{set of } \textsc{State}$

```
1: W ← Sat(φ)
2: X ← S
3: Y ← Sat(ψ)
4: while X ≠ Y do
5:    X ← Y
6:    Y ← Y ∪ (W ∩ pre∃(Y))
7: end while
8: return Y
```

Algorithm 5 $\textsc{Sat}_{\mathbf{EG}}(\varphi : \textsc{Formula}) : \textit{set of } \textsc{State}$

```
1: X ← Sat(φ)
2: Y ← S
3: Z ← ∅
4: while Z ≠ Y do
5:    Z ← Y
6:    Y ← X ∩ pre∃(Z)
7: end while
8: return Y
```

It can be seen that Algorithm 5 and Algorithm 4 these can both be calculated from the least (lfp) or greatest (gfp) fixed point of **EX** [16]:

$$\begin{aligned} \mathbf{EG}(\varphi) &= \text{gfp } Z\,[\varphi \wedge \mathbf{EX}\ Z] \\ \mathbf{E}\,[\varphi \mathbf{U} \psi] &= \text{lfp } Z\ [\psi \vee (\varphi \wedge \mathbf{EX} Z)] \end{aligned}$$

We can also define the temporal operator **EF** in the similar way:

$$\mathbf{EF}(\varphi) = \text{lfp } Z\ [\varphi \vee \mathbf{EX}\ Z]$$

2.3.2 Counterexamples and witnesses

A benefit of model checking is the ability of the model checker to generate *counterexamples* and *witnesses* to properties. In a CTL model, when a *universally* quantified formula is found to be false, the algorithm will generate a counterexample which is "a computation path which demonstrates that the negation of the formula is true" [16]. Likewise, when an *existentially* qualified formula is found to be true, the algorithm will generate a witness which is "a computational path which demonstrates why the formula is true" [16].

2.3.3 Symbolic Model Checking

Binary Decision Diagrams

Binary Decision Diagrams (**BDDs**) [8, 3, 24, 60] or, more commonly, *reduced ordered* binary decision diagrams, are one of the most widely used symbolic data structures for use in model checking. ROBDDs,

- Are canonical, and unique, to each boolean function
- Allow for operations such as negation, conjunction and implication to be easily implemented with a complexity which is directly proportional to that of the inputs.

A BDD is a *directed acyclic graph*, with exactly two terminal nodes (*drains*), one marked 1 (true) and the other 0 (false). Each of the internal nodes represents a single boolean variable and has only two outgoing edges, one solid, representing an assignment of true to that variable, and one dashed (an assignment of false). The node reached to from the "true" path is the value returned by the function $succ_1(u) = v$ where $u, v \in \mathcal{V}$ ($\mathcal{V}$ is the set of nodes in the graph). This is the same as for "false" ($succ_0$).

Boolean operations on BDDs Given two BDDs $\mathcal{B}_f, \mathcal{B}_g$ representing the functions f, g respectively, the BDD for $f \wedge g$ can be obtained by taking the BDD $\mathcal{B}_f$ and replacing all of its 1 terminals with $\mathcal{B}_g$. This is similar for $f \vee g$, except the 0 terminal is replaced [44, 24].

Semantics of BDDs The semantics of a BDD $\mathcal{B}_f$ is the value the terminal node reached when traversing the graph starting from the root node, and taking the corresponding path representing the variable at that node.

Reduction rules A "reduced" BDD ($\mathcal{B}$) is one that has undergone the following transformations, repeatedly, until a fix point has been reached [43, 3]:

- **Elimination** – For two inner nodes u, v, for which $succ_1(u) = succ_0(u) = v$, all of the incoming edges to u are directed to v, and u is eliminated from $\mathcal{B}$.
- **Isomorphism** – If two *distinct* inner nodes u, v of $\mathcal{B}$ are the roots of two structurally identical sub trees, node u is removed and all of its incoming edges are redirected to v.

Variable ordering The ordering in which variables appear in a BDD drastically change the size of the BDD, leading to totally different BDDS.

Definition 14. [24] Let $[x_1, \ldots, x_n]$ be an ordered list of variables without duplications and let $\mathcal{B}$ be a BDD, all of whose variables occur somewhere in the list. We say that $\mathcal{B}$ has the ordering $[x_1, \ldots, x_n]$ if all variable labels of $\mathcal{B}$ occur in that list and, for every occurrence of x_i followed by x_j along any path in $\mathcal{B}$, we have $i < j$.

An *ordered* BDD is one which has *some* ordering for the set of variables it represents. For a fixed variable ordering the BDD representing any propositional formula is uniquely defined. This means that equivalent formulae are all represented by the same BDD.

Literature exists to suggest that it is *generally* a good heuristic to group "dependant" variables closely together in the graph; see [44] for details.

Tests of BDDs [24] For a function $f(x_1, \ldots, x_n)$, and a ROBBD $\mathcal{B}_f$ representing that function. The function is:

- **Valid** – iff $\mathcal{B}_f$ is the single terminal node $\mathcal{B}_1$ representing true.
- **Satisfiable** – iff $\mathcal{B}_f$ is not the single terminal node $\mathcal{B}_0$ representing falsity.

BDD based algorithms There exist a various number of algorithms which are based around BDDs – these are not discussed here; the reader is referred to [24].

Kripke Structures as BDDs

The state of a system can be symbolically represented as the assignment of values to the variables of each state. The transition relation can equally be represented in the same way, as a boolean function between two sets of state variables, from the current state to the next state.

One way of doing this is to assign each $s \in \mathcal{S}$ a unique boolean vector $\{v_1, \ldots, v_n\} \forall i \leq n, v_i \in \{0, 1\}$ (n should be chosen such that $2^{n-1} < |\mathcal{S}| \leq 2^n$, where $|\mathcal{S}|$ represents the total number of states in the model). The boolean vector expressing a state in the system can be based upon the propositional formulae which hold at that state (e.g. for $s \in \mathcal{S}$, the BDD state vector can be based upon the atoms in $\mathcal{L}(s)$). If there are not enough boolean variables to give each state a unique boolean vector, then it should be padded with additional variables such that the value of n is large enough. It is easy enough to see how $\mathcal{I}$ should be represented, given that $\mathcal{I} \subseteq \mathcal{S}$.

The transition relation ($\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$) can be represented as two boolean vectors, the first being the boolean vector representing the originating state and the second being the boolean vector representing the target state.

$\mathcal{L}$ is the function mapping of $s \in \mathcal{S}$ onto propositional atoms in **AP**. It is more convenient to consider it as the converse, mapping atoms to subsets of $\mathcal{S}$ which satisfy that atom. This set of states $\mathcal{L}_p = \{s \in \mathcal{S} \mid p \in \mathcal{L}(s)\}$ [16]. It is easy to see how this set can be represented in the same way as $\mathcal{I}$ (or any other $s \in \mathcal{S}$ for that matter).

2.3.4 BDDs and Variable Orderings

The number of nodes and edges, and, as such, memory, that a ROBBDs requires is directly linked to the variable ordering which has been selected from that BDD. Selecting a "bad" ordering can cause an unfavourable growth in the size of a ROBDD.

The rest of this subsection is an *example* of:

- How to represent a transition system as BDD, and
- A demonstration of how reordering can effect the size of the BDD

Figure 2.1[2] shows a simple transition system with a total transition relation. We can easily see that the model has 4 states, and, as such, each unique state in the model can be represented using 2 bits. An assignment of boolean variables to each individual state can be seen in Figure 2.2.

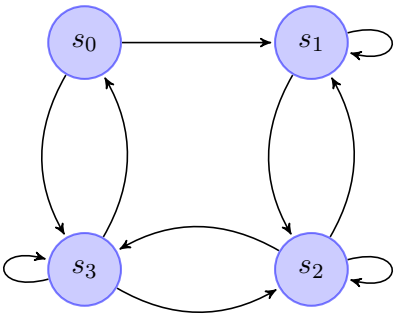

Figure 2.1: A small tranisiton system

State	x_1	x_2
s0	0	0
s1	0	1
s2	1	0
s3	1	1

Figure 2.2: Variable assignments to states

Now that we have a unique "bit string" for each state in the model, it is possible to construct a boolean formula representing the transitions between each state. The boolean function $\rightarrow$ (Figure 2.3) represents the transition relation. The function encodes each transition, and then takes two boolean assignments representing states: the initial state (the unprimed variables) and the next state (the primed variables). If the transition exists in the model, the function will evaluate to true and false otherwise.

[2] Adapted from [64].

$$
\begin{aligned}
\rightarrow (x_1, x_2, x_1', x_2') = \quad & (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \\
\vee \quad & (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge x_2') \\
\vee \quad & (\neg x_1 \wedge x_2 \wedge x_1' \wedge x_2') \\
\vee \quad & (\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2) \\
\vee \quad & \ldots \\
\vee \quad & (x_1 \wedge x_2 \wedge x_1' \wedge x_2')
\end{aligned}
$$

Figure 2.3: A boolean representation of the transition relation

It is not hard to see how the labelling function can be represented in the same way. The function, for each variable, will evaluate to true if the propositional atom holds in that state or not, and false otherwise.

Figure 2.4 shows *one possible* BDD representing the $\rightarrow$ function. It uses the following variable ordering: $x_1 < x_2 < x_1' < x_2'$. The BDD contains 8 nodes and 16 edges (each BDD has two outgoing edges).

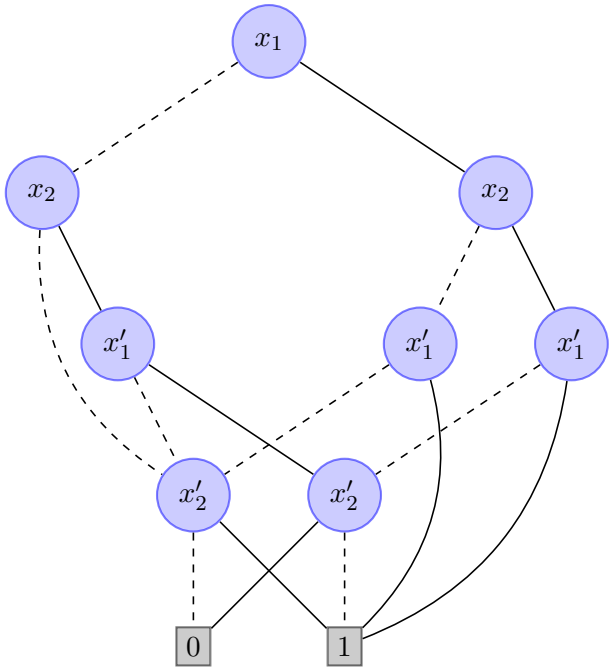

Figure 2.4: One example of ROBDD for the transition relation in 2.3

The BDD in Figure 2.5 represents the *same* boolean function $\rightarrow$, except that it uses a differnt variable ordering: $x_1 < x_1' < x_2 < x_2'$. This second reordering only requires the ROBDD to have 6 nodes and 12 edges.

It is quite obvious to see why, when using ROBDD to perform model checking, selecting a *good* variable reordering is preferable to allow for efficent state space handling.

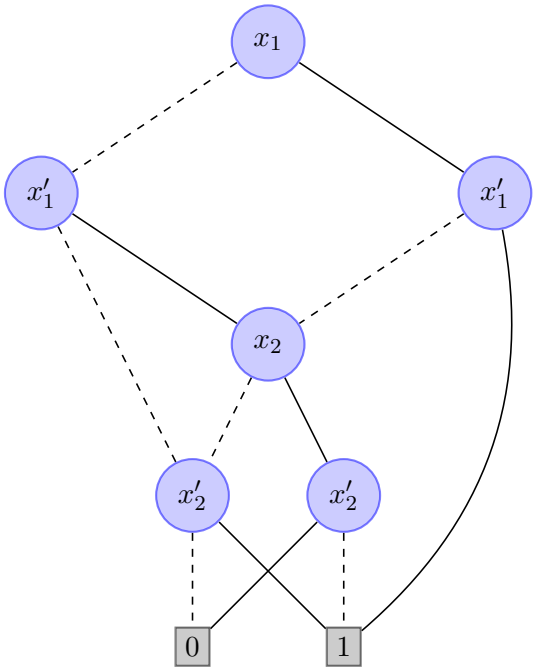

Figure 2.5: A *smaller* ROBDD representing the same transition relation 2.3

2.3.5 Alternatives to BDD Based Model Checking

BMC & SAT

One alternative to symbolic model checking based on BDDs came with the introduction of *bounded model checking* (**BMC**) [7, 5, 6, 14]. BMC searches for the minimum length counterexample which violates the system specification. The algorithm looks for a counterexample with an increasing length ($k = 0, 1, \ldots$) and checks if there exists a computation path in the model which violates the system specification in k steps.

From a temporal logic specification, and a Kripke structure, a propositional formula is generated which is satisfiable if there exists a computational path, with length k, within the model which satisfies the specification. The generated boolean formula is given to a solver, which calculates an assignment to all of the variables comprising of the formula, such that a final evaluation is true. The variable assignment is a *witness* to that path.

SAT, also known as the boolean satisfiability problem, is the problem of trying to find an assignment to all of the variables within a given formula, such that the whole formula evaluates to true.

A crucial part of the bounded model checking algorithm is that, although the path considered is finite, it may still represent an infinite path within the model if it is said to contain a *back loop* from one state in the path to an earlier state in the path. If the path does not contain a loop, then it cannot say anything about the "infinite" behaviour of that path. An example of this is that p might hold at every state path of length k, therefore be seen to be satisfying $\mathbf{G}p$, but without a back loop it cannot witness that formula because, at state s_{k+1} of a path length $k+1$, p may no longer hold.

For a path π in a model, $\pi(k)$ represents the state at element k in the path.

Definition 15. k-path [48]: Let $k \in \mathbb{N}^+$ and $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$. A *k-path* is a finite sequence $\pi = (s_0, \ldots, s_k) : \forall i, 0 < i \leq k, (s_i, s_{i+1}) \in \mathcal{R}$

Definition 16. loop [48]: a k-path π is a *loop* if $\exists l : 0 < l \leq k$ and $(\pi(k), \pi(l))$

Definition 17. k-model [48]: Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ be a model, and $k \in \mathbb{N}^+$. $\mathcal{M}_k = (\mathcal{S}, \mathcal{I}, \mathit{Paths}_k, \mathcal{L})$, where Paths_k is the set of all the paths of length k in $\mathcal{M}$.

Let $\mathcal{M}$ be a Kripke structure, and $\mathcal{M}_k$ be its respective k-model, the function $loop : Path_k \rightarrow 2^{\mathbb{N}}$ is defined as follows:

$$loop(\pi) = \{l \mid l \leq k \text{ and } (\pi(k), \pi(l)) \in \mathcal{R}\}$$

For the rest of this chapter we will be dealing with two restrictions of CTL, one called ECTL - this is a subset of CTL, in which negation can only be applied to propositional atoms $\in$ **AP**. The other called ACTL, $\varphi \in$ ACTL iff $\varphi \in \{\neg\psi \mid \psi \in \text{ECTL}\}$.

Definition 18. Bounded Semantics of ECTL [48]
For a k-model $\mathcal{M}_k$, φ, ψ are ECTL formulae. $\mathcal{M}_k, s \vDash \varphi$ denotes φ holds in the state s of a model $\mathcal{M}_k$. $\vDash$ is defined as follows:

$$\begin{aligned}
\mathcal{M}_k, s \vDash p &\quad \text{iff} \quad p \in \mathcal{L}(s) \\
\mathcal{M}_k, s \vDash \neg\varphi &\quad \text{iff} \quad g \nvDash \varphi \\
\mathcal{M}_k, s \vDash \varphi \vee \psi &\quad \text{iff} \quad (\mathcal{M}_k, s \vDash \varphi) \text{ or } (\mathcal{M}_k, s \vDash \psi) \\
\mathcal{M}_k, s \vDash \varphi \wedge \psi &\quad \text{iff} \quad (\mathcal{M}_k, s \vDash \varphi) \text{ and } (\mathcal{M}_k, s \vDash \psi) \\
\mathcal{M}_k, s \vDash \mathbf{EX}\varphi &\quad \text{iff} \quad \exists \pi \in Paths_k : (\pi(0) = g \text{ and } \pi(1) \vDash \varphi) \\
\mathcal{M}_k, s \vDash \mathbf{EG}\varphi &\quad \text{iff} \quad \exists \pi \in Paths_k : (\pi(0) = g \text{ and } \forall_{0 \leq j \leq k} \mathcal{M}_k, \pi(j) \vDash \varphi) \text{ and } loop(\pi) \neq \emptyset \\
\mathcal{M}_k, s \vDash \mathbf{E}[\varphi \mathbf{U} \psi] &\quad \text{iff} \quad \exists \pi \in Paths_k : (\pi(0) = g \text{ and } \exists_{0 \leq i < k} (\mathcal{M}_k, \pi(i) \vDash \psi \text{ and } \forall_{0 \leq j < i} \mathcal{M}_k, \pi(j) \vDash \varphi))
\end{aligned}$$

$|\mathcal{M}|$, the size of a ECTL model, is defined by the number of states in $\mathcal{S}$. $|\varphi|$, the length of a ECTL formula, is defined as follows:

- if $\varphi \in (\mathbf{AP} \cup \{\neg p \mid p \in \mathbf{AP}\})$ then $|\varphi| = 0$
- if φ is of the form $\mathbf{EX}\alpha$ or $\mathbf{EG}\alpha$, then $|\varphi| = |\alpha| + 1$
- if φ is of the form $\alpha \vee \beta$, $\alpha \wedge \beta$ or $\mathbf{E}[\alpha \mathbf{U} \beta]$, then $|\varphi| = |\alpha| + |\beta| + 1$

Definition 19. Validity of bounded semantics An ECTL formula is valid in a k-model, $\mathcal{M}_k \vDash \varphi$ iff $\forall \iota \in \mathcal{I}, \mathcal{M}_k, \iota \vDash \varphi$

From the bounded semantics above, it can be seen that $\mathcal{M}_k, s \vDash \varphi$ implies $\forall l : l \geq k,\ \mathcal{M}_l, s \vDash \varphi$. Simple induction then shows us that $\mathcal{M}_k, s \vDash \varphi$ implies $\mathcal{M}, s \vDash \varphi$. Another property from above (proof can be found in [48]) is that, if $\mathcal{M}, s \vDash \varphi$, then $\mathcal{M}_k, s \vDash \varphi$ when $k = |\varphi|$.

Creating the propositional formula The function $States(Path)$ generates the set of states from the k-model which can be reached with a path of length k:

$$States(Path) = \{s \in \mathcal{S} \mid \exists \pi \in Paths, \exists i \leq k : \pi(i) = s\}$$

Definition 20. Sub-models of $\mathcal{M}$ [48]
$\mathcal{M}_k = (\mathcal{S}, \mathcal{I}, Paths_k, \mathcal{L})$ is a k-model of $\mathcal{M}$. The structure $\mathcal{M}_k = (\mathcal{S}', \mathcal{I}, Paths'_k, \mathcal{L}')$ is a *sub-model* of $\mathcal{M}_k$, such that $Paths'_k \subseteq Paths_k$, $\mathcal{S}' = States(Paths_k)$, and $\mathcal{L}' = \mathcal{L}|_{\mathcal{S}'}$ (a reduction of the labelling function to only contain states in $\mathcal{S}'$)

Definition 21. The function f_k : CTL Formula $\rightarrow \mathbb{N}$ [48]

- $f_k(p) = f_k(\neg p) = 0$
- $f_k(\varphi \vee \psi) = max\{f_k(\varphi), f_k(\psi)\}$

- $f_k(\varphi \wedge \psi) = f_k(\varphi) + f_k(\psi)$
- $f_k(\mathbf{EX}\varphi) = f_k(\varphi) + 1$
- $f_k(\mathbf{EG}\varphi) = (k+1) \cdot f_k(\varphi) + 1$
- $f_k(\mathbf{E}[\varphi \mathbf{U} \psi]) = k \cdot f_k(\varphi) + f_k(\psi) + 1$

Algorithm 6 BMC($\mathcal{M}$: Kripke Structure, ψ : ACTL Formula) [48]

1: $\varphi \leftarrow \neg\psi$ {φ is an ECTL formula}
2: **for** $k \leftarrow 1$ to $|\mathcal{M}|$ **do**
3: $\mathcal{M}_k \leftarrow k$-model of $\mathcal{M}$
4: Select sub-models of $\mathcal{M}'_k$ of $\mathcal{M}$ with $|Path'_k| \leq f_k(\varphi)$
5: $[\mathcal{M}^{\varphi,\iota}]_k \leftarrow$ propositional formula of the transition relation of all the sub-models of $\mathcal{M}'_k$
6: $[\varphi]_{\mathcal{M}_k} \leftarrow$ propositional formula of the translation of φ over all the sub-models of $\mathcal{M}'_k$
7: $[\mathcal{M}, \varphi]_k \leftarrow [\mathcal{M}^{\varphi,\iota}]_k \wedge [\varphi]_{\mathcal{M}_k}$
8: Check the satisfiability of $[\mathcal{M}, \varphi]_k$
9: **end for**

Construction of the propositional formula $[\mathcal{M}, \varphi]_k$ is as follows. A symbolic representation is used so that the $\mathcal{S} \subseteq \{0,1\}^n$, where $n = \lceil log_2(|\mathcal{S}|) \rceil$. Each state $s \in \mathcal{S}$ can therefore be represented as a vector of propositional variables which hold at that state ($s = \{s[1], \ldots, s[n]\}, s[i] \in \mathbf{AP}$). A k-path can then be represented as a vector of length k of these states ($\pi_k = (s_0, \ldots, s_k)$). $LL^{\varphi} \subset \mathbb{N}^+$ is a finite set of a numbers.

$[\mathcal{M}^{\varphi,\iota}]_k$ constrains $|LL^{\varphi}|$ symbolic k-paths valid in $\mathcal{M}_k$. For $j \in LL^{\varphi}$, the j^{th} symbolic k-path is denoted as $(w_{0,j}, \ldots, w_{k,j})$, where $w_{i,j}\ \forall i \in \{0, \ldots, k\}$ are state variables.

The function *lit* [48] is defined as follows:

$$\begin{aligned} lit(0,p) &= \neg p \\ lit(1,p) &= p \end{aligned}$$

The following are propositional formulas, based upon the usual definition of a Kripke structure, where w, v are state variables [48]:

$$\begin{aligned} I_s(w) &\text{ iff } \bigwedge_{i=1}^{n} lit(s[i], w[i]) \\ T(w,v) &\text{ iff } (w,v) \in \mathcal{R} \\ p(w) &\text{ iff } p \in \mathcal{L}(w),\ p \in \mathbf{AP} \\ H(w,v) &\text{ iff } w = v \\ L_{k,j}(l) &= T(w_{k,l}, w_{l,j}) \end{aligned}$$

- $I_s(w)$ encodes the initial state $\mathcal{I}$ of the model, $s[i] = 1$ is encoded by $w[i]$, and $s[i] = 0$ is encoded by $\neg w[i]$
- $T(w,v)$ encodes a transition between two states (i.e. $T(w,v)$ iff $w\mathcal{R}v$)
- $p(w)$ encodes a proposition of p of ECTL
- $H(w,v)$ represents logical equivalence between states
- $L_{k,j}(l)$ encodes a backward loop connecting the k^{th} state to the l^{th} state in the symbolic k-computation j, for $0 \leq l \leq k$.

The *unrolled transition relation* at bound k, $[\mathcal{M}^{\varphi,\iota}]_k$, is calculated as follows [48]:

$$[\mathcal{M}^{\varphi,s}]_k = \mathcal{I}_s(w_{0,0}) \wedge \bigwedge_{j \in LL^{\varphi}} \bigwedge_{i=0}^{k-1} T(w_{i,j}, w_{i+1,j})$$

Where:

- $w_{0,0}$ and $w_{i,j}$ (for $i = 0, \ldots, k$ and $j \in LL^{\varphi}$) are vectors of state variables
- $|LL^{\varphi}| = f_k(\varphi)$

Finally, the ECTL formula φ has to be translated into a propositional formula $[\varphi]_{\mathcal{M}_k}$. The translation of this formula differs for paths which are, and are not, k-loop paths. These can be distinguished with $L_{k,j}(l)$. At each state $w_{m,n}$ within a k-path of index n, the temporal subformulas of the formula being translated to the k-path n are translated to the k-paths that start at that state. Starting with $w_{0,i} = w_{m,n} \forall i \in LL^{\varphi}$. $[\varphi]_k^{[m,n]}$ is the translation of the formula φ at $w_{m,n}$ to a propositional formula.

Translation of an ECTL formula [48]:

$$\begin{aligned}
[p]_k^{[m,n]} &= p(w_{m,n}) \\
[\neg p]_k^{[m,n]} &= \neg p(w_{m,n}) \\
[\varphi \vee \psi]_k^{[m,n]} &= [\varphi]_k^{[m,n]} \vee [\psi]_k^{[m,n]} \\
[\varphi \wedge \psi]_k^{[m,n]} &= [\varphi]_k^{[m,n]} \wedge [\psi]_k^{[m,n]} \\
[\mathbf{EX}\varphi]_k^{[m,n]} &= \bigvee_{i \in LL^{\varphi}} \left(H\left(w_{m,n}, w_{0,i}\right) \wedge [\varphi]_k^{[1,i]} \right) \\
[\mathbf{EG}\varphi]_k^{[m,n]} &= \bigvee_{i \in LL^{\varphi}} \left(H\left(w_{m,n}, w_{0,i}\right) \wedge \bigvee_{l=0}^{k} L_{k,i}(l) \wedge \bigwedge_{j=0}^{k} [\varphi]_k^{[j,i]} \right) \\
[\mathbf{E}\left[\varphi \mathbf{U} \psi\right]]_k^{[m,n]} &= \bigvee_{i \in LL^{\varphi}} \left(H\left(w_{m,n}, w_{0,i}\right) \wedge \bigvee_{j=0}^{k} \left([\psi]_k^{[j,i]} \wedge \bigwedge_{t=0}^{j-1} [\varphi]_k^{[t,i]} \right) \right)
\end{aligned}$$

To summarise, to create the propositional formula which will be satisfiable for a model $\mathcal{M}$, and formula φ, at a bound k. First, the algorithm has to create $[\mathcal{M}^{\varphi,\iota}]_k$, which is representative of the unrolled transition relation at bound k. Next, the algorithm forms $[\varphi]_{\mathcal{M}_k}$ which will be true if, and only if, φ is valid along a path of length k in the model $\mathcal{M}$. The final stage is to create $[\mathcal{M}, \varphi]_k = [\mathcal{M}^{\varphi,\iota}]_k \wedge [\varphi]_{\mathcal{M}_k}$. This is then passed to a satisfiability solver.

2.3.6 Model Checking Multi-Agent Systems

Interpreted Systems as Boolean Formulae

Given a model of an interpreted system $\mathcal{M}_{\mathcal{IS}}$ (see Section 2.2.2), the number of boolean variables used to represent local states of an agent is as follows:

$$nv(i) = \lceil \log_2 |\mathcal{L}_i| \rceil$$

This means that a global state can be represented with the following number of boolean variables:

$$N = \sum_{\forall i \in \text{Agents}} nv(i)$$

The evaluation function $\mathcal{L}$ is simply a mapping of states of variables in **AP**, so this can work on the boolean variables representing each state. The protocols can also be expressed in the same way.

The transition function t_i for each agent can be represented as a set of conditionals, which, when satisfied enable a transition for an agent between two local states. For more details see [52].

The model checking algorithm in Section 2.3.6 requires a representation R_t of the global transition relation between two global states (g, g') [53]:

$$R_t(g, g') \text{ iff } \forall i \in \text{Agents} : \exists a \in \mathcal{P}(l_i(g)) \wedge t_i(g, a, g')$$

Model Checking CTLK

The algorithms from the section below have been adapted from [52, 53].

Algorithm 7 $\textsc{Sat}_{\textsc{ctlk}}(\varphi : \textsc{Formula}) :$ *set of* $\textsc{State}$

if $(\varphi \in \mathbf{AP})$ **then**
  return $\mathcal{L}(\varphi)$
else if $(\varphi = \neg\varphi_1)$ **then**
  return $\mathcal{G} \setminus \textsc{Sat}_{\textsc{ctlk}}(\varphi_1)$ {$\mathcal{G}$ is the set of all states in the model}
else if $(\varphi = \mathbf{EX}\varphi_1)$ **then**
  return $\textsc{Sat}_{\text{EX}}(\varphi_1)$
else if $(\varphi = \mathbf{E}[\varphi_i \mathbf{U} \varphi_2])$ **then**
  return $\textsc{Sat}_{\text{EU}}(\varphi_1, \varphi_2)$
else if $(\varphi = \mathbf{EG}(\varphi_1))$ **then**
  return $\textsc{Sat}_{\text{EG}}(\varphi_1)$
else if $(\varphi = \mathbf{K}_i(\varphi_1))$ **then**
  return $\textsc{Sat}_{\text{K}}(\varphi_1)$
else if $(\varphi = \mathbf{E}_\Gamma(\varphi_1))$ **then**
  return $\textsc{Sat}_{\text{E}}(\varphi_1)$
else if $(\varphi = \mathbf{C}_\Gamma(\varphi_1))$ **then**
  return $\textsc{Sat}_{\text{C}}(\varphi_1)$
end if

The functions $\text{EX}_{\textsc{ctlk}}$, $\text{EG}_{\textsc{ctlk}}$ and $\text{EU}_{\textsc{ctlk}}$, are the same as in Section 2.3.1, except they use the relation R_t rather than the Kripke structure transition relation, and $\mathcal{G}$ is used instead of $\mathcal{S}$.

As for CTL, we have to define functions to find the pre-image for a set of states, where pre_{K} is the function for the modality **K**. pre_{E} and pre_{C} are defined similarly. As previously, X is a subset of $\mathcal{G}$, i is an agent and Γ is a set of agents

$$\begin{aligned}
\text{pre}_{\text{K}}(\text{X}, i) &= \{g \in \mathcal{G} \mid \exists g' : (g \mathcal{K}_i g' \text{ and } g' \in \text{X})\} \\
\text{pre}_{\text{E}_\Gamma}(\text{X}, \Gamma) &= \{g \in \mathcal{G} \mid \exists g' : (g R^{\mathbf{E}}_\Gamma g' \text{ and } g' \in \text{X})\} \\
\text{pre}_{\text{C}_\Gamma}(\text{X}, \Gamma) &= \{g \in \mathcal{G} \mid \exists g' : (g R^{\mathbf{E}}_\Gamma g' \text{ and } g' \in \text{X} \text{ and } g' \in \textsc{Sat}_{\textsc{ctlk}}(\varphi))\}
\end{aligned}$$

$\text{pre}_{\text{C}_\Gamma}$ is based on 2.2.3.

Algorithm 8 $\textsc{Sat}_{\text{K}}(\varphi : \textsc{Formula}, i : \textsc{Agent}) :$ *set of* $\textsc{State}$

$\text{X} \leftarrow \textsc{Sat}_{\textsc{ctlk}}(\neg\varphi)$
$\text{Y} \leftarrow \text{pre}_{\text{K}}(\text{X}, i)$
return $\neg\text{Y}$

Algorithm 9 $\textsc{Sat}_{\mathrm{E}}(\varphi : \textsc{Formula}, \Gamma : \textit{set of } \textsc{Agent}) : \textit{set of } \textsc{State}$

1: $\mathrm{X} \leftarrow \textsc{Sat}_{\mathrm{CTLK}}(\neg\varphi)$
2: $\mathrm{Y} \leftarrow \mathrm{pre}_{\mathrm{E}_\Gamma}(\mathrm{X}, \Gamma)$
3: **return** $\neg\mathrm{Y}$

Algorithm 10 $\textsc{Sat}_{\mathrm{C}}(\varphi : \textsc{Formula}, \Gamma : \textit{set of } \textsc{Agent}) : \textit{set of } \textsc{State}$

1: $\mathrm{X} \leftarrow \textsc{Sat}_{\mathrm{CTLK}}(\neg\varphi)$
2: $\mathrm{Y} \leftarrow \mathcal{G}$
3: **while** $\mathrm{X} \neq \mathrm{Y}$ **do**
4: $\quad \mathrm{X} \leftarrow \mathrm{Y}$
5: $\quad \mathrm{Y} \leftarrow \mathrm{pre}_{\mathrm{C}_\Gamma}(\mathrm{X}, \Gamma)$
6: **end while**
7: **return** Y

2.3.7 BMC for Multi-agent Systems

Bounded model checking of interpreted systems [47, 35, 46, 62] is based upon the logic of CTLK, and builds upon the bounded model checking method for ECTL. The syntax of ECTL (definition 3) has to be first extended to give an epistemic modality, different from that of CTLK:

Syntax of ECTLK [35]

As for ECTL (definition 3), with the following:

- If φ is a formula, $\overline{K}_i\varphi$ is a formula, $i \in$ Agents

In BNF :

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

ECTLK also includes the following modalities: $\overline{C}_\Gamma\varphi$ and $\overline{E}_\Gamma\varphi$ for $\Gamma \subseteq$ Agents, but these have been omitted here for brevity.

The epistemic modalities, as defined for the existential fragment of CTLK (ECTLK), are defined as the dual of those from CTLK; that is:

- $K_i\varphi \stackrel{\text{def}}{=} \neg\overline{K}_i\neg\varphi$
- $C_i\varphi \stackrel{\text{def}}{=} \neg\overline{C}_i\neg\varphi$
- $E_i\varphi \stackrel{\text{def}}{=} \neg\overline{E}_i\neg\varphi$

The modality $\overline{K}_i\varphi$ stands for "agent i considers it possible φ" [62, 21].

Semantics of ECTLK [62]

Again, as per the ECTL semantics but with:

$$\mathcal{IS}, g \vDash \overline{\mathrm{K}}_i\varphi \quad \text{iff} \quad \exists g' \in \mathcal{G} : g \sim_i g' \wedge \mathcal{IS}, g \vDash \varphi$$

Bounded Semantics of ECTLK [62]

The definition of an interpreted system (§2.2.2) allows for the specification of *multiple* initial states (the set $\mathcal{I}$), BMC reduces this down to only have an *single* initial state: ι.

$$\mathcal{IS}, g \vDash \overline{\mathrm{K}}_i\varphi \quad \text{iff} \quad \exists \pi \in \mathit{Paths}_k : (\pi(0) = \iota \text{ and } \exists_{0 \leq i < k}(\mathcal{IS}, \pi(i) \vDash \varphi \text{ and } g \sim_i \pi(i))$$

In the above, g and g are global states ($\mathcal{G}$ is the set of all global states), $\overline{\mathrm{K}}\varphi$ holds in the global state g if there exists a global state g', such that the local state for the agent i is invariant between the two, and φ also holds in g'. The function l_i is used to extract an agent's local state from the global state. This can also be represented with the relation $g_1 \sim_i g_2$, where the relation $\sim_i$ is defined as for CTLK.

It is worth noting here that, if $\mathcal{IS}, g \vDash \varphi$, then $\forall i \in \textsc{Agents} \ : \ \mathcal{IS}, g \vDash \overline{\mathrm{K}_i}\varphi$, given the relation $\sim_i$ is reflexive[3]. For the semantical definition of $\overline{\mathrm{E}}_\Gamma$ and $\overline{\mathrm{C}}_\Gamma$ the reader is referred to either [62] or [35].

Translation to SAT of a ECLTK formula [62, 47]

As well as having a boolean encoding for propositions such as the initial state, or the proposition variables, BMC has a boolean encoding for the epistemic relation between an agent's local states; that is, equality between two local states:

$$H_i(w, v) \quad \text{iff} \quad l_i(w) = l_i(v), \forall i \in \textsc{Agents}$$

The translation of $\overline{\mathrm{K}}_\mathrm{A}\varphi$ to SAT is as follows:

$$\left[\overline{\mathrm{K}}_\mathrm{A}\right]_k^{[m,n]} \quad = \quad \bigvee_{i \in LL^\varphi} \left(I_\iota(w_0, i) \wedge \bigvee_{j=0}^{k} \left([\varphi]_k^{[j,i]} \wedge H_\mathrm{A}(w_{m,n}, w_{j,i}) \right) \right)$$

2.3.8 Current Model Checking Technology

CUDD

CUDD [58] [68] is a C++ based BDD library which allows for easy code reuse[4]. CUDD provides:

- The data structures necessary for BDD creation, handling and manipulation
- Efficient implementations of BDD functions (and, or, add, ...)
- Utility functions for managing the BDDs
- "BDD managers" – which are basically hash tables for BDD storage

Within the CUDD BDD representation, the lower bits of pointers are used to represent the negative edges from a BDD. It also provides a method of generating Graphviz Dot [72] diagrams for the BDDs it is used to represent.

Operator Overloading As can be seen in fig. 2.3.8, CUDD's C++ API makes extensive use of operator overloading. Importantly:

* – This represents the operation AND upon two BDDs. AND can be used to calculate the intersection ($\bigcup$) between two sets represented as BDDs.

\+ – This represents the operation OR on two BDDs. OR can be used to calculate the union of two sets ($\bigcap$) between two sets represented as BDDs.

! – This represents the unary operation NOT. CUDD performs this operation in constant time [28].

[3] We're dealing with a KT45 logic, and as such, the relation $\sim_i$ is reflexive, transitive and symmetric.

[4] It should also be noted that CUDD supports *Zero-suppressed Binary Decision Diagrams* (ZDDs) and *Algebraic Decision Diagrams* (ADDs), but as these do not concern this project, they will not be covered here

```
int main(int argc, char* argv[])
{
    Cudd bddmgr; // The manager
    bddmgr = Cudd(0,0);
    BDD x = bddmgr.bddVar();
    BDD y = bddmgr.bddVar();
    BDD f = x + y;
    BDD g = y + !x;
    if ( f == g )
    {
        cout << "f is equal to g";
    }
    else
    {
        cout << "f is NOT equal to g";
    }
}
```

Figure 2.6: An example C++ program using the CUDD library [50]

BDD Manager CUDD uses "unique tables" to store BDDs. This ensures that each node is unique – in this context, unique means that there exists no other node labelled with the same variable, which also has the same children.

Cache CUDD contains a cache which is used to store computed results, which allows for the efficient manipulation of BDDs. The default, and maximum, size of CUDD's cache can be chosen by the user – too small a cache will cause useful BDDs to be overwritten. It is the cache which is scanned by the garbage collector to regain memory.

Garbage Collection CUDD uses a "stop world" garbage collector; that is, it stops the entire execution of the program whilst a garbage collection takes place. CUDD keeps a reference count for each node produced by it, recording both internal references (nodes which are internal to CUDD, or nodes which reference other nodes) as well as external references (such as those from the "external" program). It should be obvious that garbage collector is an asynchronous process, and only initiates once the cache reaches a pre-defined threshold at which stage CUDD tries to release some memory.

Dynamic Reordering As covered in Section 2.3.4, the size of a BDD is greatly affected by the variable ordering – CUDD supports a number of *dynamic reordering* algorithms which attempt to reduce the size of a BDD. Reordering within CUDD can either be invoked directly via a call to `Cudd_ReduceHeap`, or it can be triggered asynchronously when the number of nodes in the unique table exceeds a threshold.

The Reordering process is iterated until no further improvement is possible. CUDD contains numerous reordering algorithms; an example of an algorithm is `CUDD_REORDER_SIFT`. It is based upon Rudell's sifting algorithm [55] and, in the most simplistic of terms: each variable is considered in turn and placed at every possible position; once a best position has been identified, this is the new location of the variable in the ordering.

NuSMV

NuSMV [11] is an open source symbolic model checker – it is a reimplementation of a model checker developed at CMU called, unsurprisingly, SMV. It supports both SAT based BMC for verification, as well as BDD based satisfiability methods [12]. Due to the fact that these methods are usually used to solve different types of problems, it allows for interesting avenues of research.

In BDD based model checking, NuSMV first builds up a finite state machine representing the given model; it can then perform a various number of checks, including: fair CTL, LTL (via a reduction to CTL), and others.

When operating in SAT mode, NuSMV can either use its own built SAT solver, SIM, or it can write out the SAT problem in the standard DIMACS format, which allows for the use of external SAT based solvers, e.g. CHAFF. It supports the bounded model checking of LTL properties only. During this procedure it interleaves "problem generation and solution attempt via a call to [a] SAT solver, and iterates until a solution is found or the specified maximum bound is reached" [11].

It is able to operate in a *simulate* mode, in which the user can interactively select the behaviour which the system exhibits. It also stores all of the *traces* of the model checking procedure for the generation of counterexamples and witnesses.

NuSMV provides two alternative [76] ways of calculating the satisfiability of *invariant* properties; that is, properties of the form: $\mathbf{AG}(\varphi)$. An invariant property of that kind means that, in all of the reachable states, φ must hold. Rather than calculate the fixed point of $\mathbf{AG}(\varphi)$ using $\mathbf{gfp}_Z\,[\varphi \wedge \mathbf{AX}Z]$, NuSMV can handle it in two ways:

- If the full set of reachable states from the initial state has been computed, then the check simply results to:

  $$\mathbf{Reachable}(\mathcal{I}) \subseteq [\![\varphi]\!]$$

  Where the function **Reachable** represents the set of reachable states from a given state.

- It can check the property "on the fly" – rather than calculate the full state space, NuSMV can do the following check at each step of the reachability analysis

  $$\mathbf{Reachable}_k(\mathcal{I}) \subseteq [\![\varphi]\!]$$

  The function $\mathbf{Reachable}_k$ computes with a set of reachable states from the given state within k steps.

Verifying multi-agent systems with NuSMV Raimondi et al [51, 36] investigated a method for the verification of multi-agent systems with NuSMV as part of their tool set. NuSVM was used as a tool to generate the set of reachable states for the model. This was then processed by them to encode the epistemic relations, and then passed to a third tool, Akka [65], which was then used to verify the epistemic properties. The processing stage parsed the NuSMV output of all of the reachable states and generated the epistemic relation to the local states which were invariant across multiple global states. Akka is a Kripke model editor, which also supports model testing. The methodology employed for this procedure is outlined in fig. 2.3.8.

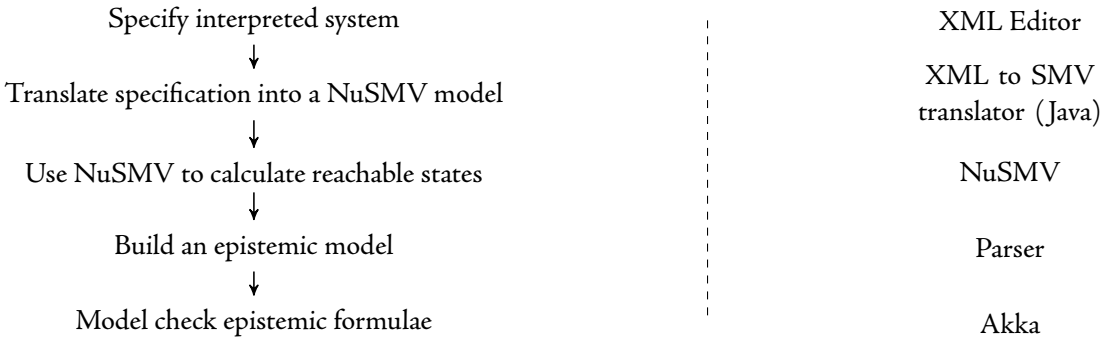

Figure 2.7: Methodology employed to verify multi-agent systems with NuSMV [51]

It should be noted that, given that NuSMV was used *just* as a tool to generate the set of reachable states, it would not be possible to make use of NuSMV's alternatives for handling invariant style properties.

Another approach was attempted by Raimondi et al [37] to reduce CTLK specifications to ARCTL (Action-Restricted CTL) [45] specifications. ARCTL is an extension to CTL, in which qualifications are allowed over

labelled paths interpreted over *labelled transition systems*. ARCTL has the same temporal operators as CTL, except that it allows for the restriction of paths whose actions satisfy a formula φ. There exists *experimental* extension to NuSMV which supports an extended syntax, and allows for the verification of ARCTL properties (which could possibly be a translation of CTLK specifications). [37] provides an extension to the SMV language for the description of interpreted systems and CTLK formulae.

MCMAS

Model Checking Multi-Agent Systems (**MCMAS**) [40] [75] is a *specialised* model checker for the automatic verification of certain aspects of a modelled multi-agent system. It supports CTLK, meaning that it is able to check standard temporal formulae, and ones dealing with epistemic modalities. It is based around the symbolic method introduced in [52], using an external BDD library. It is based around the Colorado University Decision Diagram (CUDD) [68] package. In a similar style to NuSMV, MCMAS is also able to act in an *interactive* way, and allows for the user to interactively select the joint action which should happen.

MCMAS supports the creation of counterexamples (to universal formula) and witnesses (to existential formula). MCMAS supports its own dedicated programming language based on the interpreted systems formalism [21] – ISPL. As per the interpreted systems formalism, MCMAS represents the global state as a BDD composed of each local state for each agent.

ISPL - Interpreted systems programming language MCMAS accepts descriptions of multi-agent systems in the form of ISPL files. These files contain a multi-agent system, in the form of a list of agents each with their own description, and a set of formulae which the user wishes to check. The structure of ISPL files is *roughly* based upon the work presented in [4].

Syntax of an ISPL file [54]

- `Agent` – The name which will be used by MCMAS to represent the agent.
- `LState` – These are the states which are used to the local states ($\mathcal{L}_i$) for each agent
- `Action` – The actions which an agent can perform ($\mathcal{A}_i$)
- `Protocol` – The individual protocol for each agent ($\mathcal{P}_i$)
- `Ev` – The evolution function (t_i)
- `InitStates` – The set of initial states ($\mathcal{I}$)
- `Formulae` – The formulae to be evaluated on the whole MAS
- `Evaluation` – This allows the user to declare atomic propositions based on the local states of each agent
- `Groups` – Allows for the grouping of individual agents into groups (Γ)

ISPL files allow for the definition of "red states" for an agent. These are states which violate some property of the MAS. These states are defined over the local variables of an agent, as well as observable global variables. All other states in the set of local states are labelled as "green states" - if the set of "red states" is empty, all the local states are marked as green states.

Although the core of MCMAS is written using C++, the parsing of the ISPL files is done using Flex [70] and GNU's Bison [71]. The grammar for these files is specified in the `parser/` directory of the source tree. `nssis.ll` is a description file for the lexer, while `nssis.yy` is the file for the parser.

One of the options to MCMAS is to print `bdd-stats`. These are statistics about the BDD, and corresponding memory usage, which has been consumed in model checking the provided MAS. MCMAS is able to generate Graphviz Dot files which represent counterexamples and witnesses, should they exist for the provided model and for formulae.

MCMAS also provides an Eclipse [69] interface which supports the creation of skeleton MCMAS files, as well as syntax highlighting for them. It also provides a graphical interface for executing the checking of ISPL files, and then the examination of the counterexamples/witnesses generated, and their corresponding Dot images.

The internal structure of MCMAS can be seen in fig. 2.3.8.

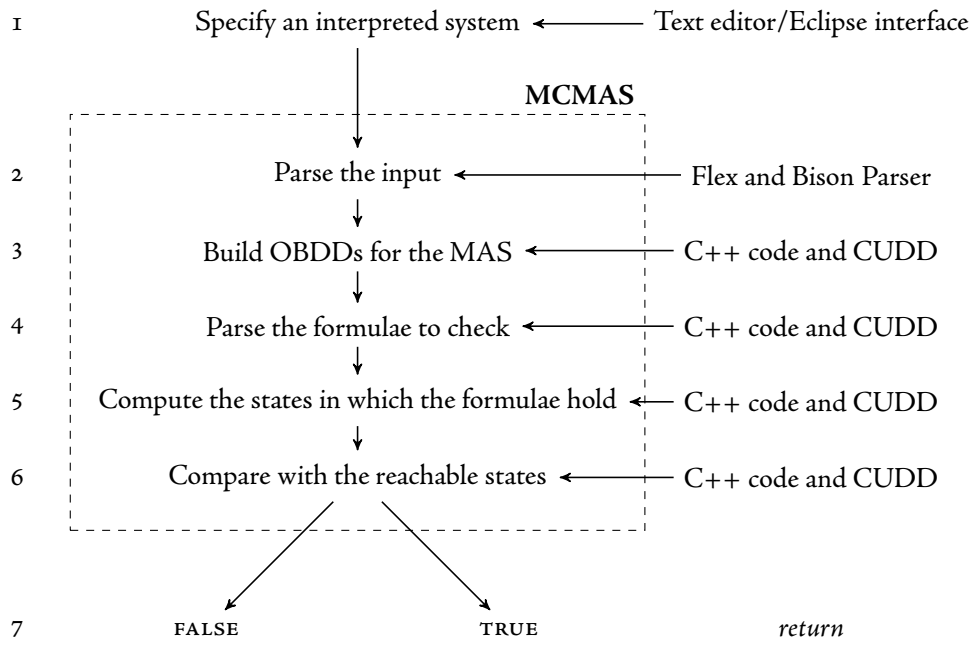

Figure 2.8: MCMAS internal structure [50]

2.3.9 BDD based BMC

In 2001, Fady Copty et al [17] investigated the possibility of using BDDs rather than SAT when performing bounded model checking. The main aim of their paper was to see if the benefits gained from performing SAT-based bounded model checking was due to the "underlying technology" used for model checking – BDDs vs SAT – or whether the gains came from the method of model checking – bounded vs unbounded model checking. They adapted Intel's BDD based *unbounded* model checker Forecast[5] to perform *bounded* model checking.

Given a description of a finite system, with a transition relation **TR**, and a set of initial states **S**, their method attempts to check an invariant property **P** by checking the reachability of the target set **T**, representing the compliment of **P**, from **S**. For each pass of their algorithm a check is made to ascertain if the *frontier* set (the current reach set) and the error set are disjoint. Given a bound k, their algorithm is as follows:

[5] see [17] for details

Algorithm 11 BoundedTraversal(**TR**, **S**, **T**, k)

```
1: Frontier_0 ← S
2: for (i = 0; i < k; i++) do
3:   if (Frontier_i · T ≠ ∅) then
4:     return (failure)
5:   end if
6:   Frontier_{i+1} ← Img(TR, Frontier_i)
7: end for
8: return (pass)
```

The function Img is used to calculate the next set of reachable states, from a given state using the transition relation.

Among other topics Amal et al 2003 [2] discuss the terminating conditions for BDD-based BMC at a depth k

- All paths of length k have been explored
- A state in the target (or error) set has been reached
- All reachable states have been explored (a fixpoint has been reached)

The final conclusions reached by Copty, by comparing Forecast against a SAT-based checker Thunder, seem to suggest that a SAT-based BMC out performs BDD-based BMC, but their comparisons are possibility flawed due to the fundamental differences between the two checkers.

The ideas discussed by Fady Copty et al are further extended by Cabodi et al in 2002 [9]. They discuss the idea of not only *forward* bounded model checking – from the initial set to the target set (FwdVer, Algorithm 12) – but also the converse, this time working from the pre-image of the error set (BwdBMC). They implement their algorithms into a model checker – Forward-Backward Verifier (FBV) – using CUDD, which they then compare against the SAT-based BMC implementation in NuSMV [10]. Whilst only considering *safety* properties, their results seem to suggest that BDD-based BMC scales better with an increasing bound of k.

Algorithm 12 FwdVer(**TR**, **S**, **T**)

```
1: k ← 0
2: R_k = New = S
3: while New ≠ ∅ do
4:   if (T · New ≠ ∅) then
5:     return (CounterEx(R))
6:   end if
7:   k ← k + 1
8:   Next ← Img(TR, Next)
9:   New ← Next · R_{k−1}
10:  R_k = R_{k−1} + New
11: end while
12: return (pass)
```

These results were backed up by Amal et al 2003 [2], when they undertook a more thorough comparison of BMC methods. It should be noted here, that unlike the work by Cabodi et al and Copty et al, Amla et al do not provide *any* form of algorithm, nor any implementation specific information.

Their work presents three BMC approaches:

- BDD based BMC – The paper looks at *liveness* properties when using a bounded reachability check (we assume their approach sis similar to Algorithm 12). One of the cases in which their algorithm terminates is when "an error state is reached".
- Explict State BMC – They perform explict state model checking, but they "kill" all state transitions after a certain depth. They look at "proving a property holds" rather than trying to find a counterexample.
- SAT based BMC – As previously discussed.

They also make the distincton that, unlike Copty's implementation, both their BDD and explict state BMC methods "can produce a positive answer if all the reachable states have been encounter at the depth checked.".

2.4 Distributed Model Checking

There exists various techniques for attempting to alleviate the infamous state space explosion problem (BMC is only one such method), which allow for the automated verification of larger systems. One such approach is an attempt to *distribute* and *parallelise* the computational work load associated with model checking. These are approaches which aim to exploit the resources available in a parallel computing environment, such as a cluster or a grid computing environment, in an attempt to solve larger, more *realistic*, "industrial" sized verification problems [77].

When the model checking procedure suffers from the state space explosion, and, as such, no longer fits completely into the computer's main memory, this causes swapping. Being unable to store the complete state space, and having to utilise backing storage, causes a significant inefficiency in the procedures used.

Many attempts to parallelise model checking involve an attempt to divide up the state space into independent subtasks which can be performed in an arbitrary order in a parallel manner. The intended result is hopefully a quicker, and more efficient, verification, whilst avoiding the slow down associated with swapping. Distributed techniques build on parallel methods and allow for the problem to be distributed between a number of machines, each with disjoint memory.

For instance, there has been research into parallelising BMC, such that multiple solvers look for counterexamples at different lengths [27].

2.4.1 Grid Based BMC with "Seed" States

Another approach towards distributing bounded model checking is to start at different depth "seed states" within the state space [27].

The approach which Iyer et al propose in [27, 25, 26] is to try and find various "candidate deep reachable states" which can then be used as *seeds* to run parallel SAT solvers from in a grid environment. They argue that, when starting SAT based BMC at a deeper state, it is possible to find states deeper in the model, as well as locate errors which may not be locatable by existing methods.

Their method uses partitioned-ROBDDs, and under approximate, to build up a partioned state space such that generating the seeds remains tractable, but this is done at the expense of completeness [26]. Once the memory use of the system exceeds a threshold, they then select only a subset of the next states to continue with forward verification.

Seed states are written out as conjunctive normal form clauses at regular intervals (e.g. after a certain number of next-state computations). These are then used to start "bug hunting" with multiple parallel SAT instances.

Figure 2.9 outlines their approach. The large triangle represents the state space which can be *realistically* explored by conventional SAT based BMC. Instead, BDDS are used to generate an under-approximated state space (the ovals). From this partitioned state space, many parallel SAT instances are started at various depths within the state space ($d_{s1} - d_{s4}$, each representing a different BMC-SAT instance). This is what allows their process to reach errors which would otherwise be difficult to catch.

The justification for their work is to perform "efficent bug-finding" [25] and, as such, their approach only looks at verifying *invariant* properties. Due to their under-approximated state space, their method also sacrifices

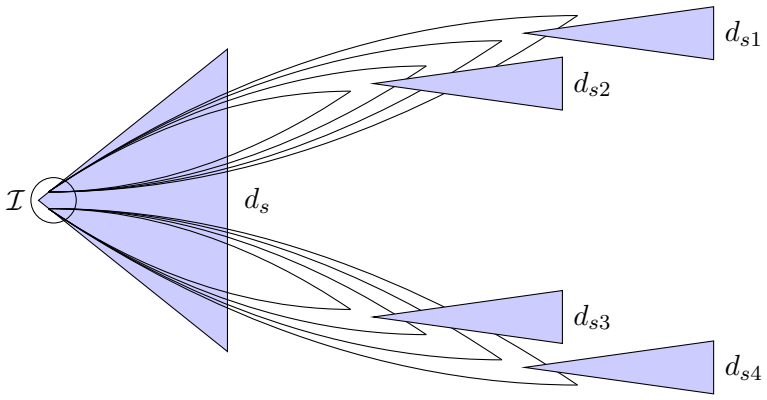

Figure 2.9: Seeding Multiple SAT-BMC runs from POBDD reachability (image adapted from [26])

completeness, although it is sound by construction [26]. If an error is found by a seed state, then the error exists in the design and a trace can be generated from the initial states to the state where the invariant ceased to hold.

2.5 Verifying correctness in real life models

There is growing interest in being able to perform model checking on real life, "industrial", models. The rise of bounded model checking, using SAT solvers, caused the number of industrial cases to rise. BMC performs more of a "bug hunting" approach, and given that most systems *do* contain bugs, BMC can perform favourably.

One area which quite a lot of focus has been given to is the verification of the correct functionality in railway systems. For instance, in [20] Faber looks at the verification of various aspects of the new European Train Control System (ETCS). His work looks at the safety of the railway to prevent crashes. A fuller evaluation of applying symbolic CTL model checking to railway interlock software is presented in [18].

2.5.1 The Train-Gate-Controller Model

An example of a simplified model based upon a real world train system is that of the *Train-Gate-Controller* system.

Alur et al [1] devised this model for use with the Mocha model checker. Their model is based around the idea of two circular train tracks, each with a train travelling in a different direction. At a particular part of the track, the trains must use a tunnel (in the original user manual this was a bridge, but this has evolved over time, and we will be using that formalism), but the tunnel 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, then it knows it is safe to enter the tunnel.

Within Mocha, each of the trains is modelled as a *reactive module*, which can perform two basic actions: `arrive` and `leave`. Each reactive train module contains a single enumerated type, representing the current state of the train: `{away, wait, tunnel}`. Each train also has access to a signal external variable `signal`.

The train module acts as follows: when it arrives at the tunnel it sends the event (i.e. it performs the action) `arrive` to the controller, and checks the `signal` variable (`state = wait`). If the `signal` is red, the train waits and continually checks the signal. When the `signal` turns green, the train enters the tunnel (`state = tunnel`) and, on its exit from the tunnel, the train sends the signal `leave` to the controller, such that it knows that the train is no longer in the tunnel.

To support *multiple* trains within this environment, Alur et al extend the model such that there were two copies of `signal` variable: `signalW` and `signalE` – representing a train approaching from the east or from the west. Similarly, the events `arrive` and `leave` were prefixed with the approach direction, such that the controller could differentiate between the events it witnessed.

In the proposed model, the controller initialised both of the lights to red – when a train arrived and signalled

to the controller, it would check if the other light was red, and only then would the signal be changed to green, allowing the train access to the tunnel. When a train leaves the tunnel, and informs the controller of this fact, it would then change that light back to red.

The behaviour of the trains can be seen in Fig. 2.10. The edge from the `wait` state to the `tunnel` state can be seen as being a *guarded* action (i.e. the action can only be performed when the condition is met).

One limitation of the work in [1] was that the authors did not discuss the properties which should be checked upon the model. The functionalities which the controller should exhibit are briefly discussed in [23]. The controller should:

- Ensure that two trains are never in the tunnel at the same time, and
- Ensure a "smooth running" of the system (e.g. the trains can always eventually move through the tunnel[6]).

Sirjani et al [57] further investigate the Train-Gate-Controller problem with respect to modelling, and verifying, the system using Rebeca [78] – "an actor-based language with a formal foundation". Their paper provides some more concrete properties, in LTL, which they use to verify their Train-Gate-Controller based Rebeca model[7]:

- **Mutual exclusion**

  $$\mathbf{G}\neg\left(\textsc{Train1-InTunnel} \wedge \textsc{Train2-InTunnel}\right)$$

  Only one train should be in the tunnel at one time

- **No deadlock**

  $$\mathbf{GF}\left(\textsc{Train1-InTunnel} \vee \textsc{Train2-InTunnel}\right)$$

  Both trains should eventually pass through the tunnel

- **No starvation**

  $$\mathbf{G}\left(\mathbf{F}\left(\textsc{Train1-InTunnel}\right) \wedge \mathbf{F}\left(\textsc{Train2-InTunnel}\right)\right)$$

  Both trains finally pass through the tunnel (there is always progress)

It should be noted that the final property is attempting to see if the controller acts in a fair way, and will eventually allow any train waiting to pass through the tunnel. Another property which they state, which also corresponds to how controller deals with requests, is:

$$\mathbf{G}\left(\textsc{Controller-Signal1} \rightarrow \mathbf{F}\left(\textsc{Train1-InTunnel}\right)\right)$$

This states that, once a train receives a signal from a train (saying that it is waiting to enter the tunnel), eventually that train does enter the tunnel[8].

A Multi-Agent Train-Gate-Controller

The first time that the Train-Gate-Controller was considered in a multi-agent systems context was in the work by van der Hoek et al [23]. A flaw with the paper, with respect to this current work, is that the paper was concerned with looking at ATL properties on this model. As such, properties expressed in either CTL or CTLK were clearly not given.

[6] This property is attempting to express that *starvation* does not occur within the model

[7] The properties in the paper are:

- specified with $\Box$ and $\Diamond$,
- presented in a hybrid Rebeca/logic/C-like notation
- based on the "bridge" model

I have re-written them here in the LTL style as used previously in this report, as well as adapting them to the "tunnel" scenario.

[8] This can really be seen as a liveness property, but the authors do not state this.

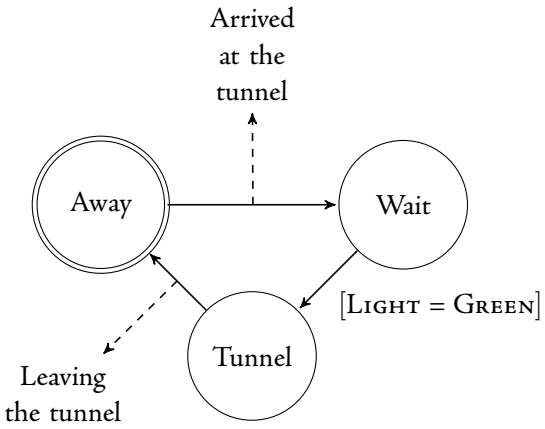

Figure 2.10: An automaton modelling a train from the Train-Gate-Controller model [1, 23]

The work presented by Kacprzak et al in [31, 29, 30] looks at building upon the work of van der Hoek [23], but this time they attempt to present an interpreted systems formalism of the Train-Gate-Controller problem. Their approach is looking to use the Train-Gate-Controller example in *un*bounded model checking on multi-agent sytems.

Their work makes the assumption that the function of the controller is to ensure that two trains are never in the tunnel at one time, and that trains "follow the lights diligently (i.e. they stop on red)". In contrast to the work of Alur and van der Hoek, the controller only transitions to the *red* state once a train enters the tunnel (i.e. the controller is, by default, in the green state).

The local states of the agent in the interpreted systems are as follows [29]:

$$\begin{aligned} \mathcal{L}_{train_1} &= \{away_1, wait_1, tunnel_1\} \\ \mathcal{L}_{train_2} &= \{away_2, wait_2, tunnel_2\} \\ \mathcal{L}_{controller} &= \{red, green\} \end{aligned}$$

The local states take the obvious meanings in the context. The global state, as usual in the interpreted systems, is comprised of all of the local states for each agent, i.e: $\mathcal{G} = \mathcal{L}_{train_1} \times \mathcal{L}_{train_2} \times \mathcal{L}_{controller}$. In the scenario presented in [31, 29, 30] the initial state is taken to be: $\iota = (away_1, green, away_2)$.

The local transition structures, with respect to the joint actions, for the two trains can be seen in fig. 2.11.

Which agents are affected when a joint action takes place, along with their pre- and post-states enabling that action, can be seen in table 2.12. The joint actions from the table 2.12, can be very roughly translated as follows:

- a_1 represents Train1 signalling the controller that it wishes to the enter the tunnel. (similarly for $a_4$with Train2).
- a_2 corresponds to the joint action allowing Train1 to enter the tunnel (again, similarly for a_5).
- a_3 is the joint action in which the train leaves, and signals to the controller (same as for a_6 and Train2).

In [31, 30] attempt to devise epistemic propositions based on a single controller and two trains. They define the following two propositional atoms: in_tunnel_1 and in_tunnel_2 (which take the obvious meanings), the valuation function determining which states they hold as defined as:

- $in\_tunnel_1 \in \mathcal{V}(g)$ iff $l_{\text{Train1}}(g) = tunnel_1$ for $g \in \mathcal{G}$
- $in\_tunnel_2 \in \mathcal{V}(g)$ iff $l_{\text{Train2}}(g) = tunnel_1$ for $g \in \mathcal{G}$

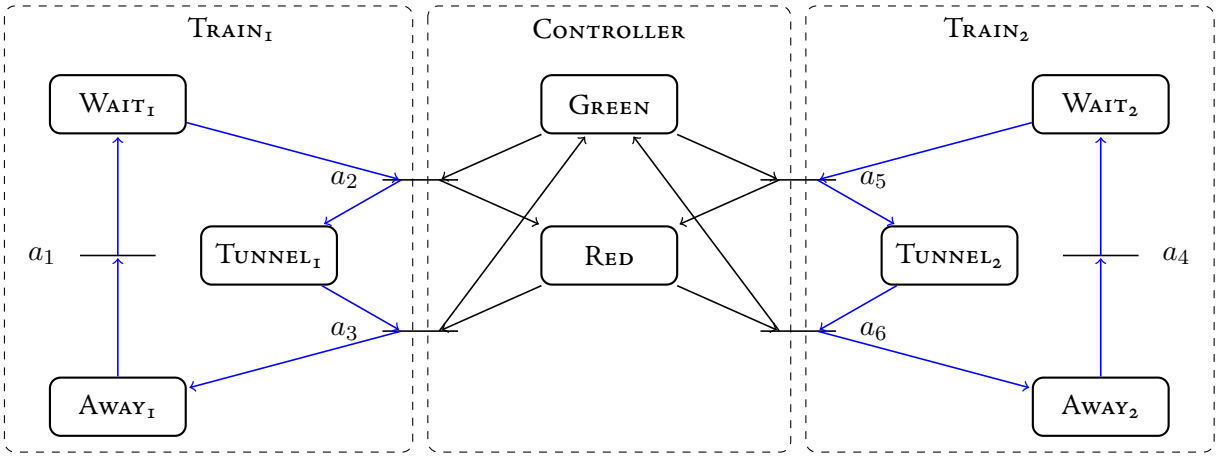

Figure 2.11: The local transition structures for the two trains and the controller [29]

Action	Agent	Pre-State	Post-State
a_1	$train_1$	$away_1$	$wait_1$
a_2	$train_1$ $controller$	$wait_1$ $green$	$tunnel_1$ red
a_3	$train_1$ $controller$	$tunnel_1$ red	$away_1$ $green$
a_4	$train_2$	$away_2$	$wait_2$
a_5	$train_2$ $controller$	$wait_2$ $green$	$tunnel_2$ red
a_6	$train_2$ $controller$	$tunnel_2$ red	$away_2$ $green$

Figure 2.12: Descriptions of Actions in the Train-Gate-Controller [29]

Upon these propositional atoms, they then build the following formula:

1. $\varphi_0 = \neg\mathbf{AX}(\neg in\_tunnel_1)$
   It is possible that the first train will be in the tunnel in the next state
2. $\varphi_1 = \mathbf{AG}(in\_tunnel_1 \rightarrow \mathrm{K}_{train_1}(\neg in\_tunnel_2))$
   When the first train is in the tunnel, then it knows the second train is not in the tunnel
3. $\varphi_2 = \mathbf{AG}(\neg in\_tunnel_1 \rightarrow (\neg\mathrm{K}_{train_1}(in\_tunnel_2) \wedge \neg\mathrm{K}_{train_1}(\neg in\_tunnel_2)))$
   When the first train is not in the tunnel, it does not know if the other train is in the tunnel or not.

The novel approach presented in [29] is to build a *parameterized* model, and a supporting formula, to see the effective of attempting to verify properties exposed to "combinatorial explosion". They generalise the property φ_2 with N trains:

$$\varphi_2(\mathrm{N}) = \mathbf{AG}\left(\neg in\_tunnel_1 \rightarrow \left(\neg\mathrm{K}_{train_1}\left(\bigwedge_{i=2}^{N} \neg in\_tunnel_i\right) \wedge \neg\mathrm{K}_{train_1}\left(\bigvee_{i=2}^{N} in\_tunnel_i\right)\right)\right)$$

.

Chapter 3

Preliminaries

3.1 Discussion on Prior art

Nearly all approaches to bounded model checking look at conversion of the model, and the property, to that of the boolean satisfiability problem. This is not an ideal solution; there is no *obvious* way to directly convert an existing model checker using binary decision diagrams to the SAT problem without *significant* re-engineering. Plus, this then adds an extra requirement for either an external SAT solver, or the implementation of one. This point is clearly illustrated, with the exception of NuSMV, by the distinct lack of existing model checkers supporting both BDD methods and a translation to SAT.

The conversion to SAT also ignores a lot of optimisations, such as variable reordering, which have been designed specifically for model checking. The research which has been put into optimising BDD based methods heavily eclipses that of the research for model checking methods based on SAT.

Another problem with a majority of the existing approaches to BMC is that (with the exception of the *unstated* algorithm in [2]) they only deal with the falsification of properties. This is not a favourable solution, given that BDD based methods of satisfaction also enable the user to prove if a property *is* satisfiable.

When performing *forward verification*, as in Copty's approach, we can be more intelligent. The majority of most symbolic model checkers perform forward verification, building up a reachable state space until a fixed point is reached. An approach such as Copty's, which takes this fixed point into consideration, is clearly a preferable solution, as opposed to assuming that not finding a bug at a *given* depth is an indication that no bug exists at *any* depth.

Approaches taken by NuSMV for invariant satisfaction, or the previously discussed BDD based bounded model checking methods, only look at very *simple* properties – that is, properties which can be expressed through the assignment of propositional atoms. NuSMV's approach is to check if the reachable states are a subset of the states in which the atom holds. In contrast, the BDD based BMC approaches of [17, 9, 2] look at trying to falsify *safety* properties through the intersection of the reachable states and the error states, which invalidate the *safety* property.

It should be immediately obvious that the full lexicon of expressible properties in a logic such as CTLK *cannot* be expressed by simply providing the model checker with a single state and then attempting a reachability check.

3.2 CUDD Specifics

The API provides the functionality for quantification of variables within a BDD. The function `ExistsAbstract` [63] builds the following BDD, where x, y and z are the BDDs representing the variables which we wish to quantify with respect to:

$$\mathcal{B}_h = \exists\,(x, y, z)\,\mathcal{B}_f$$

An implementation of the above quantification (using CUDD) can be seen in fig. 3.1 [63]. Conceptually, `ExistsAbstract` (and the similar function `Exists`) use *Shannon's expression* [16] to construct the quantified BDD:

$$f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1})$$

This quantification attempts to express if it is possible to make the BDD (representing the function f) $\mathcal{B}_f$ true by an assignment of either false (right hand side) or true (left hand side) to the variable x.

$f|_{x \leftarrow b}$ is a *restriction* upon the variable x, as used in the function f, to a value of $b \in \{0, 1\}$. The BDD for a restriction to 0 can be computed by removing the node n, representing the variable x, and redirecting all incoming edges to $succ_0(n)$. A restriction to a 1 can be computed in the same way, but the incoming edges are directed to $succ_1(n)$.

```
// build the extracted BDD xyz
BDD temp = x * y * z;
// h = there_exists(xyz)f
BDD h = f.ExistAbstract(temp);
```

Figure 3.1: Existential quantification of variables within CUDD

3.3 MCMAS Internals

This section attempts to inform the reader of some implementation specifics which occur in the current version[1] of MCMAS.

3.3.1 Global Variables

The following is a list of global variables which are used throughout MCMAS' code:

- `BDD reach` – *the reachable states* – a BDD representing the current reachable set of states
- `BDDvector *v` – *the local states* – a vector of BDDs, each describing an *exact* local state for each agent (see previous part)
- `BDDvector *pv`[2] – *the next states* – a vector of BDDs, each describing a *unique* local state for an agent
- `BDDvector *vRT` – *the transition relation* – a *per agent* mapping between `v` and `pv`, constructed from the conjunction of the protocol and the evolution function (both represented as BDDs).
- `map<string, basic_agent *> *is_agents` – *the interpreted systems agents* – an `std::map` of strings (of agents' names) to instances of agents.
- `modal_formula_vector *is_formulae` – *the interpreted systems formulae* – an `std::vector` of all the formulae, as given in the ISPL for that model.

3.3.2 Important Classes

`Object`

This is a base class which the majority of classes extend – it is very similar to the `Object` class in Java. Importantly, the class includes a virtual `to_string()` method.

`modal_formula : public Object`

This is the class which MCMAS uses to represent and store modal formulae. When the ISPL code is parsed it generates `modal_formula` and stores them in the `is_formulae` vector.

[1] Version 0.9.7

[2] "A boolean vector is an array of BDDs where each BDD represents one bit of an expression" [67]

Variables

- `unsigned char op` – MCMAS does not support any kind of inheritance to distinguish between different types of modal formulae. Each type of modal formula has an associated `unsigned char` value (e.g. an atomic proposition is represented by 0, **AG** is represented by 10, and K is represented by 30).
- `Object* obj[]` – MCMAS uses this variable to store the "arguments" to the formula, for example another `modal_formula`.

Methods

- `BDD check_formula()` – Checks the modal formula with respect to the current `reach` set. The return value is the BDD representing the set of states in which the formula holds.
- `modal_formula * push_negations(int level)` – "Pushes" negations down a certain number of `levels` through the entire formula (e.g. using De Morgan's Laws or re-writing formula with a leading negation to use the dual). Returns a pointer to a new formula with pushed negations.
- `bool is_ACTLK()` – Checks if the given formula is in ACTLK.
- `bool is_ECTLK()` – As above, except for ECTLK.

`basic_agent`

Used to represent an agent within MCMAS. Each agent has a respective: `name` (its name); `vars` (the variables which comprise an agent's local state); `actions` (the actions that agent can perform); `protocol` (which actions can be performed in a given state); `evolution` (how an action affects an agent's local state).

3.3.3 Satisfiability checking within MCMAS

As stated in Section 2.3.1, a model ($\mathcal{M}$) satisfies a formula (φ) if the all the initial states ($\mathcal{I}$) are included in the set of states at which that formula holds. That is:

$$\mathcal{M} \vDash \varphi \text{ iff } \mathcal{I} \subseteq [\![\varphi]\!]$$

This means that, to check the satisfiability of the formula φ, we can construct (and check the satisfiability of) the formula:

$$\iota \rightarrow \varphi$$

where ι represents a propositional atom which is true at the initial states of the model only. This implication is *implicitly* true at all states except the initial ones (the antecedent is false, so the formula is true regardless of the value of the consquent). MCMAS employs this method to easily check the satisfiability of a formula over an entire model. If $[\![\iota \rightarrow \varphi]\!]$ is equal to reachable states (`reach`) we can deduce that the formula φ holds at the initial states of the model.

More in-depth *implementation specifics* of MCMAS, including the construction of SAT_{CTLK} (with the exception $\overline{\text{K}}$), are *not* discussed here. The reader is referred to [49] [74] for further information.

3.4 Models

MCMAS[3] comes with a set of example models, the following four of which all have properties which are expressible in ACTLK[4]. Some of the properties given below have been constructed, for this work, purely to adhere to the logic ACTLK.

The Bit Transmission Problem

Imagine two processes, a sender S and a receiver R, who communicate over a *possibly* faulty communication line. S continually sends a bit to R, until it receives an `ack` from R. R does nothing until it receives a bit from S, and then infinitely sends an `ack` to R. If S receives the `ack` from R, then S knows R has received the bit. Given that S does not acknowledge the `ack`, R will never know if S received the *ack*.

An expression such as this can be formalised in CTLK [41]:

$$\mathcal{IS} \vDash AG\left(\textbf{recack} \rightarrow K_S\left(K_R\left(\textbf{bit = 0}\right) \vee K_R\left(\textbf{bit = 1}\right)\right)\right)$$

Lomuscio et al [42] extended this model to include possible failures. Their work added the following faults to the receiver:

1. The protocol for the agent no longer enforces it to send an acknowledgement when it receives a message, and
2. It allows for the possibility that it can send an acknowledgement without previously receiving a message

ACTLK Properties The property is *true* on the original BTP model, and *false* on the faulty model:

- φ_{BTP1} – `AF(K(Receiver,bit0) or K(Receiver,bit1))`

The Dining Cryptographers

The Dining Cryptographers is a problem which was introduced by Chaum in 1988 to illustrate the anonymous sending of messages with unconditional send and recipient untraceability. The idea is as follows: three cryptographers are out for dinner and learn that their meal has already been paid for, but they desire to discover who has paid – one of them, whilst staying anonymous, or their employer, the National Security Agency. They devise the following protocol: each of the cryptographers flips a coin behind their menu so that only they and the person to their right can see the output. The cryptographers then announce if the two coins which they can see (theirs and the one to their left) is the same or different. If one of the cryptographers has paid of the meal, then he, or she, will announce the opposite to the difference of the coins that they can see. If an even number of "same" the NSA has paid.

This problem can be modelled as a multi-agent systems problem, where each of the cryptographers is an agent and the environment encapsulates the values of the coins. The environment non-deterministically chooses if a cryptographer or the NSA has paid. A "cryptographer" agent has four local variables, one for each coin, one stating if the coins are the same or different, and one saying if that agent has paid or not. The protocol of each agent determines if they should lie or not, given if they are the payer or not.

MCMAS can then be used to check if there is an odd number of "same", which means that a cryptographer has paid. If an agent has not paid, and there is an odd number of "same" utterances, then the agent knows that someone paid, but he does not know who.

ACTLK Properties

- φ_{DC1} – `AG((odd and !c1paid) -> (K(DinCrypt1, c2paid or c3paid)))` – True
- φ_{DC2} – `AG((odd and !c1paid) -> (K(DinCrypt1, c2paid or c3paid)) and (K(DinCrypt1, c2paid) or K(DinCrypt1, c3paid)))` – False

[3] As of version MCMAS 0.9.8.3.

[4] Some of these do not have *obvious* translations to English.

"Software Developement"

The work of Lomuscio et al in [39, 38] presents a model based upon the composition of services based upon a contract. Their model contains seven agents: "a principal software provider (PSP), a software provider (SP), a software client (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) wants a piece of software developed and deployed on a hardware supplier (H) by the technical expert (E). There are two parties which are to 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.

If any of the above parties deviate from the above (e.g C requires software changes which either PSP or SP do not agree with, or the software fails in testing too many times) the contract is violated.

ACTLK Properties

- φ_{SD1} – `A(HardwareSupplier_green U HardwareSupplier_end)` – False – The hardware supplier is always in compliance until it has finished the contract.

The Book Store

This model is similar to the "software development model" found above, in as much as it deals with contract violations. The model contains two agents: a Purchaser and a Supplier.

The `Supplier` waits for an order from the purchaser and decides if it should accept, or rescind, the order. The agent then waits to receive the payment, which it can then accept or decline. The "e-goods" (books) are made available to the purchaser; if the purchaser is unhappy with the goods the supplier can offer a "remedy" or a refund. If, at any stage, the supplier does not "follow protocol" (i.e. it performs `terminate` action) the contract is violated.

The `Purchaser` initiates a contract with the supplier, pays for the goods and downloads them. If the Purchaser is unwilling to accept the goods it can return them to the Supplier. The agent can violate the contract by refusing to pay for the goods (by performing the `terminate` action).

ACTLK Properties

- φ_{BS1} – `AF (K(Supplier, contract_success))` – False – The supplier, at some stage, knows that the contract has ended successfully. That is, both parties have adhered to the protocol and both goods and payment have transferred hands. This is violated at any stage, by either of the parties, by performing the `terminate` action.

- φ_{BS2} – `AG (payment_received -> AF(supplier_compliance))` – True – When the supplier receives the payment it complies with the rest of the transaction

- φ_{BS3} – `AG (payment_received -> AF(AX(supplier_compliance))` – False – When the supplier receives the payment *from the next state*[5] it complies with the transaction.

- φ_{BS4} – `AG ((supplier_compliance and purchaser_compliance) -> K(Supplier, AF contract_end))` – False – If both parties comply the supplier knows that eventually the contract will end (i.e. it will not be prematurely terminated).

[5] In the previous property, supplier_compliance holds in the same state in which payment_received holds, which, when using *weak* until, is true.

Chapter 4

Original Contributions

This chapter outlines our contribution of an original algorithm for binary decision diagram based bounded model checking. We include discussion of how such algorithm could be implemented within an existing model checker. We look at the model checker for multi-agent systems – MCMAS.

Section 4.1 lays out an overall view of our approach, as well as the devised algorithm. We also include a discussion of certain "variations on a theme", displaying an element of flexibility within the method. In the final part of this section we look at implementing this method into MCMAS.

In Section 4.2 we present a method for $\text{SAT}_{\text{ECTLK}}$, which supports the $\overline{\text{K}}$ operator. This is a *fundamental* requirement to our methods in the preceding section, when verifying *epistemic* logic in a bounded context. We also include a BDD based implementation of $\text{SAT}_{\overline{\text{K}}}$ for MCMAS.

Section 4.3 covers an extension to the first method, further displaying the flexibility of the approach. We show how the method can be distributed, requiring only limited changes to both the method and an implementation of that method.

Finally, in Section 4.4, we conclude by setting forward a model which could be used to show the possible benefits, and limitations, of each of these methods.

4.1 BDD based BMC

Our main algorithm (Algorithm 13) performs an incremental state space generation, including a check at each "depth". We continue this process until either we find a counterexample to the original formula, or we reach a *fixed point* in the state space.

4.1.1 BDD based BMC with "early termination"

Algorithm 13 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 both Cabodi's approach (of Algorithm 12) and Copty's approach (of Algorithm 11), 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 (Algorithm 12) or the frontier set (Algorithm 11) of

states with a target error state. In comparison, the algorithm we set forward here performs a full satisfiability check (as per Section 3.3.3) on the whole state space.

It can be seen that our approach is more flexible, and expressive, given that it is possible to express more properties than a single error state can describe. A striking difference between our algorithm and Copty's is that our algorithm keeps a BDD based representation of the entire reachable state space (the variable Reach in the above), whilst Copty's implementation only keeps the current frontier set of states.

The Algorithm (13) presented has two "exit" points: lines 5 and 12. The first of which is the case that the algorithm has found a counterexample to ψ – this is what we refer to as "early termination". This is due to the fact that, as soon as we find the counterexample to the ACTLK formula, we are able to return early (i.e. terminate the algorithm) and no longer have to continue building a reachable state space. The second exit point (line 12) is only accessible if we **break** the main loop body (line 9). The terminating condition for the loop is that we have reached 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 (we are adding nothing "new" to the Reach set).

To calculate the next Reach set (line 9) we initially generate only the set of "next" states. These are the states which are reachable one step away from the current reachable states set (i.e. with one application of the transition relation). We can construct these states from the conjunction of transition relation and the current reach set. The disjunction of these "next" states with the current reachable states results in the next set of reachable states (at a BMC depth of $k + 1$, with k applications of the transition relation).

In the actual implementation, the set of next states is stored in a temporary variable (at function, not loop, scope), allowing us to easily determine when a fixed point has been reached. If the previous, and the current, set of next states are the same, then the algorithm can no longer find any new reachable states (the next set is a subset of the reachable states), and, as such, fixed point has been reached.

4.1.2 Variations on BDD-BMC

Checking the satisfiability of $[\![\iota \rightarrow \varphi]\!]$ at each successive depth is not a "free" operation. This calculation may consume additional memory resources above and beyond the cost of only building the reachable states. This is an unwanted overhead when compared to performing non-bounded[1] BDD based model checking.

Also, the calculation of the set of satisfiable states is not an instantaneous process – this results in a time penalty at each depth. Again, this is an overhead not exhibited by non-bounded model checking.

In an attempt to alleviate the space/time penalty of checking at each successive bound, *heuristics* can be used to decide if a check should be performed at a specific bound. This procedure is highlighted in Algorithm 14.

It can be seen on Line 4 that we only perform the satisfiability check when the heuristic is satisfied.

Examples of possible heuristics for selecting when to check for satisfaction include:

- Different size increments – our original implementation did a check at *every* bound; alternatively it could be performed after a certain number of iterations (e.g. at every 10^{th} depth i.e $k\%10 == 0$).
- "One shot" – see Section 5.5
- Number of states – the satisfaction check could be guarded on the number of reachable (or next) states exceeding a certain threshold
- Memory used – similar to above, except the guard is on the memory used to hold the reachable state
- Time consumed – rather than checking on a depth bound, the check could be performed after a variable unit of time

[1] When we refer to non-bounded model checking we refer to MCMAS' default behaviour of building up the state space until a fixed point is reached. Only once a fixed point is reached is a single satisfiability check performed. This is not the same as unbounded model checking [29]

Algorithm 14 Heuristic-BDD-BMC(ψ : ACTLK Formula, $\mathcal{I}$: Initial State, Trans : Transition Relation, Heuristic) : String × Boolean

```
1: φ ← ¬ψ {φ : ECLTK Formula}
2: Reach ← I {Reach : BDD}
3: while True do
4:   if Heuristic met then
5:     if [[ι → φ]] = Reach then
6:       return Heuristic met × False
7:     end if
8:   end if
9:   Reach ← Reach ∨ (Reach ∧ Trans)
10:  if Reach Unchanged then
11:    break
12:  end if
13: end while
14: return Fixed point × [[ι → ψ]] = Reach
```

"One shot" BMC

The second method of BMC we have implemented is BMC with a "one shot" heuristic (Algorithm 15). The crux of this approach is, rather than checking $[\![\iota \rightarrow \varphi]\!]$ against the current reachable states at every depth, we build up the reachable states to the given "one shot" depth – and only then do we compare $[\![\iota \rightarrow \varphi]\!]$ against the reachable states. If the model satisfies φ at the one shot depth the algorithm returns false (φ is the counterexample). If we are unable to satisfy the negation we return true, although this could possibly be an incomplete result, as we have not built up all of the states.

Algorithm 15 "One shot" BMC(ψ : ACTLK Formula, $\mathcal{I}$: Initial State, Trans : Transition Relation, OneShotBound : int) : String × Boolean

```
1: φ ← ¬ψ {φ : ECLTK Formula}
2: Reach ← I {Reach : BDD}
3: for k ← 0 to OneShotBound do
4:   Reach ← Reach ∨ (Reach ∧ Trans)
5:   if Reach Unchanged then
6:     return Fixed point case : [[ι → ψ]] = Reach
7:   end if
8: end for
9: return One shot case : [[ι → φ]] = Reach
```

The method employed in Algorithm 15 differs significantly from that of Algorithm 13. The original algorithm will only terminate once a complete result has been found (either from reaching a fixed point or from finding a counterexample). The "one shot" BMC , as presented here, *may* return an incomplete (and useless) result. As discussed previously, the motivation for this approach is that the calculation of $[\![\iota \rightarrow \varphi]\!]$ at each depth is not a "free" operation; it can affect the amount of memory used by the model checker. (In MCMAS' case, this is the size of CUDD's cache). This results in a memory increase which is not displayed in "regular" verification. This algorithm avoids this overhead by only calculating the satisfiability set once, and then exiting.

The implementation of this algorithm is not supposed to be "used" directly by a user, due to the possibility of returning a false positive. Instead, we have developed a wrapper script to MCMAS which attempts an iterative depth approach to verification of a given model. When one instance of MCMAS finishes we regain all of the memory used by that instance. Then, when a new instance of MCMAS is launched, we start with a

"fresh" CUDD cache. This implementation is intended to overcome the memory overhead of our first method, but does this at the expense of time. Each time we start with a deeper one shot bound we have to effectively recalculate the set of reachable states, which may have already been calculated by the previous instance.

4.1.3 An Implementation

The remainder of this section discusses an *actual* implementation of our algorithm into MCMAS. The developed code can be found in Appexdix A.

In the following we attempt to outline the *significant* additions to the MCMAS code base to implement bounded model checking:

- **New Types** (**Figure** A.1) and **New Globals** (**Figure** A.2) – For efficiency, rather than calculating an ECTLK formula from a ACTLK each time, we keep a pair of both types of formulae, which allows us to easily change between the two.
- **Conversion of ACTLK to ECTLK** (**Figure** A.3) – We convert from an ACTLK formula to an ECTLK one by construction of the negation of the ACTLK formula and then "pushing" the negations through. We extended the `push_negations` function to support re-writing a negated K modality to that of a $\overline{\mathrm{K}}$ modality.
- **Checking the ECTLK formulae** (**Figure** A.4) – The original MCMAS implementation iterated over the `is_formulae` vector. In an attempt to reduce the effect our code had on the original implementation, we added a function which iterated over the `bmc_formulae` vector.
- **Implementing Algorithm 13: Part 1** (**Figure** A.5) – In this Figure we outline our implementation of Lines 1 – 11 from the original algorithm. The main difference between the algorithm and the implementation is that, rather than a "while true", our loop is guarded on still having formulae to check.
- **Implementing Algorithm 13: Part 2** (**Figure** A.5) – Our implementation terminates with Line 12 from the algorithm. We print out all of the formulae for which we have found counterexamples; any remaining formulae are checked with the original `check_formulae` function.

4.2 $\text{SAT}_{\overline{K}}$

The heart of bounded model checking lies in being able to satisfy an existential formula, without the requirement of having a representation of the entire state space which calculating the satisfaction of a universal formula would require. Model checking the existential fragment of CTLK (ECTLK) can follow the same procedures as for model checking ECTL (Algorithm 2), but with the addition of checking the dual of K; $\overline{K}$.

One possible way of calculating the satisfiability of a $\overline{K}_{\text{AGENT}}(\varphi)$, would be to evaluate $\neg K_{\text{AGENT}}(\neg\varphi)$ using Algorithm 8. Although this would be a feasible strategy it has the disadvantage that it pays an overhead to perform two negations, as well as being a rather *inelegant* solution.

A preferable way of calculating this procedure would be to provide, and implement, a direct method for $\text{SAT}_{\overline{K}}$ which is not dependent upon SAT_K.

Conceptually, our method for satisfaction of formulae of the form $\overline{K}_{\text{AGENT}}(\varphi)$ can be seen in algorithm 16. To find the set of states in which the previous formula holds, we first calculate the set of states in which φ holds. Next, we utilise the relation pre_K for the given agent, which returns the set of *all* global states in which the local state for the agent is invariant ($\sim_i$, as per Algorithm 8).

Algorithm 16 $\text{SAT}_{\overline{K}}(\varphi : \text{FORMULA}, i : \text{AGENT}) : \textit{set of}\ \text{STATE}$

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

4.2.1 BDD based $\text{SAT}_{\overline{K}}$

The first stage towards making a BDD based $\text{SAT}_{\overline{K}}$ is to be able to easily locate, and represent, the *reachable* states for which the local states are invariant for a given agent. The process for calculating these states can be seen in figure 4.1, showing a *simplified*[2] `project_local_state` method.

```
BDD basic_agent::project_local_state(BDD *state, BDDvector* v)
{
    BDD tmp = bddmgr->bddOne();

    // For all of the state variables before the agent ...
    for (int j = 0; j < get_var_index_start(); j++)
    {
        // ''and'' them on
        tmp = tmp * (*v)[j];
    }

    // and after the agent ...
    for (int j = get_var_index_end() + 1; j < v->count(); j++)
    {
        // ''and'' them on
        tmp = tmp * (*v)[j];
    }

    return state->ExistAbstract(tmp);
}
```

Figure 4.1: The *simplified* `project_local_state` method

[2]The actual implementation does not differ much, but it also has to quantify over the set of *global observable* variables and, as these have not been discussed previously, they have been omitted for clarity

This method is in the `basic_agent` class, which is MCMAS's lowest encapsulation of agent. The two methods `get_var_index_start` and `get_var_index_end` return the first and last index, respectively, into the vector of states (`v`) for that agent. A temporary BDD `temp` is constructed from the conjunction of *all* the other state variables for *all* the other agents in the system. Finally, the BDD representing a quantification of these states is constructed and returned.

The BDD returned from `project_local_state` is, in essence, a BDD representing *only* the local states for the agent, with all of the other states in `v` being set to "don't cares"[3].

```
BDD get_nK_states(BDD *state, string name)
{
    // Look up the agent from its name
    basic_agent *agent = (*is_agents)[name];

    // Project the local state over [[phi]]
    BDD localstate = agent->project_local_state(state, v);

    // ''and'' that state over the reachable states
    return reach * localstate;
}
```

Figure 4.2: The global function `get_nK_states`

The function in figure 4.2 builds upon the previous method – it takes the set of the states, $[\![\varphi]\!]$ (the first argument, `state`), and a unique ID for the agent (`string agent`). Once a reference to the agent is found from `is_agents`, the BDD representing the local state for that agent is constructed, which is subsequently ANDed with the set of reachable states. The resulting BDD represents the set of *all reachable states in which the local state is indistinguishable from a local state in* $[\![\varphi]\!]$ *for the given agent*.

Figure 4.3 displays our additions to a skeleton `modal_formula::check_formula()`. The integer constant `50` (stored in `op`, Section 3.3.2) is the formula identifier which MCMAS uses to represent formula of the kind $\overline{\mathrm{K}}_{\mathrm{AGENT}}(\varphi)$.

The "arguments" to the formula $\overline{\mathrm{K}}$, AGENT and φ are stored in the `obj` local variable. `obj[0]` is a pointer to agent class for AGENT, whilst `obj[1]` is a pointer to the modal formula representing φ.

[3] In the spirit of Karnaugh maps

```
BDD modal_formula::check_formula()
{
    // Returns a BDD encoding the set of states
    // in which the current formula is true

    BDD result, af;
    string name;
    Object *id;

    switch (op)
    {
    /*************************************************************************
     * SNIP
     * ************************************************************************/
        case 50: // KB
        {
            // id is the identity of an agent
            id = ((modal_formula*) obj[0])->get_operand(0);

            // Name is the name of agent
            name = ((atomic_proposition *) id)->get_proposition();

            // af is the set of reachable states in which the formula holds
            af = ((modal_formula*) obj[1])->check_formula();

            // result is the reachable states which are
            // indistinguishable for agent name
            result = get_nK_states(&af, name);
            break;
        }
     /*************************************************************************
     * SNIP
     * ************************************************************************/
    }

    return result;
}
```

Figure 4.3: A *reduced* check_formula method

4.3 Distributed Verification of ACTLK

This section presents an extension to our algorithm in Section 4.1 in which we demonstrate how the technique can be distributed in an attempt to utilise the available resources in a disjoint memory architecture. Our work builds on the ideas of Iyer et al in [25] but, rather that just hunting for bugs, we can also show correctness of the system under verification. We display a technique of using a Java based "wrapper" for MCMAS that enables the model checker to work in a networked fashion, using multiple hosts in a grid to both reduce the time taken and the memory used for verification.

4.3.1 The key idea of grid based BDD-BMC

Our approach uses a method of state space partitioning to allow for multiple "hosts" to perform bounded model checking on different areas of the state space. The main consideration for this approach is in checking *invariant* properties, expressing that a given condition must hold at every state in the reachable set.

Invariant properties in CTLK contain **AG** as the top most connective in the parse tree. The simplistic parse tree, for the formula $\mathbf{AG}(\varphi)$, can be seen in Figure 4.4. This formula will be satisfied from the initial state if, at every reachable state, φ holds.

Figure 4.4: The parse tree for $\mathbf{AG}(\varphi)$

Our method places an extra restriction upon the logic ACTLK, stating that **AG** is at the top – we refer to this as $\mathrm{A^GCTLK}$. The construction of the formula beneath this connective remains unaffected (we allow regular ACTLK formula). The falsification of a formula can be displayed by finding a *single reachable state* (from the initial state) in which the property beneath **AG** is no longer satisfied. We have previously shown that NuSMV supports "on the fly" checking of invariant properties in a very similar way (Section 2.3.8).

In a Kripke model, the transition relation is *transitive*; we deduce here that the reachability between two states is also transitive (if a state s_1 is reachable from s_0, and s_2 from s_1, then s_2 is reachable from s_0).

We translate the problem of falsifying $\mathbf{AG}(\varphi)$ in the initial state, from finding a single reachable state in which $\neg\varphi$ holds, to finding a reachable state in which $\mathbf{AG}(\varphi)$ does not hold. That is, we find a reachable state in which $\mathbf{EF}(\neg\varphi)$ holds.

We use Iyer's terminology of "seed states" to represent the different reachable states in which we attempt to show the falsifiability of $\mathbf{AG}(\varphi)$.

4.3.2 Outline of grid based BDD-BMC

In the design approach we have taken for this algorithm we have two types of hosts: a single master and multiple slaves. The single master instance performs initial verification and seed generation. After this, the node acts solely as a "co-ordination host" between the multiple slaves.

Our method works in three stages:

Fixed Depth BMC – Initially, we perform a heuristics based approach to BMC – we check the satisfiability of $[\![\iota \rightarrow \mathbf{EF}(\neg\varphi)]\!]$ at every depth, attempting to find a counterexample, until a predetermined depth.

Generate Seed States – Once the given depth has been exceeded, each unique state in the frontier set of next states is recorded in a way which makes them amenable to either network transfer or to storage to a backing device.

Parallel Seed BMC – Parallel BMC solvers search for a counterexample (Algorithm 13), each starting with a different frontier state.

The first two stages in the above are only performed on the master instance, after which all of the computational work load is offloaded to the slaves.

Although, as we have presented it here, the master only performs BMC up to a specific depth, our method allows us to be much more flexible in our approach. Realistically, we can use *any* heuristic – for instance, seeds can be generated when:

- The memory used to represent the reach set becomes too large (e.g. it reaches the maximum memory of the machine, which would mean that the swapping of the program's memory to disc would soon occur)
- The number of states in the "next" set exceeds a threshold (e.g. the state threshold could be related to the number of nodes in the grid)
- The time taken to calculate $[\![\iota \rightarrow \mathbf{EF}(\neg\varphi)]\!]$ on the current reach exceeds a certain value

4.3.3 Uniqueness of the Approach

The uniqueness of our approach hinges on attempting falsification in a concurrent way between multiple hosts. Once the initial phase has terminated, and the seed states have been generated (fig. 4.5), the master instance can then disseminate each of the seed states between the nodes. If there are more seed states than slaves, the master instance can re-allocate a slave with a new seed state once it has completed its original processing. The slaves all use Algorithm 13, which proceeds until either a counterexample is reached or a fixed point is established. This means that the slaves can return two results: *false* or *maybe true*. Algorithm 13 takes an initial state as an argument here; where this algorithm is utilised in a slave, the initial state is the seed state allocated to that slave. The final line of the algorithm Line 12 is replaced with $[\![\iota \rightarrow \varphi]\!] \neq \mathit{Reach}$ such that, when a seed state has zero successors, the slave returns maybe true.

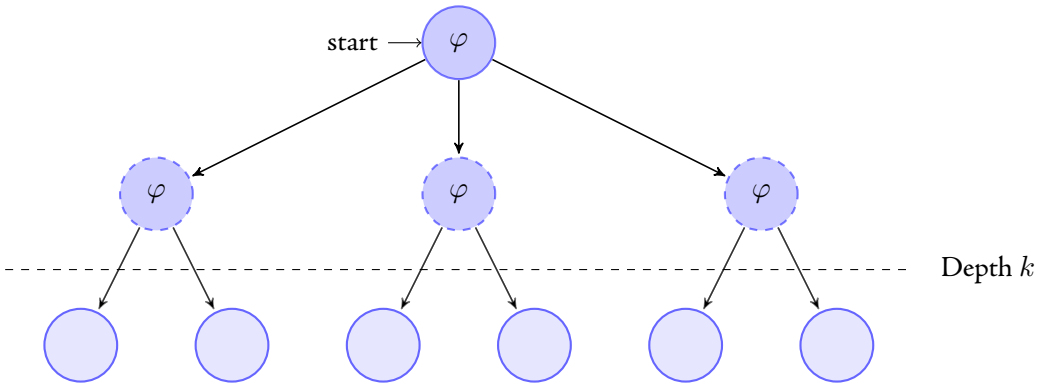

Figure 4.5: The initial state of seeded BMC, up to a seed depth of k ($k = 1$). "Dashed" states represent the set of seed states.

A return of false from a single slave means that a counterexample has been found from the seed state to which it was allocated. As previously stated, not only are the seed states reachable from the initial state (by construction), but the state in the seed tree which invalidated the property is also reachable (represented by "Seed 3" in fig. 4.6). When a slave instance informs the master that a counterexample has been found, the master can then "kill off" the other slave instances, reducing the over-processing. It is then possible for the master instance to return a definite value of false to the user. Unlike regular BMC, which finds the minimum counterexample, seeded BMC may not find the most optimal counterexample, given that the execution trace found in the node *could* be longer.

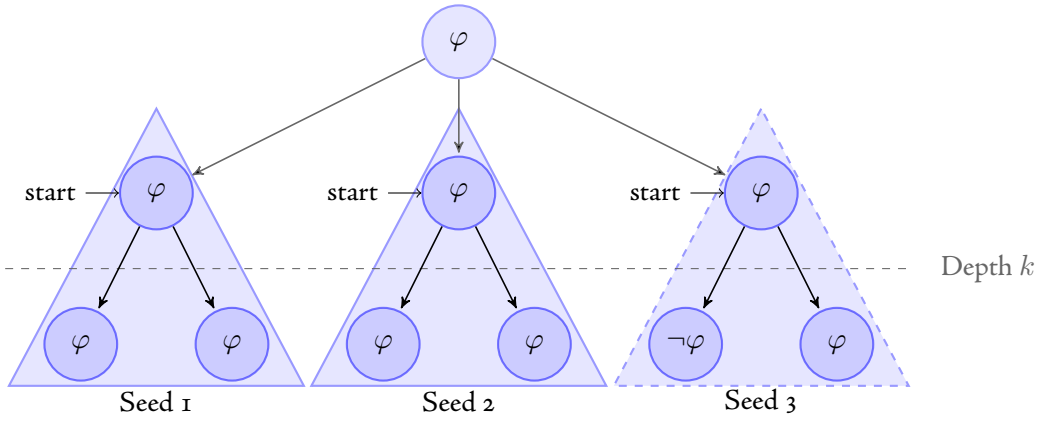

Figure 4.6: Seeded BMC in which one of the slave instances (starting from Seed 3) can falsify the property

In contrast, a slave returning *maybe true* means that the slave cannot falsify the property from its given seed state. This is not a *complete* indication of the actual result of the verification process. It is possible that the selected seed for that slave may not have a reachable state which violates the invariant property (it may not have *any* reachable states).

The flip side to this is if *all* results from the seeds return this value (fig. 4.7). From this we can see that, from all of the seed states, the property cannot be falsified. This, in conjunction with the initial check by the master instance performed (up to the generation depth), ensures that the invariant is satisfied on the whole model. From this we are able to deduce that, from all reachable states from the initial state, a violation of the invariant cannot be found. As such, the verifier can return a definite answer of true to the user.

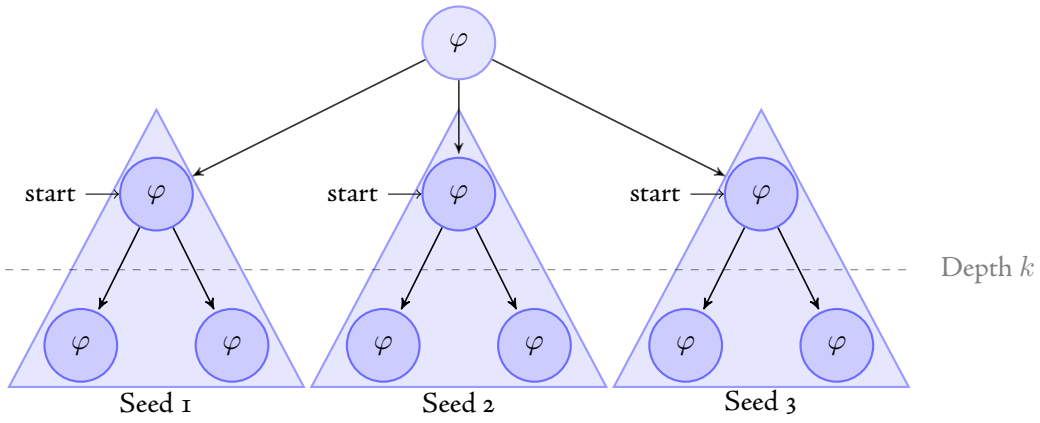

Figure 4.7: Seed BMC in which *all* of the slaves return *maybe true*

4.3.4 Distributing MCMAS

We have developed a *framework* which can be used as an extension to MCMAS, allowing for easy distribution. Our framework is not directly tied to the model checker – it acts as a wrapper to the model checker, allowing for communication between the master and slave instances. Our approach can be utilised across different model checkers, providing that they return output in a specific format.

We use a networked file system which is common to all of the nodes of the system. This allows for easy transfer of data (the model and the initial states) between different hosts.

MCMAS was extended to take some parameters, the first of which is a flag specifying if it should be launched as either a master or a slave. If we are launching a master instance we provide it with a directory which it uses to store the seeds, as well as the depth to which it should perform BMC until the seed states are generated. The only extra argument the slave takes is a file which it should use as the initial seed state.

It can be seen that this method requires the model checker to be able to save, and retrieve, states from disk. The DDDMP Library [79], which works with CUDD, defines file formats which can be used for the storage of BDDs. We use this library in both the master and the slave. The slaves use Algorithm 13, which already takes an initial state from which it should start the state space traversal – conventionally, MCMAS would pass the initial state specified in the model. We now pass the seed state as the initial state, as specified in the arguments to MCMAS (Figure 4.8). The master iterates over all of the states in the current frontier set, writing each of them to the given directory, and then exits. This is highlighted in Figure 4.9, which generates the seeds when BMC reaches the specified depth.

```
// DDDMP specifics
Dddmp_VarMatchType varmatchmode = DDDMP_VAR_MATCHIDS;
Dddmp_VarInfoType varoutinfo = DDDMP_VARIDS;

// Parse the file name
char filename[100];
strcpy(filename, initbddfile.c_str());

// Load the seed state
DdNode* b;
b = Dddmp_cuddBddLoad(bddmgr->getManager(), varmatchmode, NULL,
        NULL, NULL, DDDMP_MODE_DEFAULT, filename, NULL);

// Construct a BDD representing this state
BDD temp(bddmgr, b);

// Assign the state to both initial states
// and the initial reach states
in_st = temp;
reach = temp;

// BMC continues as previously ...
```

Figure 4.8: The internals of the "slave" instance to load the given seed state

Our Java wrapper contains `Slave` and `Master` classes, each of which provide an interface to the two types of MCMAS instances. The master acts as coordinator to the whole process – performing the initial BMC and assigning states in an iterative manner to each of the slaves, until a desired result is reached.

The slave instances connect to a given master and await information pertaining to the seed state, which they should use. After a slave has completed BMC from the given slave it conveys the result of the verification, including statistics such as memory use, back to the master.

When a slave indicates to the master that it has found a counterexample for its seed state, the master then "terminates" all of the other slaves, causing the verification of the other seed states to halt. We implement this in a basic way by closing all of the sockets used by the slaves to communicated with the master. Exception handling for a closed socket is used on the slave, terminating the execution of MCMAS and exiting the Java process when this exception is detected.

A sequence diagram displaying the whole process can be seen in Figure 4.10.

```
// k is the current BMC depth the algorithm has explored to
// If we reach the seed depth, and still have formulae to falsify
if (k == seed_depth && !bmc_formulae->empty())
{
    // DDDMP specifics
    Dddmp_VarMatchType varmatchmode = DDDMP_VAR_MATCHIDS;
    Dddmp_VarInfoType varoutinfo = DDDMP_VARIDS;
    int status = 0;

    // Identifier for each state
    int state_iter = 0;

    // While we still have next states
    while (nextstates != bddmgr->bddZero())
    {
        // Pick a random node, which is the state in v,
        // from the set of next states
        BDD singlenextstate = nextstates.PickOneMinterm(*v);

        // If the state we've chosen _isn't_ valid, then skip over it
        if (!is_valid_state(singlenextstate, *v) continue;

        // Otherwise
        // Set up the file name
        sprintf(filename, "%s/state_%04d.out", working_directory,
                    state_iter);

        // Get the decision diagram representing that node
        DdNode* a = singlenextstate.getNode();

        // Write the it out to a file
        status = Dddmp_cuddBddStore(bddmgr->getManager(), NULL, a, NULL,
                    NULL, DDDMP_MODE_TEXT, varoutinfo, filename, NULL);

        // Check if we were unable to write the node out
        if (status != 1)
        {
            cout << "DDDMP failed: ";
            cout << "we were unable to write the node to a file";
            cout.flush();
            exit(-1);
        }

        // Remove the current state from the set of next states (set minus)
        nextstates -= singlenextstate;

        // Increment our state identifier
        ++state_iter;
    }
}
```

Figure 4.9: The internals of the "master" instance to save the next set of states to a file

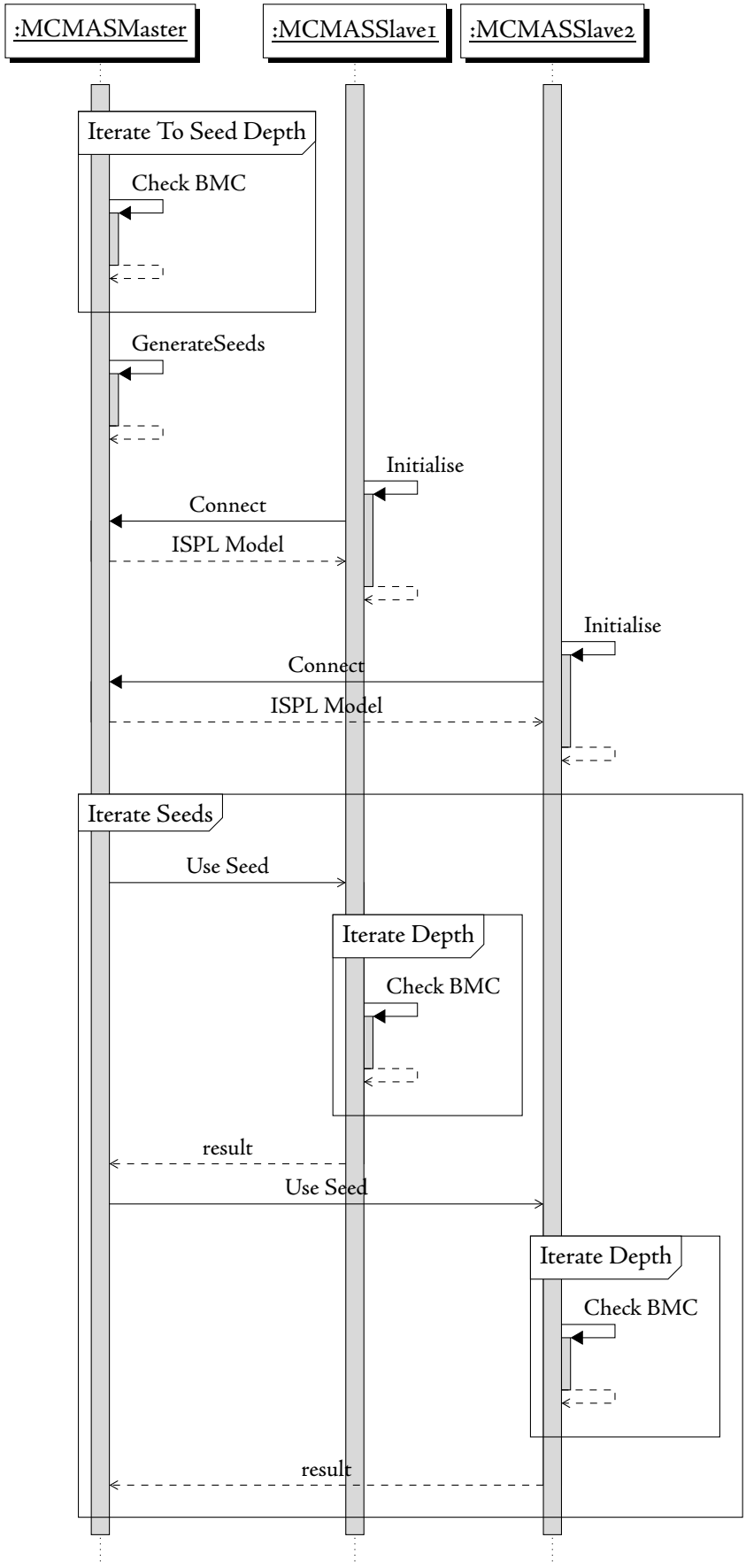

Figure 4.10: Sequence diagram for the distributed bounded model checking of CTLK

4.3.5 Consideration of other connectives

Although we originally placed a restriction of ACTLK such that **AG** was the top level connective, it is possible to extend our scheme to also support **AF**.

The immediate problem of a seed based approach to bounded model checking, with respect to an **AF** formula, is that it is possible to find a counterexample to the original formula from a seed state – even though the formula is true from the initial state.

For example, if we take a model in which, from the initial state, $\mathbf{AX}(\varphi)$ holds – then, from the same initial state, $\mathbf{AF}(\varphi)$ holds (i.e. $\varphi \in \mathcal{L}(s) \Leftrightarrow \mathcal{R}(\iota, s)$).

Attempting to falsify the original property by checking the satisfaction of $\mathbf{EG}(\neg\varphi)$, as per the initial stage of our distributed method for this initial state, will not succeed. When BMC cannot satisfy an ECTLK formula, the algorithm continues under the assumption that it has not yet considered enough states to find a counterexample. But, in this situation, $\mathbf{AF}(\varphi)$ is true, so will continue until the full state space has been explored and a fixed point is reached.

Prior to reaching the fixed point, the distributed version of BMC will reach the depth at which the seed states should be generated, and seeded BMC will begin. Attempting to find a counterexample with $\mathbf{EG}(\neg\varphi)$ will now be true at every seed state (because transition relation is serial), resulting in an incorrect result. (Our algorithm will assume that the counterexample would also be valid from the initial state and return false).

This situation can be avoided by modifying the initial bounded check, as performed by the master. A final check, prior to generating the seed states, can be performed to check the satisfaction of $\mathbf{AF}(\varphi)$ and, if it is satisfied on the current truncated model, returns true to the user and does not continue with seeded BMC.

If we are unable to satisfy this extra check, we generate seeds such that they are the final state in a path through the model along which φ is never satisfied. If, from such seed, we are able to find a path witnessing $\mathbf{EG}(\neg\varphi)$ (representing a k-loop), then a path starting from the original initial state, which passes through this state and includes the loop, is an *infinite* path upon which φ is never satisfied. This path is then a counterexample to the original formula which exists in the full model.

4.4 A scalable model

To allow us to effectively investigate the efficiency of our BMC implementation we required not only a *scalable* model. The model also required the existence of *meaningful* and *expressible* properties which could be falsified at a *variable* depth but, under certain parameters to the model, could equally be true.

The resulting model is one which builds on Kacprazak's *parameterised* model of [29] allowing for a variable number of agents in the system. Inspiration for the design of the new model was also taken from [19], in as much as this work was the catalyst for designing a possibly *faulty* multi-agent system.

Our model is a combination of these two factors, in the context of the Train-Gate-Controller model (§2.5.1). Where it is necessary to make a distinction between the original model and our new model, our model shall be referred to as the *faulty* train-gate-controller.

4.4.1 The Faulty Controller

We scale the number of agents by allowing the controller to handle the merging of an arbitrary number of tracks into a single track in the tunnel. An automaton displaying the new behaviour can be seen in figure 4.11; the descriptions of the transitions, including assignments and required preconditions from this automaton, are shown in table 4.12.

Note to the reader: The protocol for an agent in a given local state can be inferred from the given preconditions for each possible action in that state.

In a faulty model with N trains the controller contains N c_5 edges (table 4.12), each of which is the transition of a different train entering the tunnel. This is represented in the table with the unquantified "**?**" variable.

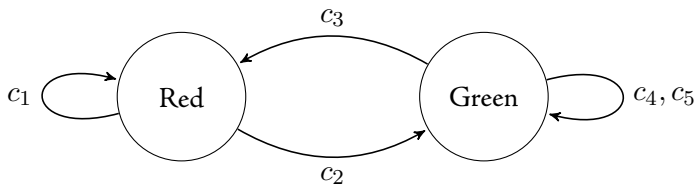

Figure 4.11: An automaton modelling the controller in the faulty Train-Gate-Controller model

Label	Action	Assignments	Preconditions
c_1	IDLE	`i++`	$i < 2$
c_2	EXIT_TRAIN		$i = 2$
c_3	IDLE	`train?_waiting := true`	$Train?.action =$ SIGNAL
c_4	IDLE		
c_5	ENTER_TRAIN?	`i := 0`	$Train?.action =$ ENTER
		`train?_waiting := false`	$train?\_waiting = true$

Figure 4.12: Descriptions of actions of the controller in the faulty Train-Gate-Controller model

In comparison to the original train-gate-controller, where both of the actions *enter* and *exit* were joint actions for the controller and a train, in our faulty model we only have the per-train joint action *enter*.

The controller has been extended to function under the assumption that trains take *at most* two synchronous system evolutions to leave the tunnel. This is represented by the guard on the action c_2, and the additional IDLE action c_1. This guard only enables the transition back to the green state when a counter, representing the number of evolutions since the first train entered the tunnel, has exceeded the threshold of two.

4.4.2 The Faulty Train

We have adapted the trains from the original model, such that they now contain a SERVICE and a BREAK action. The trains also contain an additional *service counter* representing the number of synchronous system evolutions they have performed since they were last serviced. The service action resets the service counter to zero, whilst the break action occurs when a fault occurs in the train. Faults occur in trains when they are not serviced regularly enough, i.e. the service counter exceeds a predefined threshold, and can only be exhibited when the train is in the tunnel. An automaton displaying the state transitions for this *faulty* train model can be seen in figure 4.13. The trains also contain a *max counter*, which, once the service counter reaches this value, forces a train to perform the service action[4].

We have modelled the faulty system with 3 *types* of trains:

- *Type 1* – Faulty – table 4.14 – Once the service counter exceeds the breaking threshold the trains can non-deterministically break in the tunnel. Once a train has broken in the tunnel it is unable to repair itself and is in the tunnel for the rest of the run of that system. If the service counter reaches the maximum counter the train can no longer non-deterministically choose the leave action and definitely breaks.

- *Type 2* – Faulty – table 4.15 – Same as *type 1*, with the exception that all the non-deterministic break action allows is for the train to get "stuck" in the tunnel for a single evolution. The train can break an infinite number of times (displaying the behaviour of always being in the tunnel), or can eventually perform the leave action.

[4]This can be seen as similar to "aircraft maintenance checks" [73], where a plane requires a set of a mandatory "checks" after a certain number of flight hours. This is still only a preventative measure and not a *guarantee* that no fault will occur before this.

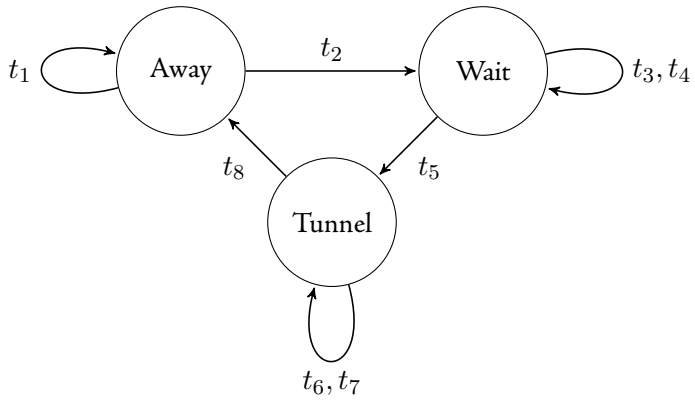

Figure 4.13: An automaton modelling a train from the faulty Train-Gate-Controller model

- *Type 3* – Correct – table 4.16 – Trains can never perform the break action; the only action that can be performed in the tunnel is the leave action. This is the same behaviour for the trains as in the original system [1].

The tables for the descriptions of the actions of the three types of faulty train allow for implicit requirements for some actions to take place. For example, we require that, once the service counter reaches the maximum counter that the train is serviced, this requirement is modelled by the subsumption of the preconditions of the transition t_2 by those of transition t_1 in table 4.14. The action BACK can only be performed when the service counter is strictly less than the maximum counter; this means that the only action which is allowed by the protocol from this state is the SERVICE action. A second example of this is the requirement that a train *must* break if it is in the tunnel and the service counter reaches the maximum counter – again, this can be seen in the subsumption of the preconditions of the transition t_8 by transition t_7 (table 4.14).

Label	Action	Assignments	Preconditions
t_1	SERVICE	`servicecount := 0`	
t_2	BACK	`servicecount++`	$servicecount < maxcounter$
t_3	SERVICE	`servicecount := 0`	
t_4	SIGNAL	`servicecount++`	$servicecount < maxcounter$
t_5	ENTER	`servicecount++`	$servicecount < maxcounter$ $Controller.action =$ ENTER_TRAIN? $? = this$
t_6	BREAK		$broken = true$
t_7	BREAK	`broken := true`	$broken = false$ $servicecount \geq threshold$
t_8	LEAVE	`servicecount++`	$broken = false$ $servicecount < maxcounter$

Figure 4.14: Descriptions of actions for a *type 1* faulty train

4.4.3 Specifications

To display the effectiveness (or possible *in*effectiveness) of our BMC implementation, we need to be able to specify properties upon the model which can be falsified in a model with faulty trains (i.e. trains of the type 1 or 2). These are properties which, if evaluated on a model with correct trains (i.e. type 3), should not be falsifiable

Label	Action	Assignments	Preconditions
t_6	*Removed*		
t_7	BREAK		$servicecount \geq threshold$
t_8	LEAVE	`servicecount++`	

Figure 4.15: Descriptions of actions for a *type 2* faulty train

Label	Action	Assignments	Preconditons
t_1	SERVICE	`servicecount := 0`	
t_2	BACK	`servicecount++`	$servicecount < maxcounter$
t_3	SERVICE	`servicecount := 0`	
t_4	SIGNAL	`servicecount++`	$servicecount < maxcounter$
t_5	ENTER	`servicecount++`	$servicecount < maxcounter$ $Controller.action =$ ENTER_TRAIN? $? = this$
t_6	*Removed*		
t_7	*Removed*		
t_8	LEAVE	`servicecount++`	

Figure 4.16: Descriptions of actions for a working train

(i.e. they should be satisfiable in the model).

It should be clear to the reader that a faulty controller, when used in a model with faulty trains, may allow extra trains to enter the tunnel even though there is still another train currently occupying it (e.g. A train of type 1 enters and subsequently breaks in the tunnel. Two synchronous evolutions occur - the controller then moves into the green state and allows another train to also enter).

We feel compelled to point out to the reader at this point that neither our faulty controller model, nor the bounded model checking which has yet to be presented, support any kind of notation. This means that the starvation property from [57] cannot be used as it would be false both in a model with faulty, and a model with correct, trains.

The following formulae are described in a model containing two trains, TRAIN_1 and TRAIN_2. The propositional atoms $\text{TRAIN}_1\text{\_IN\_TUNNEL}$ and $\text{TRAIN}_2\text{\_IN\_TUNNEL}$ hold iff the local state for agent equals the tunnel. The formulae φ_2 to φ_5 can be constructed pairwise with each unique pair of agents within the system.

Formula 1 ($\varphi_{\text{TGC}1}$) "There always exists a future state in which the train no longer occupies the tunnel"

$$\begin{aligned} \textbf{ACTLK} &= \textbf{AG}\,(\textbf{AF}\,(\neg \text{TRAIN}_1\text{\_IN\_TUNNEL})) \\ \textbf{ECTLK} &= \neg\textbf{EF}\,(\textbf{EG}\,(\text{TRAIN}_1\text{\_IN\_TUNNEL})) \\ \textbf{ISPL} &= \texttt{AG(AF(!train1\_in\_tunnel));} \end{aligned}$$

Formula 2 ($\varphi_{\text{TGC}2}$) **Mutual Exclusion**: "Two trains never occupy the tunnel at the same time"[5]

$$\begin{aligned} \textbf{ACTLK} &= \textbf{AG}\,(\neg \text{TRAIN}_1\text{\_IN\_TUNNEL} \vee \neg \text{TRAIN}_2\text{\_IN\_TUNNEL}) \\ \textbf{ECTLK} &= \neg\textbf{EF}\,(\text{TRAIN}_1\text{\_IN\_TUNNEL} \wedge \text{TRAIN}_2\text{\_IN\_TUNNEL}) \\ \textbf{ISPL} &= \texttt{AG((!train1\_in\_tunnel or !train2\_in\_tunnel))} \end{aligned}$$

[5]This formula could be written in a more intuitive way as, $\textbf{AG}\neg\,(\text{TRAIN}_1\text{\_IN\_TUNNEL} \wedge \text{TRAIN}_2\text{\_IN\_TUNNEL})$ [57]. We have used De Morgan's law to present a specification which adheres to the requirement that, in ACTLK, negation may only appear in front of atoms.

Formula 3 ($\varphi_{\text{TGC}3}$) "When a train is in the tunnel it knows that another train is not"[6]

$$\begin{aligned}
\textbf{ACTLK} &= \mathbf{AG}\left(\text{Train}_1\text{_in_tunnel} \rightarrow \mathbf{K}_{\text{Train}_1}\left(\neg\text{Train}_2\text{_in_tunnel}\right)\right)\\
\textbf{ECTLK} &= \neg\mathbf{EF}\left(\text{Train}_1\text{_in_tunnel} \wedge \overline{\mathbf{K}}_{\text{Train}_1}\left(\text{Train}_2\text{_in_tunnel}\right)\right)\\
\textbf{ISPL} &= \texttt{AG(train1_in_tunnel -> K(Train1, (!train2_in_tunnel)));}
\end{aligned}$$

Formula 4 ($\varphi_{\text{TGC}4}$) "Trains always know that they have exclusive use of the tunnel"

$$\begin{aligned}
\textbf{ACTLK} &= \mathbf{AG}\left(\mathbf{K}_{\text{Train}_1}\left(\neg\text{Train}_1\text{_in_tunnel} \vee \neg\text{Train}_2\text{_in_tunnel}\right)\right)\\
\textbf{ECTLK} &= \neg\mathbf{EF}\left(\overline{\mathbf{K}}_{\text{Train}_1}\left(\text{Train}_1\text{_in_tunnel} \wedge \text{Train}_2\text{_in_tunnel}\right)\right)\\
\textbf{ISPL} &= \texttt{AG(K(Train1, (!train1_in_tunnel or !train2_in_tunnel)));}
\end{aligned}$$

Formula 5 ($\varphi_{\text{TGC}5}$) "Trains are aware that there is a gap between leaving and the next train entering the tunnel"[7]

$$\begin{aligned}
\textbf{ACTLK} &= \mathbf{AG}\left(\text{Train}_1\text{_in_tunnel} \rightarrow \mathbf{K}_{\text{Train}_1}\left(\mathbf{AX}\left(\neg\text{Train}_2\text{_in_tunnel}\right)\right)\right)\\
\textbf{ECTLK} &= \neg\mathbf{EF}\left(\text{Train}_1\text{_in_tunnel} \wedge \overline{\mathbf{K}}_{\text{Train}_1}\left(\mathbf{EX}\left(\text{Train}_2\text{_in_tunnel}\right)\right)\right)\\
\textbf{ISPL} &= \texttt{AG(train1_in_tunnel -> K(Train1, AX(!train2_in_tunnel)));}
\end{aligned}$$

Parameterised The formulae $\varphi_{\text{TGC}3}$ and $\varphi_{\text{TGC}5}$ can be parameterised in a similar way to [29], for a system composed of N trains:

"When a train is in the tunnel, it knows that no other train in the whole system is in the tunnel"

$$\varphi_{\text{TGC}3}(\text{N}) = \mathbf{AG}\left(\text{Train}_i\text{\_in\_tunnel} \rightarrow \mathbf{K}_{\text{Train}_i}\left(\bigwedge_{j=1}^{i-1} \neg\text{Train}_j\text{\_in\_tunnel} \wedge \bigwedge_{j=i+1}^{N} \neg\text{Train}_j\text{\_in\_tunnel}\right)\right)$$

"When a train is in the tunnel it knows that no other train in the whole system will enter the tunnel in the next evolution"

$$\varphi_{\text{TGC}5}(\text{N}) = \mathbf{AG}\left(\text{Train}_i\text{\_in\_tunnel} \rightarrow \mathbf{K}_{\text{Train}_i}\left(\bigwedge_{j=1}^{i-1} \mathbf{AX}\left(\neg\text{Train}_j\text{\_in\_tunnel}\right) \wedge \bigwedge_{j=i+1}^{N} \mathbf{AX}\left(\neg\text{Train}_j\text{\_in\_tunnel}\right)\right)\right)$$

We have developed an "ISPL generator" for this model, allowing us to generate models of an arbitrary size containing a configurable number of any of the three types of trains, with a configurable breaking depth. The generator creates all of the formulae discussed here. In these models the controller is modelled as part of the environment.

Auto generated ISPL code for a controller (modelled by the environment) in a train-gate-controller model with 2 trains, a $maxcounter$ of 20 and $breakingdepth$ of 10 can be found in Figure 4.17. The generated ISPL for a *type 1* faulty train, in the same model, can be found in Figure 4.18.

[6] Similar to α_1 from [30].

[7] Or, "trains are aware that the controller allows for a gap between one train leaving and the next entering".

```
Agent Environment
    Vars:
        lights : { red, green };

        train1_waiting : boolean;
        train2_waiting : boolean;

        counter : 0..2;
    end Vars

    Actions = { enter1, enter2, idle };

    Protocol:
        lights = green and train1_waiting = true : { enter1 };
        lights = green and train2_waiting = true : { enter2 };
        Other: { idle };
    end Protocol

    Evolution:
        counter = counter + 1 if counter < 2 and lights = red;

        lights = green and counter = 0 if counter = 2 and lights = red;

        lights = red and train1_waiting = false if
            train1_waiting = true and lights = green and
            Action = enter1 and Train1.Action = enter;

        lights = red and train2_waiting = false if
            train2_waiting = true and
            lights = green and Action = enter2 and Train2.Action = enter;

        train1_waiting = true if Action = idle and Train1.Action = signal;

        train2_waiting = true if Action = idle and Train2.Action = signal;
    end Evolution

end Agent
```

Figure 4.17: An example "controller" environment

```
Agent Train1
    Vars:
        state : { wait, tunnel, away };
        serviced : 0..20;
        broken : boolean;
    end Vars

    Actions = { signal, enter, leave, back, service, break };

    Protocol:
        serviced = 20 and (state = away or state = wait) : { service };

        state = wait and serviced < 20 : { signal, service, enter };

        -- Trains work correctly if the non-deterministic
        -- break action is removed. Replace the following lines:
        state = tunnel and serviced < 5 and broken = false: { leave };

        state = tunnel and serviced >= 5 and
            serviced < 20 and broken = false : { leave, break };

        state = tunnel and broken = true : { break };

        state = tunnel and serviced = 20 and broken = false : { break };
        -- With:
        -- state = tunnel : { leave };

        state = away and serviced < 20 : { service, back };
    end Protocol

    Evolution:
        serviced = 0 if Action = service;

        state = tunnel if broken = true and state = tunnel;

        state = tunnel and broken = true if
            state = tunnel and Action = break;

        state = wait and serviced = serviced + 1 if
            serviced < 20 and state = wait and Action = signal;

        state = wait and serviced = serviced + 1 if
            serviced < 20 and state = away and Action = back;

        state = tunnel and serviced = serviced + 1 if
            serviced < 20 and state = wait and Action = enter
            and Environment.Action = enter1;

        state = away and serviced = serviced + 1 if
            serviced < 20 and state = tunnel and Action = leave;

        -- The only edge case is that the train cannot be
        -- serviced in the tunnel
        state = away and serviced = 20 if
            serviced = 20 and state = tunnel and Action = leave;
    end Evolution
end Agent
```


Figure 4.18: An example *type 1* faulty train

Chapter 5

Evaluation

5.1 Fixed Point Methods on Non-total Transition Relations

The core of any model checker is its implementation of the SAT_{CTLK} of Section 2.3.1. The sub-procedures of this are based on either a least, or greatest, fixed point calculation (i.e. the calculation continues until the result stabilises).

These methods rely on the total transition relation of a Kripke Structure ($\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, Section 2.1). That is, for all states $s \in \mathcal{S}$ there exists a state $s' \in \mathcal{S}$, such that $(s, s') \in \mathcal{R}$.

When performing BDD based bounded model checking using these procedures, the transition relation is *not* guaranteed to be serial. It is quite possible that, at our current depth, there may be states on the "fringe" (i.e. at the end of a path through the model) of the reachable states for which there does not exist a successor state.

In the following subsections, we demonstrate that these fixed point methods are, in fact, *correct*, even when used upon truncated paths through the model – they do not return "false positives".

5.1.1 SAT_{EX}

To show the falsification of a universal next formula ($\mathbf{AX}\varphi$), we attempt to check the satisfiability of its existential dual – $\mathbf{EX}\neg\varphi$. As noted in Section 3.3.3, the formula the implementation tries to satisfy is the following: $\text{INIT} \rightarrow \mathbf{EX}\neg\varphi$, that is: are the initial states for the model a subset of the states in which $\mathbf{EX}\neg\varphi$ holds.

Within MCMAS, checking an atomic proposition returns all of the *reachable* states in which the proposition holds. (This is performed through set intersection upon the set $[\![\varphi]\!]$ and `reach`, see Section 2.3.8). The pre-image calculation is based upon `vRT` (the per-agent transition relation vector); all calculations upon this set are also taken with conjunction (i.e. intersection) of the reachable states. We can perform set intersection of the reachable states, and the states returned by the pre-image function, such that we only get a set of states which is subsumed by the reachable set.

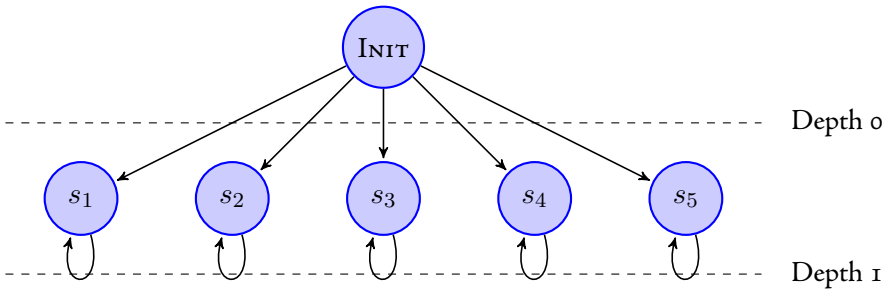

Figure 5.1: Checking $\mathbf{AX}(\varphi)$

In Figure 5.1, presume that $\mathcal{L}(\varphi) = \{s_1, s_2, s_3, s_4\}$ (i.e $\mathcal{L}(\neg\varphi) = \{\text{INIT}, s_5\}$). Initially, at a depth of 0, the `reach` set contains only the INIT state and φ does not hold at this state. As such $[\![\neg\varphi]\!]$ is $\{\text{INIT}\}$. The pre-image calculation ($\text{pre}_\exists(\{\text{INIT}\})$) returns the empty set ($\emptyset$), due to the restriction that the starting states of the transition relation *must* be in the reachable set of states, and that, in this model, there are no states which transition *into* INIT.

Therefore, $\mathbf{EX}\neg\varphi$ does not hold in the initial state, so the ECTLK formula is false and we do not have a counterexample to our original formula. We shall refer to the set of the state at which a particular formula (φ) holds at a particular bound, as: $[\![\varphi]\!]_{\text{BOUND}}$.

When we increment k to a depth of 1, giving us: $[\![\neg\varphi]\!]_1 = \{\text{INIT}, s_5\}$ the pre-image computation of this set gives us $\{\text{INIT}\}$. (Again, the pre-image of INIT is empty, whilst the pre-image of s_5 is INIT). So we have found a witness of the existential formula, meaning that we have a counterexample to the universal formula. As such, the BMC algorithm can terminate.

5.1.2 SAT_{EG}

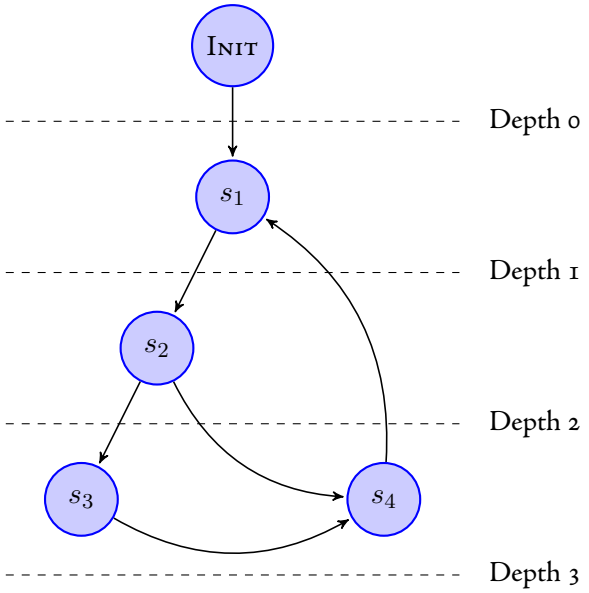

Figure 5.2: Checking $\mathbf{AF}(\varphi)$

$$
\begin{aligned}
\text{pre}_\exists(\text{INIT}) &= \{\} \\
\text{pre}_\exists(s_1) &= \{\text{INIT}, s_4\} \\
\text{pre}_\exists(s_2) &= \{s_1\} \\
\text{pre}_\exists(s_3) &= \{s_2\} \\
\text{pre}_\exists(s_4) &= \{s_3, s_2\}
\end{aligned}
$$

Figure 5.3: The labelling function and existential pre-image function for Figure 5.2

In Figure 5.2 (the state pre-image function can be seen in Figure 5.3) assume that $\mathcal{L}(\varphi) = \{s_3\}$ and, as such, $\mathcal{L}(\neg\varphi) = \{\text{INIT}, s_1, s_2, s_4\}$. Using $\text{SAT}_{\mathbf{EG}}$ from Section 2.3.1 (the reader is reminded that $\text{Y} \leftarrow \text{X} \cap \text{pre}_\exists(\text{Z})$), we can then calculate the fixed point for the satisfiability of an EG formula as follows:

Depth 0:

Iteration 0	Iteration 1
$\text{X} \leftarrow \{\text{INIT}\}$	$\text{X} \leftarrow \{\text{INIT}\}$
$\text{Z} \leftarrow \{\text{INIT}\}$	$\text{Z} \leftarrow \{\}$
$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
	$[\![\mathbf{EG}\neg\varphi]\!]_0 = \emptyset$

Depth 1:

Iteration 0	Iteration 1	Iteration 2
$\text{X} \leftarrow \{\text{INIT}, s_1\}$	$\text{X} \leftarrow \{\text{INIT}, s_1\}$	$\text{X} \leftarrow \{\text{INIT}, s_1\}$
$\text{Z} \leftarrow \{\text{INIT}, s_1\}$	$\text{Z} \leftarrow \{\text{INIT}\}$	$\text{Z} \leftarrow \{\}$
$\text{Y} \leftarrow \{\text{INIT}\}$	$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
		$[\![\mathbf{EG}\neg\varphi]\!]_1 = \emptyset$

Depth 2:

Iteration 0	Iteration 1	Iteration 2	Iteration 3
$\text{X} \leftarrow \{\text{INIT}, s_1, s_2\}$	$\text{X} \leftarrow \{\text{INIT}, s_1, s_2\}$	$\text{X} \leftarrow \{\text{INIT}, s_1, s_2\}$	$\text{X} \leftarrow \{\text{INIT}, s_1, s_2\}$
$\text{Z} \leftarrow \{\text{INIT}, s_1, s_2\}$	$\text{Z} \leftarrow \{\text{INIT}, s_1\}$	$\text{Z} \leftarrow \{\text{INIT}\}$	$\text{Z} \leftarrow \{\}$
$\text{Y} \leftarrow \{\text{INIT}, s_1\}$	$\text{Y} \leftarrow \{\text{INIT}\}$	$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
			$[\![\mathbf{EG}\neg\varphi]\!]_2 = \emptyset$

Depth 3:

Iteration 0	Iteration 1
$X \leftarrow \{\textsc{Init}, s_1, s_2, s_4\}$	$X \leftarrow \{\textsc{Init}, s_1, s_2, s_4\}$
$Z \leftarrow \{\textsc{Init}, s_1, s_2, s_4\}$	$Z \leftarrow \{\textsc{Init}, s_1, s_2, s_3, s_4\}$
$Y \leftarrow \{\textsc{Init}, s_1, s_2, s_3, s_4\}$	$Y \leftarrow \{\textsc{Init}, s_1, s_2, s_3, s_4\}$
	$[\![\mathbf{EG}\neg\varphi]\!]_3 = \{\textsc{Init}, s_1, s_2, s_3, s_4\}$

We can see that the calculation of $\textsc{Sat}_{\mathbf{EG}}$ only returns a result once the result is definitely true in the model; we do not get any kind of "false positive".

5.1.3 $\textsc{Sat}_{\mathbf{EF}}$

We can repeat the same for the calculation of a formula of the form **EF** – the satisfiability for an **EF** is the least fixed point for **EX**.

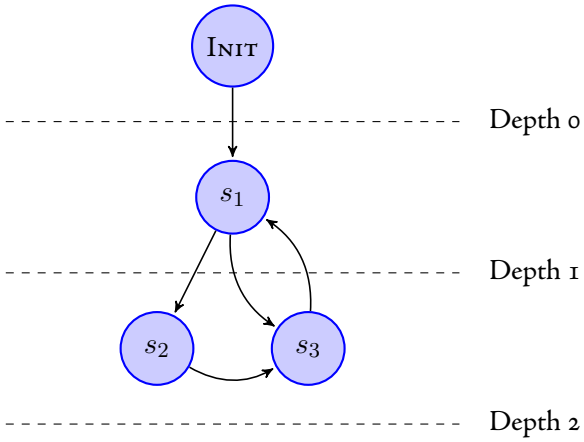

Figure 5.4: Checking $\mathbf{AG}(\varphi)$

$$
\begin{aligned}
\mathcal{L}(\varphi) &= \{s_1, s_3, \textsc{Init}\} \\
\mathcal{L}(\neg\varphi) &= \{s_2\} \\
\text{pre}_\exists(\textsc{Init}) &= \{\} \\
\text{pre}_\exists(s_1) &= \{\textsc{Init}, s_3\} \\
\text{pre}_\exists(s_2) &= \{s_1\} \\
\text{pre}_\exists(s_3) &= \{s_1, s_2\}
\end{aligned}
$$

Figure 5.5: The labelling function and existential pre-image function for Figure 5.4

Depth 0:

Iteration 0	Iteration 1
$X \leftarrow \{\}$	$X \leftarrow \{\}$
$Z \leftarrow \{\textsc{Init}\}$	$Z \leftarrow \{\}$
$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
	$[\![\mathbf{EF}\neg\varphi]\!]_0 = \emptyset$

Depth 1:

Iteration 0	Iteration 1	Iteration 2
$X \leftarrow \{\}$	$X \leftarrow \{\}$	$X \leftarrow \{\}$
$Z \leftarrow \{\textsc{Init}, s_1\}$	$Z \leftarrow \{\textsc{Init}\}$	$Z \leftarrow \{\}$
$Y \leftarrow \{\textsc{Init}\}$	$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
		$[\![\mathbf{EF}\neg\varphi]\!]_1 = \emptyset$

Depth 2:

Iteration 0
$X \leftarrow \{x_2\}$
$Z \leftarrow \{\textsc{Init}, s_1, s_2.s_3\}$
$Y \leftarrow \{\textsc{Init}, s_1, s_2.s_3\}$
$[\![\mathbf{EF}\neg\varphi]\!]_2 = \{\textsc{Init}, s_1, s_2.s_3\}$

Again, it can be seen that the fixed point methods work appropriately when dealing with models with truncated paths.

5.1.4 SAT_{EU}

To calculate the $[\![\mathbf{E}\,[\varphi \mathbf{U} \phi]]\!]$, W stores $[\![\varphi]\!]$, Y stores $[\![\psi]\!]$ and X the set of all reachable states.

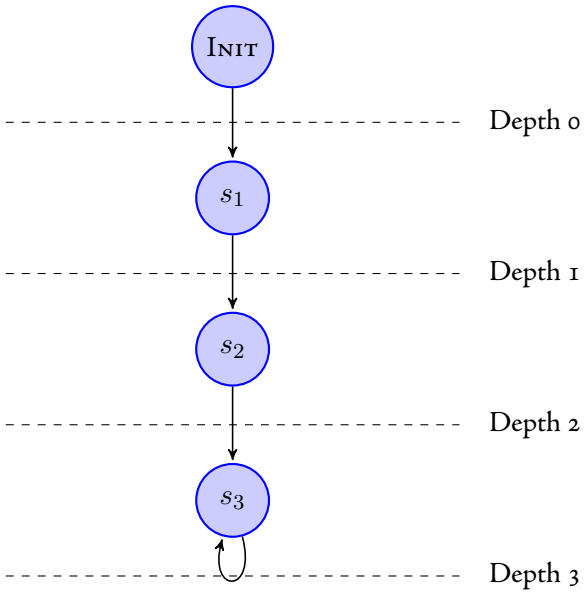

Figure 5.6: Checking $\mathbf{E}\,[\varphi \mathbf{U} \psi]$

$$
\begin{aligned}
\mathcal{L}(\varphi) &= \{\text{Init}, s_1, s_2\} \\
\mathcal{L}(\psi) &= \{s_3\} \\
\text{pre}_\exists(\text{Init}) &= \{\} \\
\text{pre}_\exists(s_1) &= \{\text{Init}\} \\
\text{pre}_\exists(s_2) &= \{s_1\} \\
\text{pre}_\exists(s_3) &= \{s_2, s_3\}
\end{aligned}
$$

Figure 5.7: The labelling function and existential pre-image function for Figure 5.6

Depth 0:

Iteration 0	Iteration 1
$W \leftarrow \{\text{Init}\}$	$W \leftarrow \{\text{Init}\}$
$X \leftarrow \{\text{Init}\}$	$X \leftarrow \{\}$
$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
	$[\![\mathbf{E}\,[\varphi \mathbf{U} \phi]]\!]_0 = \emptyset$

Depth 1:

Iteration 0	Iteration 1
$W \leftarrow \{\text{Init}, s_1\}$	$W \leftarrow \{\text{Init}, s_1\}$
$X \leftarrow \{\text{Init}, s_1\}$	$X \leftarrow \{\}$
$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
	$[\![\mathbf{E}\,[\varphi \mathbf{U} \phi]]\!]_1 = \emptyset$

Depth 2:

Iteration 0	Iteration 1
$W \leftarrow \{\text{Init}, s_1, s_2\}$	$W \leftarrow \{\text{Init}, s_1, s_2\}$
$X \leftarrow \{\text{Init}, s_1, s_2\}$	$X \leftarrow \{\}$
$Y \leftarrow \{\}$	$Y \leftarrow \{\}$
	$[\![\mathbf{E}\,[\varphi \mathbf{U} \phi]]\!]_2 = \emptyset$

Depth 3:

Iteration 0	Iteration 1	Iteration 2	Iteration 3
$W \leftarrow \{\text{Init}, s_1, s_2\}$	$W \leftarrow \{\text{Init}, s_1, s_2\}$	$W \leftarrow \{\text{Init}, s_1, s_2\}$	$W \leftarrow \{\text{Init}, s_1, s_2\}$
$X \leftarrow \{\text{Init}, s_1, s_2\}$	$X \leftarrow \{s_3\}$	$X \leftarrow \{s_2, s_3\}$	$X \leftarrow \{s_1, s_2, s_3\}$
$Y \leftarrow \{s_3\}$	$Y \leftarrow \{s_2, s_3\}$	$Y \leftarrow \{\text{Init}, s_1, s_2, s_3\}$	$Y \leftarrow \{\text{Init}, s_1, s_2, s_3\}$

Iteration 4
$W \leftarrow \{\text{Init}, s_1, s_2\}$
$X \leftarrow \{\text{Init}, s_1, s_2, s_3\}$
$Y \leftarrow \{\text{Init}, s_1, s_2, s_3\}$
$[\![\mathbf{E}\,[\varphi \mathbf{U} \phi]]\!]_3 = \{\text{Init}, s_1, s_2, s_3\}$

5.2 $\text{Sat}_{\overline{\text{K}}}$ on Truncated Paths

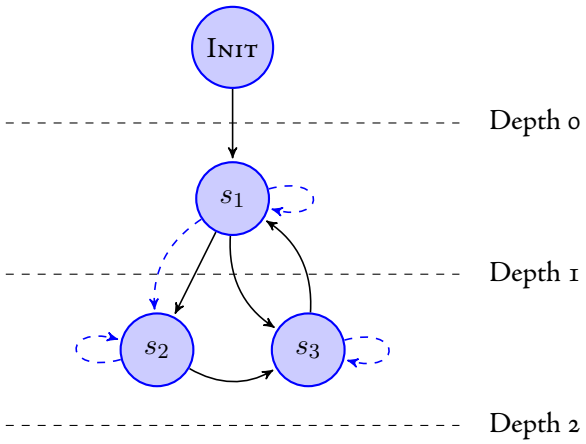

Figure 5.8: A model showing the local state equivalence relation

$$
\begin{aligned}
\mathcal{L}(\varphi) &= \{s_2\} \\
\text{pre}_\exists(\textsc{Init}) &= \{\} \\
\text{pre}_\exists(s_1) &= \{\textsc{Init}\} \\
\text{pre}_\exists(s_2) &= \{s_1\} \\
\text{pre}_\exists(s_3) &= \{s_2, s_3\} \\
\text{pre}_\text{K}(s_1) &= \{s_1, s_2\} \\
\text{pre}_\text{K}(s_2) &= \{s_1, s_2\} \\
\text{pre}_\text{K}(s_3) &= \{s_3\}
\end{aligned}
$$

Figure 5.9: The labelling function and existential pre-image, and knowledge pre-image, functions for Figure 5.8

In Figure 5.8; $l_i(s_1) = l_i(s_2)$. That is $l_i(s_1)$ and $l_i(s_2)$ are local states which are indistinguishable for Agent "X"; the epistemic relation is represented by the dashed line. The epistemic accessibility relation is *reflexive*; as such, every state is related to itself.
Y is the set of states returned from the knowledge pre-image function upon the $[\![\varphi]\!]$:

Depth 0:

Iteration 0
$\text{X} \leftarrow \{\}$
$\text{Y} \leftarrow \{\}$
$[\![\overline{\text{K}}_\text{X}(\varphi)]\!]_0 = \emptyset$

Depth 1:

Iteration 0
$\text{X} \leftarrow \{\}$
$\text{Y} \leftarrow \{\}$
$[\![\overline{\text{K}}_\text{X}(\varphi)]\!]_1 = \emptyset$

Depth 2:

Iteration 0
$\text{X} \leftarrow \{s_2\}$
$\text{Y} \leftarrow \{s_1, s_2\}$
$[\![\overline{\text{K}}_\text{X}(\varphi)]\!]_2 = \{s_1, s_2\}$

5.2.1 Correctness of the Algorithm $\text{Sat}_{\overline{\text{K}}}$

The algorithm $\text{Sat}_{\overline{\text{K}}}$ is sound and complete[1]:

Proposition 1. For every ECTLK formulae φ, IS $\vDash \varphi$ iff $\text{Sat}_{\overline{\text{K}}}(\varphi) \equiv \text{G}$ (G is the set of global states)

Proof. ($\Rightarrow$) By induction on the structure of φ. Let $\varphi = \overline{\text{K}}_i(\psi)$ and let IS, $g \vDash \overline{\text{K}}_i(\psi)$. This means that there exists a $g' \in \text{G} \wedge g \sim_i g'$ such that IS, $g' \vDash \psi$. By the induction step, $g' \in [\![\psi]\!]$; also we have $\mathcal{R}_i(g, g')$ by definition of $\mathcal{R}_i$. This implies that $g \in [\![\overline{\text{K}}_i(\psi)]\!]$, i.e. $g \in [\![\varphi]\!]$. □

Proof. ($\Leftarrow$) Straightforward, as the induction steps above are symmetrical. □

[1] Adapted from [53].

5.3 Model Checking of A^{G}CTLK with Seed States

The method of partial state space evaluation which we use in our method of bounded model checking is both *sound* and *complete* when we look at the restriction we placed upon the logic universal fragment of ACTLK. We said that, in A^{G}CTLK, all formulae are *invariant* – that is, the top most connective in the parse tree *must* be an **AG**.

Proposition 2. Seeded bounded model checking is *sound* with respect to the total model when a counterexample for $\mathbf{AG}(\varphi)$ 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. $\mathbf{EF}(\neg\varphi)$ holds in the seed state). As such, there exists a path in the full model which starts at the initial state and passes through this error state. From the semantics of CTLK, we also have $\mathbf{EF}(\neg\varphi)$ in the initial state. □

Proposition 3. Seeded bounded model checking is *complete* with respect to the total model when a counterexample for $\mathbf{AG}(\varphi)$ 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 $\mathbf{EF}(\neg\varphi)$, and neither could any of the partial state spaces starting from each individual seed, this means that there does not exist a reachble 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 which satisfies $\mathbf{EF}(\neg\varphi)$, so $\mathbf{AG}(\varphi)$ is satisfied by the model. □

5.4 Performance and Benchmarking

5.4.1 An initial investigation

The machine used for this evaluation was a dual core PC with 4GB of memory and an Intel Core 2 Duo clocked at 3.00GHz, with a 4096 KiB cache. The machine was running 32-bit Ubuntu Linux 8.04.2, a vanilla 2.6.24-19-generic kernel and glibc 2.7[2]. All experiments were performed four times, with the results presented here being the average across all four runs. Realistically, the only metric which required averaging was time, given that MCMAS is a deterministic process and will yield the same results each time for all other metrics.

The initial evaluation of our algorithm seemed to suggest that it *massively* under-performed that of "regular" model checking. Even though the implementation would, in some cases, only explore 25% of the state space compared to full forward verification, it still required *more* memory. These first tests were performed using MCMAS linked against a *vanilla* version (release 2.4.1) of the CUDD library (i.e. using garbage collection, asynchronous sift reorderings and a default cache). We initially looked at various sized models containing *Type 2* trains, whilst attempting to falsify the φ_{TGC5} property.

The results of these initial benchmarks can be seen in Table 5.1 – T represents the number of trains in the model, M is the maximum value of the service counter for those trains, and B denotes the service counter threshold at which the trains exhibit a fault. *Decrease* shows the comparative resource utilisation between BDD based BMC and MCMAS's default ("regular") model checking method – a value greater (less) than 1 indicates a decrease (increase).

Model			Decrease		
T	M	B	Memory	Time	States
2	10	4	0.9750	0.4535	3.6318
2	10	6	0.9632	1.2931	3.2970
3	10	4	0.1561	0.0366	1.4815
4	10	2	0.2338	0.0204	1.2672

Table 5.1: Memory, time and states results, with a *vanilla* CUDD verifying the φ_{TGC5}

It can be seen that, in the examples above, BMC did not produce *any* reduction in the resources used, with the exception of a model with 3 trains in which the regular final verification performed took longer than the intermediate checks as performed by BMC, but the memory used was still higher.

It became increasingly apparent that MCMAS required an *unusual* amount of memory to represent initial states, and this value did not fluctuate significantly from start to finish (i.e. there was little variance between the memory required when verification began, and the memory held by MCMAS when verification terminated). For instance, in a model with two Type 2 trains, a full counter of 10 and a breaking threshold of 4, MCMAS required 4911108 bytes to represent the single initial state, and yet only required 6210388 to hold the entire fix point of states (an additional 10605 states).

MCMAS was initially using the default CUDD constructor (Figure 5.10), which gave an initial pre-allocated cache size (256 KiB). This was changed such that, rather than being initialised with `CUDD_CACHE_SLOTS`, CUDD's constructor was passed 0, such that it initialised with *no* cache. This meant that, when verification began (after setting up all of the state variables, the transition relation, etc), MCMAS required, in the same model as the above, 721188 bytes to represent the single initial state, which then increased to 4152900 bytes to represent all of the 10605 in the fixed point state space. The results of performing the same evaluation as above, but without a default CUDD cache size, can be seen in Table 5.2

Interestingly, reducing the memory required to hold the initial state *also* reduced the total memory required to represent the full state space, regardless of the method of verification performed (6210388 bytes originally, 4152900 with the "tweaked" CUDD).

[2] `vector35` in the Department of Computing at Imperial College London.

```
/* initial size of subtables */
#define CUDD_UNIQUE_SLOTS 256

/* default size of the cache */
#define CUDD_CACHE_SLOTS  262144

Cudd
(
    /* The initial number of BDD variables */
    unsigned int numVars = 0,

    /* The initial number of ZDD variables */
    unsigned int numVarsZ = 0,

    /* The intitial size of the unique tables */
    unsigned int numSlots = CUDD_UNIQUE_SLOTS,

    /* The initial size of the cache */
    unsigned int cacheSize = CUDD_CACHE_SLOTS,

    /* Maximum memory occupation (0 is unlimited) */
    unsigned long maxMemory = 0
);
```

Figure 5.10: CUDD's default constructor (with additional comments)

Model			Decrease		
T	M	B	Memory	Time	States
2	10	4	1.6042	0.7292	3.6318
2	10	6	1.1884	2.2424	3.2970
3	10	4	0.2556	0.0389	1.4815
4	10	2	0.2510	0.0095	1.2672

Table 5.2: The relative reductions in memory, time and states explored with no initial CUDD cache

Although CUDD can adjust the size of the cache during execution, having *too* small a cache will reduce the number of unique BDD functions which can be stored in it, meaning that useful results will often be over-written. This also causes an increase in the number of cache misses. Each time the cache, as well as the unique tables, fill up, CUDD attempts to garbage collect unreferenced results from the cache. As such, having too small a cache then results in an increased number of garbage collections.

When CUDD is required to create a new internal node, and the number of nodes exceeds a given threshold[3], CUDD attempts automatic variable reordering. When performing BDD based BMC we are required to store a number of intermediate "working" results, all of which are represented as BDDs using CUDD nodes and are stored in the unique table. This means that BMC also affects the number of variable reorderings which CUDD performs.

Model			# Reorderings		# Garbage Collections	
T	M	B	Original	BDD-BMC	Original	BDD-BMC
2	10	4	6	11	10	16
2	10	6	12	9	17	11
3	10	4	13	55	27	69
4	10	2	17	26	30	61

Table 5.3: The number of asynchronous reorderings and garbage collections performed by CUDD

For the same models as previously, Table 5.3 illustrates the number of garbage collections, and variable reorderings, which CUDD performed during BMC. In some cases, BDD based BMC required over four times as many variable reorderings, and twice as many garbage collections, when compared to MCMAS's default behaviour.

The authors felt that the most *plausible* explanation for the under-performance of BDD based BMC was as a consequence of CUDD's automatic variable reordering. If CUDD performs 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).

'Sift'ing the variables to such a reordering could provide possible preferential orderings for either the regular approach or the bounded approach. This is heavily dependent upon the possible ordering generated; that is, it is not possible to say if a reordering should, or should not, be applied for a given method.

To provide a fair benchmark between MCMAS's regular approach and the approach set forward in this document, we edited a version of CUDD which permanently turned off both variable reorderings *and* garbage collection.

The internal CUDD function `cuddGarbageCollect` (in `cuddTable.c`) and the method `Cudd_ReduceHeap` (in `cuddReorder.c`) available in the API were changed such that they return immediately on function entry[4]. It should be noted that, as stated above, BMC utilises temporary variables and, without garbage collection to clean them up, these will cause an additional overhead which would not be present otherwise. As such, disabling this functionality is not necessarily a beneficial improvement for BMC.

Resource decreases, for a build of CUDD with these features disabled, can be seen in Table 5.4.

The final result for a four train model (italicised in Table 5.4) is a test which *did not complete*. CUDD halted the execution of MCMAS with a non-zero exit code and the string `Unexpected error`, indicating a serious, and unknown, problem. The test failed after 3101.51 s for regular verification (3100.11 s for BMC), exhausting 967.09 MiB (967.14 MiB).

[3] See [58] for more details.

[4] Both of these function bodies now contain `return;` as the first line.

Model			Decrease		
T	M	B	Memory	Time	States
2	10	4	4.0044	13.8500	3.6318
2	10	6	3.7481	12.6667	3.2970
3	10	4	1.4811	6.2529	1.4815
4	10	2	*1.0000*	*1.0005*	*1.0000*

Table 5.4: Statistics with no default cache, and with reordering and garbage collection both disabled

To allow for coherent and *fair* results, the rest of the results in this chapter—with the exception of the one-shot results in Section 5.5—have all been gathered with an MCMAS build with a zero sized initial CUDD cache, disabled asynchronous variable reordering and no garbage collection, even though the latter two of these *detrimentally hamstring* the model checker.

Our justification for disabling these features is that we wished to evaluate our novel *approach*, rather than benchmarking a specific *implementation*, and the benefits which such an implementation gains from the optimisations (such as 'sift'ed variable reorderings) arising from an auxiliary library.

5.4.2 The Faulty Train Gate Controller

The faulty train gate controller model, as presented in Section 4.4, provides us with a unique model for benchmarking our BMC implementation. This is because, in a faulty model (i.e. one which contains either type 1 or type 2 trains), eventually there will be a demonstrable counterexample prior to reaching a fixed point in the entire state space.

Using a model such as this allows us to demonstrate the possible benefits, and drawbacks, of using such an approach under different circumstances, such as a different number of agents (using parameterised formulae), with various size formulae – or with formulae which are true on the model and, as such, a counterexample cannot be found.

In the graphs which follow the resource usage of our implementation is expressed as a percentage of that which is required to verify the same model in MCMAS's default approach. Memory, time and states should be immediately obvious as to which metrics they represent. We can calculate the percent used by BMC with respect to full verification as follows:

$$(\text{BMC value}/\text{full value}) * 100$$

The "depth" metric represents the number of iterations (i.e. checks) which bounded model checking has to perform until it finds a counterexample *or* the state space reaches a fixed point. The "depth" of model checking, when performing full unbounded verification, is the number of iterations in which the the algorithm generates new "next" states before a fixed point is reached.

Figure 5.11 depicts bounded model checking of a model containing two *type 1* trains, with a maximum service counter of a 100 – trying to falsify the φ_{TGC2} property. It is clearly illustrated in the figure that our implementation is able to find a counterexample using resources proportional to that of the depth at which the property is found to be false.

The final group of results, WORKING, shows the attempted falsification of the same property but on a model with *type 3* trains, in which the property cannot be falsified. In this case, it can be seen that bounded verification only pays a very minor overhead in terms of memory used and time taken, when compared to regular verification of the same state space. Due to the inability of this formulae to be falsified on this model, bounded model checking has to explore the same number of states, and to the same depth, as the standard approach.

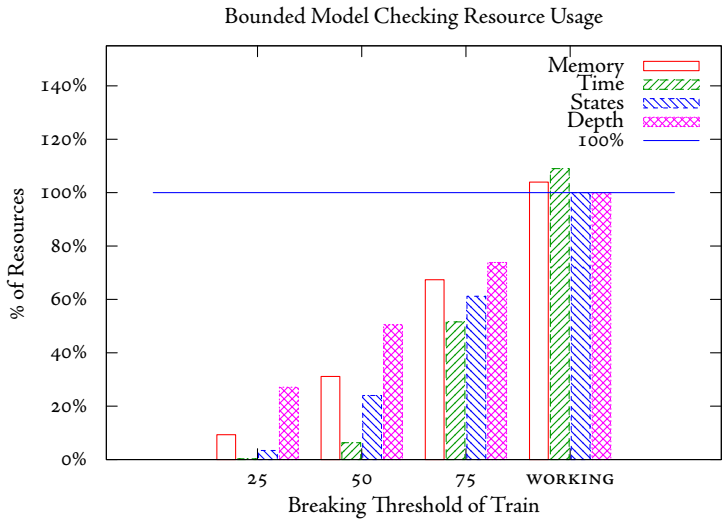

Figure 5.11: % resource use of BMC against regular model checking, with a model containing two type 1 trains, with a maximum depth of 100 and various breaking depths - φ_{TGC2}

The next four figures (Figure 5.12 to Figure 5.15) show that bounded model checking is an appropriate verification technique across different size models when using more *complex parameterised* formulae, which also deal with knowledge. All four show the attempted falsification of the φ_{TGC3} property – the number of agents in the model in Figure 5.12 and 5.13 is 2, whilst being 3 for Figure 5.14 and 5.15. We pass the number of agents in the model as the parameter of the formulae.

The graphs plotting data from models containing *type* 2 trains (Figure 5.12 and Figure 5.14) both show two interesting anomalies.

In the first, with 2 trains, although the initial results depict the expected trend – as the depth increases the resource requirements increase – towards the deeper breaking depths at which the property can be falsified, the resource requirements go *down*. This is still a favourable result in terms of BMC – it implies that, as the fixed point of reachable states is approached, the last few add more states than in the initial iterations.

In the second, with 3 trains, it appears that checking a shallower bound requires *more* resources than a deeper one – but this not the case. In the first two, the breaking bound is *lower* than the minimum amount of joint actions performed by both the train and controller to allow the train to enter the tunnel. This is reflected by the similarities between the results for a breaking bound of 1 and 3; once the breaking bound is higher than the minimum number of joint actions, we see a decrease percentage of resources required.

A point of distinction is that Figure 5.14 shows that, although BMC may not show drastic improvements, in terms of memory usage, over full verification, it does not show any significant penalties either. The point made previously about this model, showing that the breaking bound is less than that of the number of joint actions to enter the tunnel, is reiterated in this graph as well.

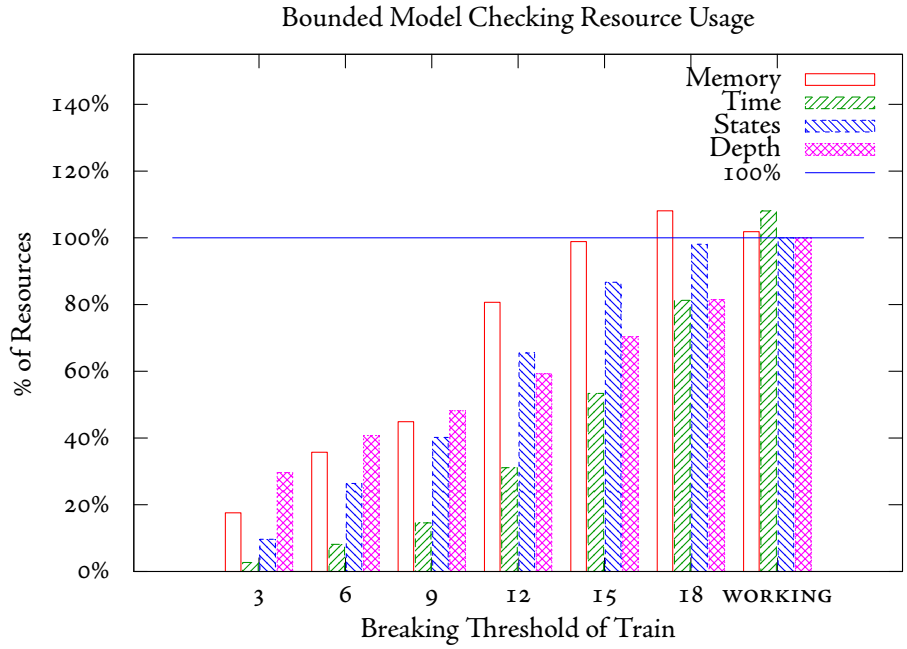

Figure 5.12: Resource usage in a model with two *type 1* trains and a maximum counter of 20 – φ_{TGC3}

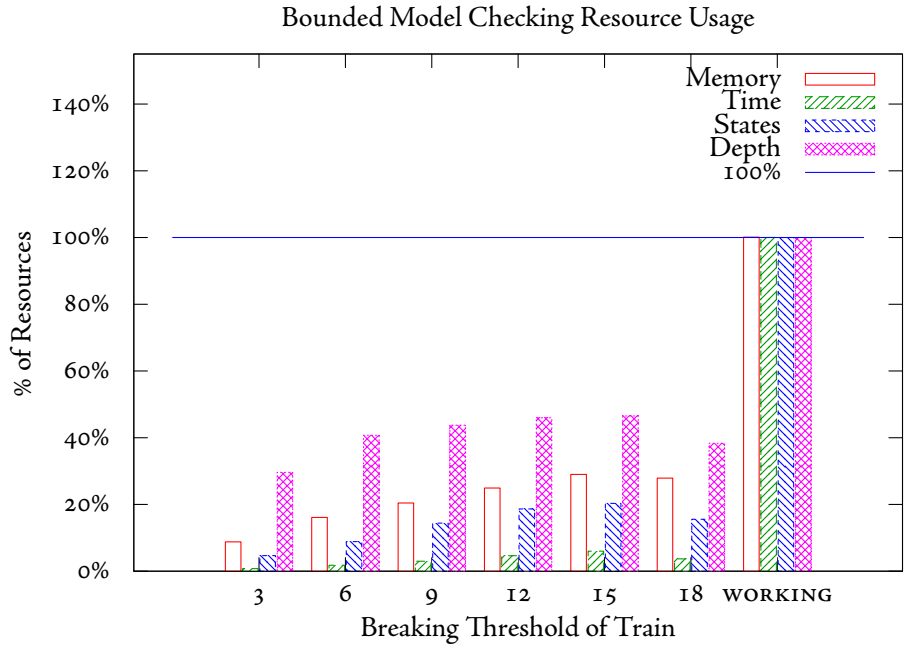

Figure 5.13: Resource usage in a model with two *type 2* trains and a maximum counter of 20 – φ_{TGC3}

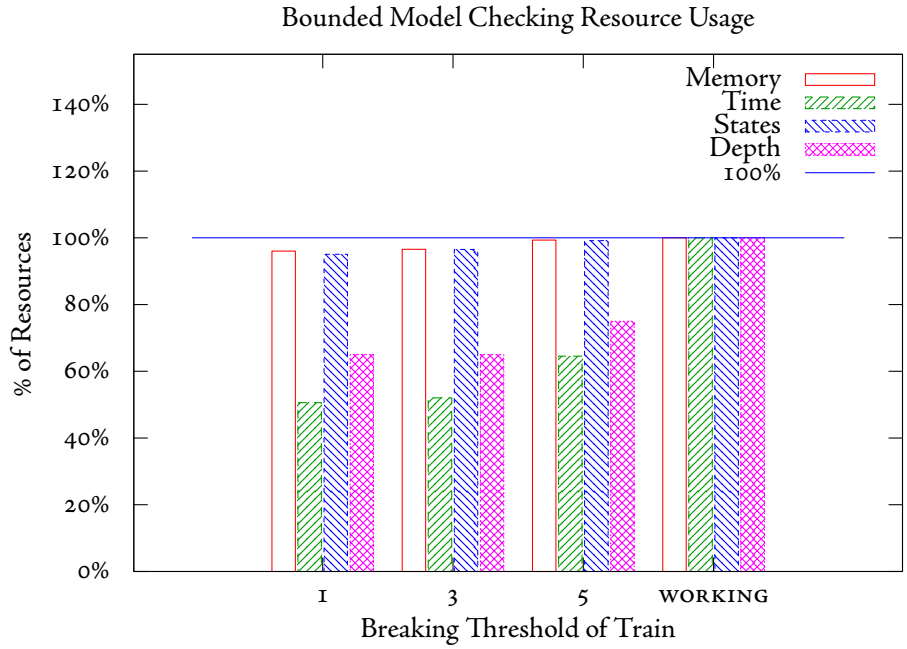

Figure 5.14: Resource usage in a model with three *type 1* trains and a maximum counter of 7 – φ_{TGC3}

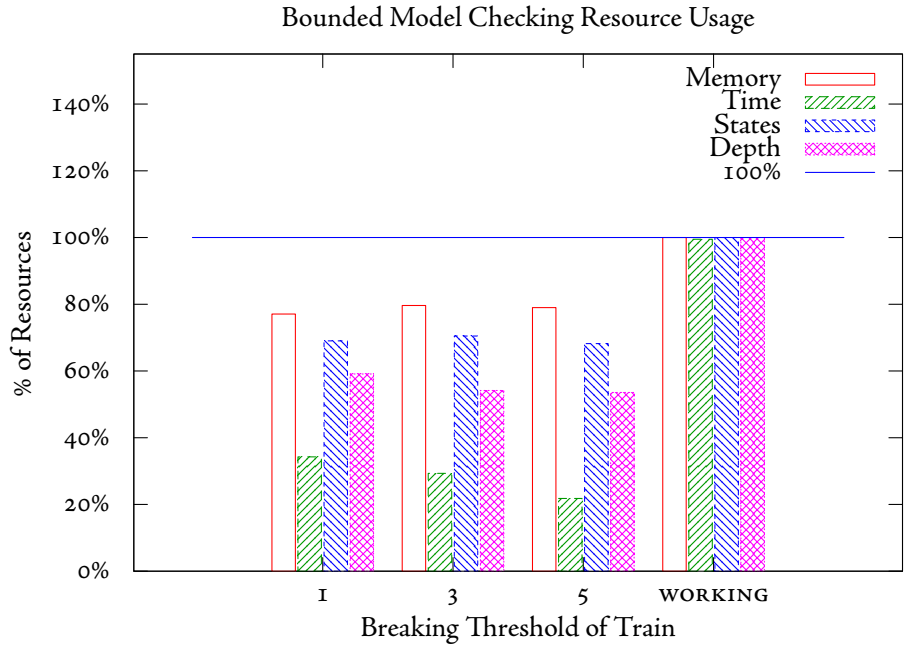

Figure 5.15: Resource usage in a model with three *type 2* trains and a maximum counter of 7 – φ_{TGC3}

The final four graphs for the faulty train gate controller model, Figure 5.16 to Figure 5.19, illustrate the difference in either memory required, or time taken, for falsifying different properties in the same model. The formulae $\varphi_{\text{TGC}4}$ and $\varphi_{\text{TGC}5}$ both use their parameterised versions and are based on the number of agents (trains) in the model.

The resource usage illustrated in models which have shallow breaking bounds (Figure 5.17 and Figure 5.19) both display the same correlation of results as previously. Again, in these particular models, there is a minimum number of joint actions required until a train can enter a tunnel; if the breaking depth is lower than this minimum number of moves, then no savings can be garnered.

In the graphs showing memory usage (Figure 5.16, Figure 5.17) we can see that there is a *slight* overhead when checking $\varphi_{\text{TGC}5}$ at a deep breaking depth, but this overhead appears to be less in the working model. This is because the number of iterations required to find a counterexample at the deepest breaking bound is more than for reaching a fixed point in the working model. For example, in a model with 3 trains and a maximum counter of 7, BMC requires 15 iterations to find a counterexample at a breaking depth of 5, but in the working model it only requires 11 to reach the fixed point.

The figures depicting the time taken for verification (Figure 5.18 and Figure 5.19) are the only ones which show any form of significant overhead for bounded model checking. When checking the property $\varphi_{\text{TGC}1}$ it can be seen that, in both models, bounded model checking is not preferable. It should be noted that this overhead is *only* demonstrated in a model in which the property cannot be falsified, meaning that BMC has to perform significantly more calculations in comparison. We feel obliged to point out that, given how we perform bounded verification and the fact that calculating the satisfiability set of the given formulae is not "free", this is exactly the result we would expect.

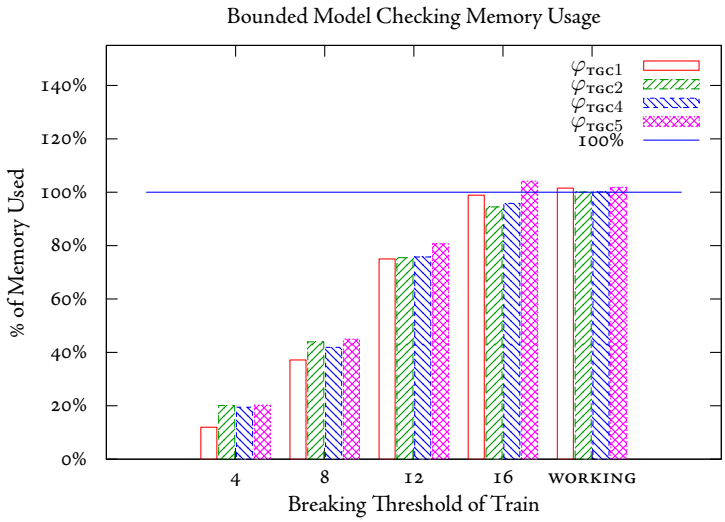

Figure 5.16: Memory usage for two *type* 2 trains, with a full service depth of 20 – when checking various formulae

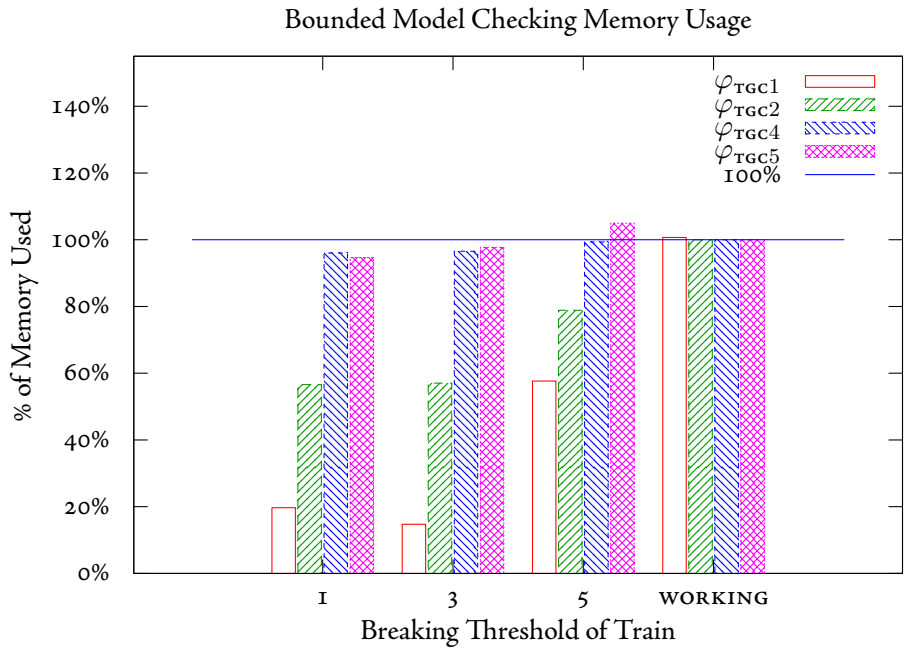

Figure 5.17: Memory usage for three *type 2* trains, with a full service depth of 7 – when checking various formulae

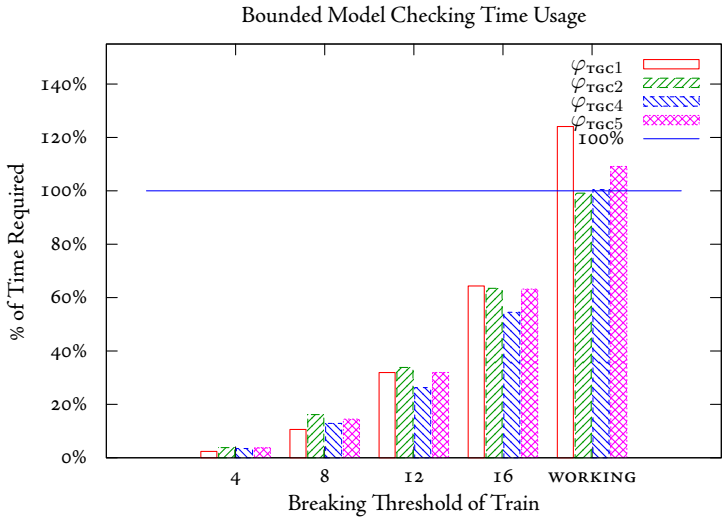

Figure 5.18: Time required for two *type 2* trains, with a full service depth of 20 – when checking various formulae

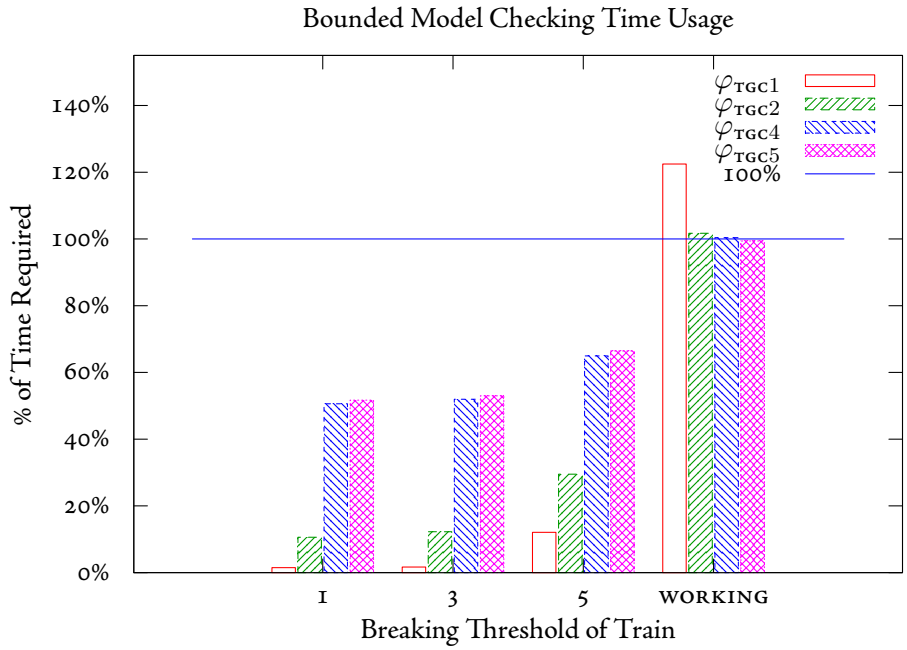

Figure 5.19: Time required for three *type* 2 trains, with a full service depth of 7 – when checking various formulae

5.4.3 MCMAS 0.9.8.5 Examples

After running all of the examples which are included with MCMAS, it is immediately obvious that, with the exception of one model, they are all very *small*, trivial examples. This means that the time required to check these models is very small – when measured with `/usr/bin/time` they all take 0.0s of "real" time to complete. This means that, with the exception of the larger model, BMC never displays an overhead or a benefit when looking at time alone.

The Book Store

The benchmarking results for The Book Store example can be seen in Figure 5.20. It can be immediately seen that, for ACTLK formulae ($\varphi_{BS1} - \varphi_{BS4}$) which are applicable to the model, BMC offers very little improvement in both memory used and time taken. Although we do not pay any overhead for using BMC, we do not get any improvements, in terms of resources used, either.

BMC does not display a memory improvement because the difference in memory required to hold the initial state space and the entire reachable states is *very* low. Ignoring the cost of checking the formulae, MCMAS requires 570324 bytes to hold the initial state and only 588196 bytes to hold the entire reachable states (i.e. the state space reaches a fixed point) – a difference of 17 KB. Due to the lack of extra memory required to store the rest of the model, when BMC can falsify the property (in all but with φ_{BS2}) the benefits are not clearly illustrated.

For the three properties which are falsifiable (φ_{BS1}, φ_{BS3} and φ_{BS4}) we can easily identify that MCMAS only has to explore a fraction of the states it would usually explore when performing BMC.

MCMAS originally required 10 iterations to reach the fixed point in the state space for the book model, and for φ_{BS2}, where the property is not falsifiable, we can clearly see that BMC pays a very slight memory increase for performing 10 satisfiability checks in comparison to a single check.

The Correct Bit Transmission Problem

φ_{BTP1}, when evaluated on a *correct* model for the bit transmission problem (Figure 5.21), is true, which means that BMC should pay an overhead. The case here is that the fixed point is reached within two state space

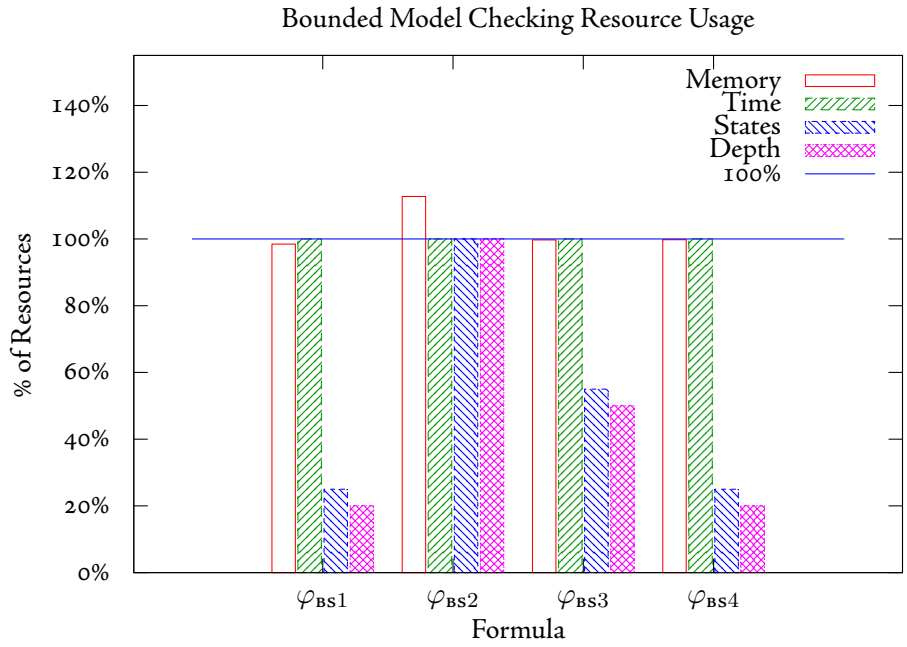

Figure 5.20: The Book Store

iterations, meaning that BMC only performs two extra checks. The memory required to both check and store the entire reachable states is negligible, which means that BMC does not display any disadvantage.

The Faulty Bit Transmission Problem

In the faulty BTP model (Figure 5.22), φ_{BTP1} is falsifiable – BMC can find a counterexample in the initial state. Again, we do not see any benefits in resource requirements. This is because, as the size of the model is small, there is very little memory difference between holding just the initial states and holding the set of all reachable states (the model has two initial states and only 18 reachable states in the full state space). This means that, although we can terminate early, this is not demonstrable.

The initial states are representable with 545444 bytes; BMC requires only 160 *bytes* to falsify the property, whilst full verification only requires an extra 2848 bytes to represent and check the entire state space.

The Dining Cryptographers

Although, in the dining cryptographers (Figure 5.23), φ_{DC2} is false in this model, we are unable to find a counterexample until we reach the same depth at which the fixed point is found, so BMC displays no immediate benefits. Whilst φ_{DC2} is true on the model, the fixed point in the state space is reached after two iterations, so performing two extra checks does not pay much of an overhead.

The "Software Development" Example

The only ACTLK property which is provided for the software development example is false in the initial state. This is an example in which BMC excels, as is clearly illustrated by the graph (Figure 5.24). Full verification of this model requires 70 iterations to reach the fixed point, nearly 14s (versus 0s) and uses nearly 60 times as much memory (full verification uses 118 MB, whereas the initial states are representable in only 2 MB).

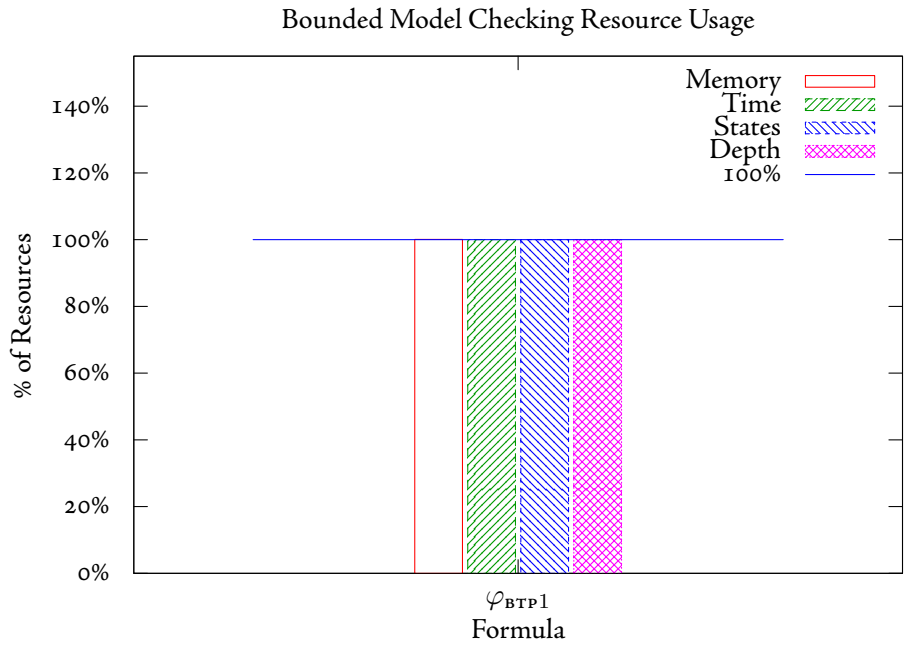

Figure 5.21: The Correct Bit Transmission Problem

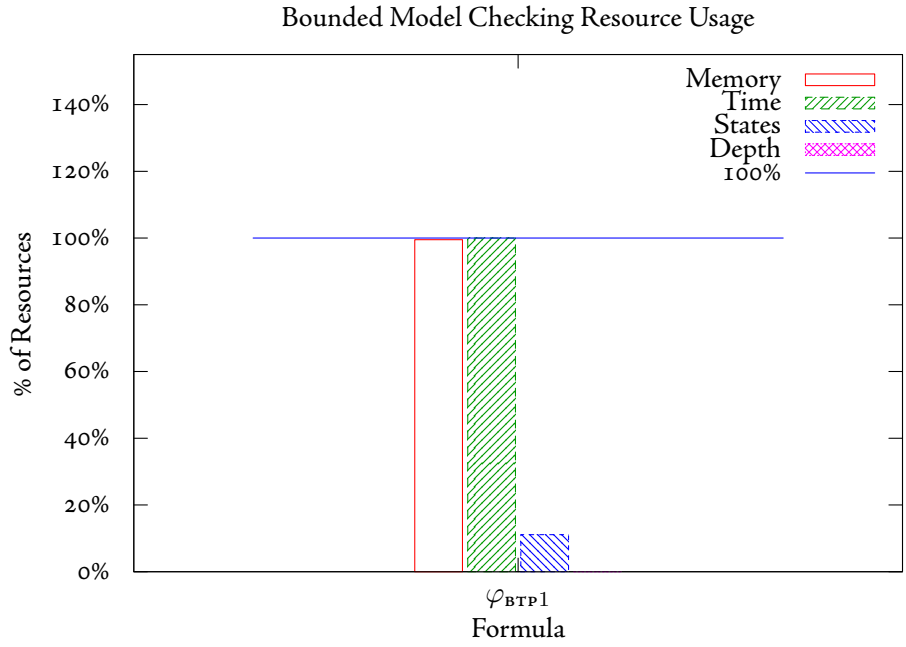

Figure 5.22: The Faulty Bit Transmission Problem

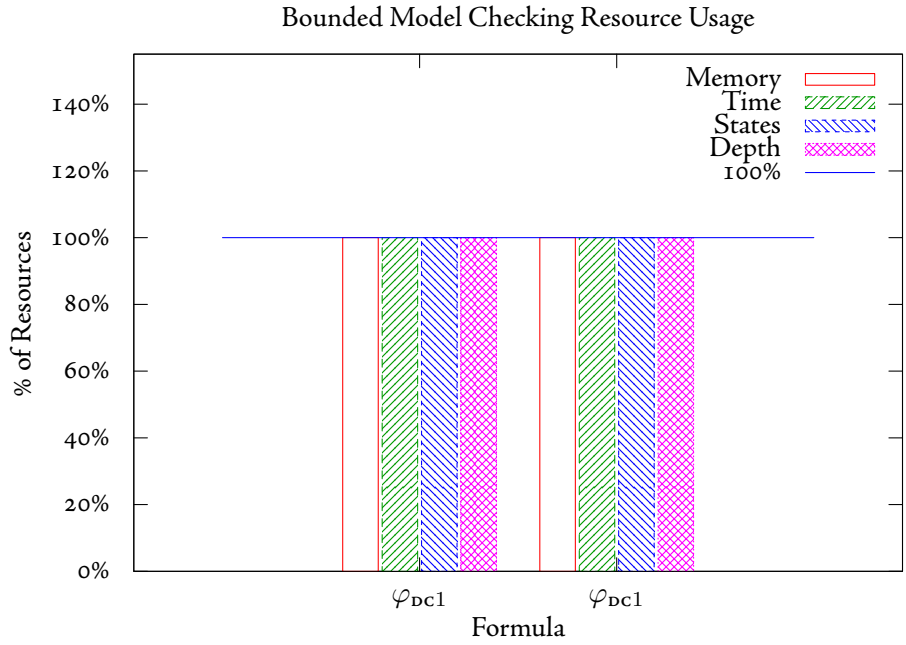

Figure 5.23: The Dining Cryptographers

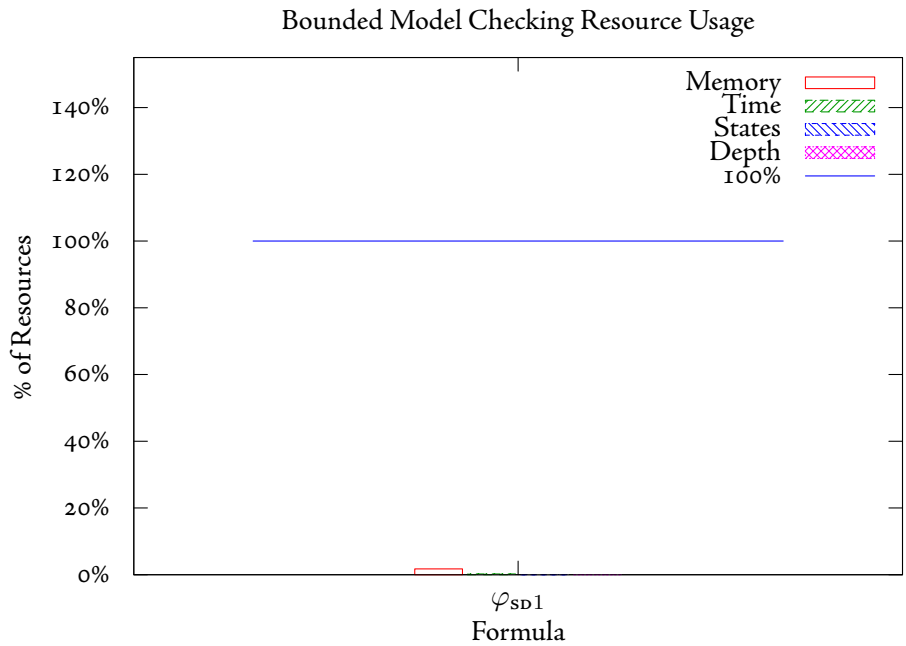

Figure 5.24: The "Software Development" Example

5.4.4 Length of Counterexample Found

Our attempt to implement BDD based bounded model checking on top of an existing model checker allows us to gain some functionality for "free" – in this case, counterexample generation. MCMAS is already capable of generating counterexamples (and witnesses) to ACTLK (ECTLK) formulae. It can do so by printing out the list of states, and joint actions between, displaying a trace through the model which invalidates the property.

To deem counterexample generation from BMC "successful", we felt we had two goals to satisfy:

1. Groce et al [22] make the distinction that "bounded model checkers often produce counterexamples that are difficult to understand due to the values chosen by a SAT solver". By harnessing MCMAS's counterexample methods, we should generate *understandable* counterexamples when using bounded model checking.

2. Biere et al [7] state that bounded model checking "finds counterexamples of minimal length". As such, we should ideally generate counterexamples which are smaller, or of equal length, to that of regular verification.

A comparison between the length of the counterexample generated between MCMAS's regular behaviour and our implementation can be seen in Table 5.5. The counterexamples have been constructed for various formulae in a two train model, composed of *type* 2 trains, a maximum service counter of 20 and a breaking depth of 10.

Method	Formula				
	$\varphi_{\text{TGC}1}$	$\varphi_{\text{TGC}2}$	$\varphi_{\text{TGC}3}$	$\varphi_{\text{TGC}4}$	$\varphi_{\text{TGC}5}$
Regular	25	17	4	4	12
BMC	13	16	4	4	*FAIL*

Table 5.5: Length of counterexamples generated between BMC and full verification

Figure 5.25 and Figure 5.26 show a counterexample for $\varphi_{\text{TGC}1}$ using regular, and then bounded, model checking. The authors argue that the counterexample in Figure 5.26 is *significantly* easier to understand.

$<\text{action}_C;\text{action}_{T1};\text{action}_{T2}>$ represents the actions performed by the Controller, Train1 and Train2 respectively. It can clearly be seen in the second figure that, in the transition from state 10 to state 11, Train1 performs the break action and from then on that train is in the tunnel, which invalidates the liveness property (in the final state for both traces, the train is "broken"). This trace is more convoluted in the second trace because of the multiple "break" actions performed by the train.

BMC was *unable* to generate a counterexample for $\varphi_{\text{TGC}5}$ – CUDD caused the program to terminate with a non-zero status, and printed out the string "`Unexpected error`"[5].

It should be noted that the counterexamples which were generated were done so by finding a counterexample to the original ACTLK formulae, as specified in the ISPL, using the K modality – and not the $\overline{\text{K}}$ modality[6].

Despite this, it was felt that counterexample generation was successful, both in terms of readability and length (i.e. BMC did not generate a longer counterexample).

[5] Forcing MCMAS to continue building up a set of reach states beyond its usual termination depth allowed us to find a counterexample shorter than for full verification for this property. But, as this was not an automated process, the result was omitted.

[6] `push_negations(int depth)` was modified such that, when performing counterexample generation, it did not translate K to its dual.

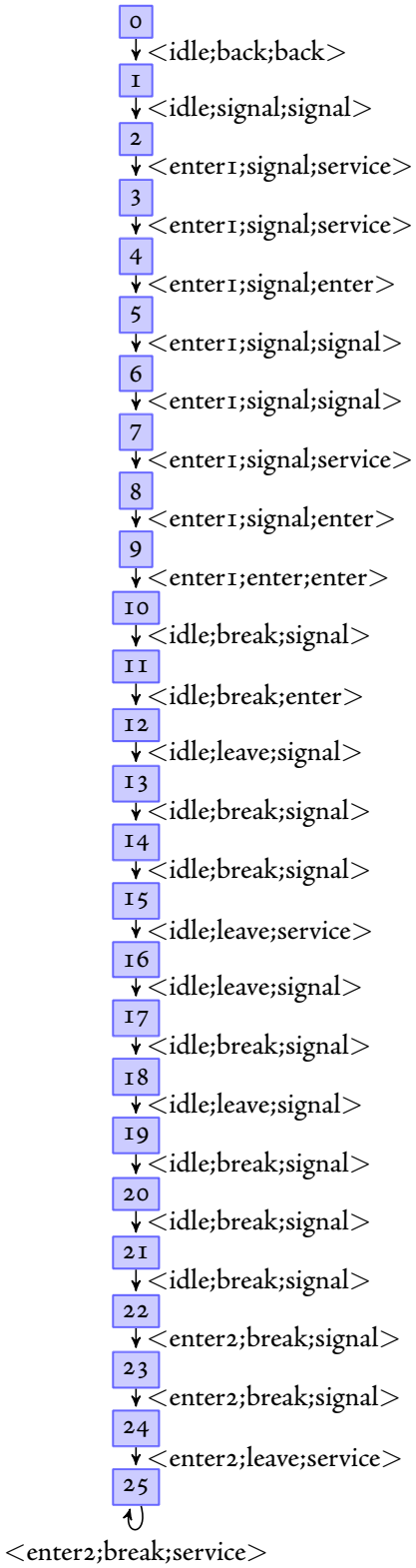

Figure 5.25: Counterexample for φ_{TGC1} from "regular" model checking

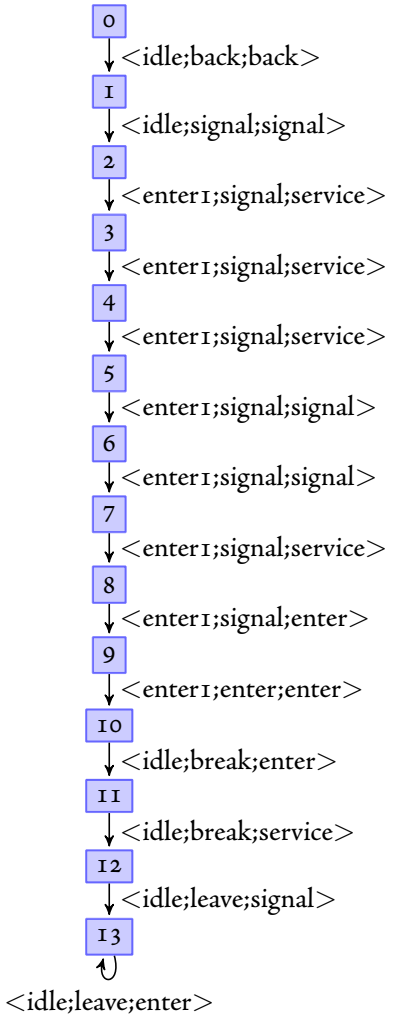

Figure 5.26: Counterexample for φ_{TGC1} from bounded model checking

5.4.5 Stress Testing MCMAS

BDD based BMC, as shown previously, can falsify properties on models requiring less memory than conventional verification uses. This means that there will be cases in which BDD based BMC will be able to verify the model, whilst other verification techniques will be unable to complete.

Results showing this can be found in Table 5.6 – T is the number of trains, M is the maximum service counter, B is the depth of the breaking bound.

Model			Regular			BMC		
T	M	B	Memory	Time	States	Memory	Time	States
3	20	5	*2131866612*	*8085.88*	*5834990*	64694612	2.43	74017
3	20	10	*1919076244*	*34383.60*	*4755710*	319959508	84.51	587164
3	20	15	1971699364	22448.24	3452620	961126804	1657.57	1853920
3	20	WORKING	1101036372	2836.36	2688260	1117194660	4007.68	2688260
4	20	5	*1814837044*	*3487.12*	*8560450*	829675748	406.35	1549040
4	20	10	*1371231508*	*3707.59*	*6195380*	*1380098852*	*7066.06*	*6195380*
4	20	15	*1371165972*	*3917.96*	*6195380*	*1380098852*	*7086.08*	*6195380*

Table 5.6: Stress testing MCMAS (Memory is given in bytes, Time in seconds)

The italicised results are ones in which MCMAS failed to complete the verification of the model. It can be seen from these results that there are cases in which BMC can *halt and succeed*, whilst regular model checking *halts and fails*.

To attempt to see how robust our implementation was, we performed more "tests" than depicted here. We attempted to verify models containing up to 6 agents (the maximum service depth was kept at 20, with varying depth breaking of 5, 10, 15 and a working model), but all these models caused both regular and bounded model checking to fail. Interestingly, these results did not suggest that a memory limit of approximately 1.36 GiB (which the latter results in Table 5.6 would suggest) was the only limiting factor. In a model with 6 trains, MCMAS failed at a memory limit of 1.69 GiB and approximately 11970900 states (nearly twice as many as shown in the table), but after a time of roughly 1500 seconds.

5.5 Evaluation of One-Shot BMC

The proposed method of "one shot" BMC was an attempt to alleviate the *memory* overhead associated with performing a satisfiability check at every depth. This BMC-related "penalty" was our original justification for turning off automatic reordering within CUDD, but, as one-shot BMC does not have this problem, we are able to benchmark against a version of MCMAS linked against a *vanilla* CUDD (zero sized initial cache, with both variable reorderings and garbage collection *enabled*).

We used the script developed to perform iterative one-shot model checking upon two models – one with two *type* 2 at a breaking depth of 20 (Figure 5.7) and the other with three *type* 2 at a breaking depth of 7 (Figure 5.8) – attempting to falsify φ_{BTP2}.

As before, a decrease *below* 1 represents that the system had an increased resource requirement in that configuration.

As expected, one-shot BMC can, in terms of memory used, out-perform the standard approach to model checking when the property can be falsified. In a model in which a counterexample cannot be found, the verification process performed by one-shot is *identical* to that of regular model checking. This is illustrated by the identical memory usage between the two approaches in a working model.

This approach was designed to alleviate the memory overhead at the expense of time and, with the exception of the anomalous result at a breaking depth of 10[7] this is mirrored in these results.

[7]The time taken to perform the single satisfaction check in regular model checking took an *unusually* long time, which is why one-shot appears beneficial in these results

B	Decrease	
	Memory	Time
5	1.00	0.50
10	1.78	1.24
15	1.70	0.63
WORKING	1.00	0.05

Table 5.7: Improvements for One-Shot BMC and full verification with reordering (2 Trains, Max Counter 20)

B	Decrease	
	Memory	Time
2	1.13	0.91
4	1.88	0.39
6	1.70	0.82
WORKING	1.00	0.14

Table 5.8: Improvements for One-Shot BMC and full verification with reordering (3 Trains, Max Counter 7)

5.6 Evaluation of Distributed MCMAS

There were *two* different factors which required consideration when benchmarking:

1. Depth of seed state generation – The threshold depth at which the seed states generated would affect the number of seed states which had to be verified
2. The number of slaves – If the master instance had more slaves available for it to use this should cause a decrease in the time which verification took.

Evaluation Difficulties

Our method of distributed model checking lends itself well to models which have deep counterexamples, and which suffer from the state space explosion problem, which it attempts to alleviate through state space partitioning. To be able to show proficient benchmarks we require models with these problems. The models which come with MCMAS are either too small or have shallow counterexamples, although the faulty train gate controller model can be constructed such that it displays a deep counterexample. Due to the cyclic design of the model, which allows agents to eventually return to a previous local state, the counterexample can be found from *every* seed state.

Another problem arises here: because the set of "next" states can transition into the current "reach" set, the next set for this model *subsumes* the reach set. This means that it is possible to have a seed state which is the same as the original initial state, as specified in the ISPL model[8].

Our distributed approach has been benchmarked using the faulty train gate controller, but, given the abilty to find counterexamples from any state in a faulty model, the distributed falsification of properties is shown in a particularly good light. This also gives rise to the fact that the property is falsified by the first slave to return a counterexample (usually the first slave that connects to the master instance). This does not invalidate or make these results any worse, it is simply an unintended bias of our method to this particular model.

Machine Specification

The networked hosts of these benchmarks were *identical* machines to that used to perform the previous benchmarks. The machines used were `vector30` through to `vector40` when *idle*.

The seed states were saved to the networked file system "`bitbucket`" – a networked file server for unmetered disc space – running a 813 GiB XFS file system with a 4 KiB block size (in a RAID configuration).

Each slave was connected to the master and every host to bitbucket, using gigabit ethernet.

5.6.1 Depth of seed states

We attempted to falsify properties on three different models, each with 3 trains, a maximum service counter of 7 and a breaking threshold of 4. The difference between the three models was the *type* of trains in each model; for the models which contained *type 3* trains, the property was not falsifiable. All of these benchmarks were performed using a single master and three slaves.

[8] It would be possible to perform `next\reach` (set minus) here to give a *strict* set of next states, but our implementation does not do this.

In Tables 5.9, 5.11 and 5.13:

BMC memory and states is the total number of states, and the memory (bytes) used to represent those states, which our original BMC algorithm had to explore to find a counterexample *or*, in the case of a working model, reach a fixed point.

Master memory (bytes) is the memory required to explore the model up to the given depth, prior to seed state generation, whilst "states" is the number of seed states which are generated (which, in this model, as noted above, is in fact the set of reachable states).

Slave "max" memory (bytes) is the maximum memory used by any one slave during the entire process. This value represents the *maximum* resource requirement for each slave[9], for the particular model, when performing distributed model checking.

The total states represents the summation of all the states explored by every single host in the verification process. Again, a weakness in these results is that, as soon as one slave returns a counterexample, the whole verification process terminates and we lose any intermediate results for the other slaves.

Tables 5.10, 5.12 and 5.14 show the decrease gained by using *distributed* bounded model checking against serial bounded model checking. A decrease below one indicates an increase for that model/method.

Depth 3

	BMC		Master		Slave (Max)		Max	Total
Model	Memory	States	Memory	States	Memory	States	Memory	States
Type 1	70396964	70181	825572	75	69742244	69459	69742244	69534
Type 2	198678532	284588	825572	75	175579220	258699	175579220	258774
Type 3	39705828	41681	825572	75	42212644	41681	42212644	3001032

Table 5.9: The resource usage of distributed bounded model checking at a seed depth of 3

	Decrease		
Model	Memory	Time	States
Type 1	1.009	0.817	1.009
Type 2	1.132	1.251	1.100
Type 3	0.941	0.024	0.014

Table 5.10: A comparison between BMC and distributed bounded model checking at a seed state generation depth of 3

For a seed state generation depth of 3, Tables 5.9 and 5.10, we can see that, for models in which the property can be falsified (*type 1* and *type 2*), the distributed approach is able to find a counterexample with less memory and less states, which, as stated above, is to be expected in this model.

Verification of *type 1* trains takes longer when distributed because bounded verification is able to quickly find a counterexample, whereas our distributed approach has the various overheads – the master to iterate over all of the reachable states and write them to disk, as well as both instances having to read and parse the same ISPL code. It should be noted here that the time taken for a slave to connected to the master, subsequent communication to take place between the nodes, and seeds to transfer to and from bitbucket is assumed to be negligible.

A model with *type 3* trains is expected to under-perform in distributed bounded verification because every single node, for every single seed, has to be explored to the fixed point, which means the same state space is computed for every seed.

[9]If one slave returns *faster* than another slave and has used *less* resources, this is the "maximum" value used. This is because, when one slave finds a counterexample, the master "kills" off all other instances, which causes any running instances of MCMAS to be terminated without the recording of any statistics.

The avid reader might wonder why verification from certain seeds may require *more* memory than the full BMC approach; this is caused by reordering. Starting from a different seed state may result in a different variable ordering `reach` BDD. This, in turn, may result in some slaves requiring more memory to represent the set of all reachable states, when compared to the memory required in serial bounded model checking

Depth 4

Model	BMC		Master		Slave (Max)		Max Memory	Total States
	Memory	States	Memory	States	Memory	States		
Type 1	70396964	70181	1156980	348	68132260	69724	68132260	70072
Type 2	198678532	284588	1157876	348	114868452	166133	114868452	166481
Type 3	39705828	41681	1157876	348	43777924	41681	43777924	14379945

Table 5.11: The resource usage of distributed bounded model checking at a seed depth of 4

Model	Decrease		
	Memory	Time	States
Type 1	1.033	0.772	1.002
Type 2	1.730	4.429	1.709
Type 3	0.907	0.005	0.003

Table 5.12: A comparison between BMC and distributed bounded model checking at a seed state generation depth of 4

We can see in Table 5.12 that, when we generate "deeper" seeds, this results in reaching a counterexample *quicker* because there are now less iterations required until a counterexample can be found. This is mirrored across memory and states as well.

Depth 5

Model	BMC		Master		Slave (Max)		Max Memory	Total States
	Memory	States	Memory	States	Memory	States		
Type 1	70396964	70181	1824660	1227	64144676	70169	64144676	71396
Type 2	198678532	284588	1659940	867	113094884	164529	113094884	165396
Type 3	39705828	41681	1824660	1227	43859860	41681	43859860	51017544

Table 5.13: The resource usage of distributed bounded model checking at a seed depth of 5

Model	Decrease		
	Memory	Time	States
Type 1	1.097	0.559	0.983
Type 2	1.757	4.134	1.721
Type 3	0.905	0.001	0.001

Table 5.14: A comparison between BMC and distributed bounded model checking at a seed state generation depth of 5

The number of states explored for distributed bounded model checking, when the formulae cannot be falsified, stands out in Table 5.13. At a depth of 5, the master instance generates 1227 seeds. This means

that, to be able to infer that the property is never falsified over the entire model, bounded model checking is performed until a fixed point is reached from *every* seed[10].

5.6.2 Number of slaves

Because every seed state, in a model with *type 1* or *type 2* trains, can lead to a demonstrable counterexample, this makes using this model difficult when demonstrating how varying the number of seeds can affect the time taken for verification.

To allow for *some* meaningful results we used a model containing three *type 3* trains with a maximum counter of 7. In this model no counterexample can be found, meaning that every single seed state is explored to the fixed point of states.

# Hosts	Time	Decrease
2	118.88	0.016
4	58.78	0.032
6	39.93	0.048
8	30.37	0.063

Table 5.15: Time for seeded bounded model checking, when compared to BMC, for a varying number of hosts when a counterexample cannot be found.

Table 5.15 illustrates the reductions in time gained through using different numbers of slaves at a seed generation depth of 3. The original time taken by BMC was 1.90s. Although the table displays increases, we can see that, as the number of hosts is increased, this increase gets lower. As such, in a model where a counterexample could be found, we can see that using more slaves would decrease the verification time (and increase verification efficiency).

5.6.3 Disk space overhead

Our current distributed implementation saves the set of all seed states to disk. This is done through the DDBMP library (version 2.0.3). It would be *unfair* to present the previous results without discussing the overhead which saving these states to disk imposes. Figure 5.16 shows the total size and the average seed size (in bytes), required for storing the seeds at different depths in a model with three *type 1* trains, a maximum counter of 7 and a breaking threshold of 4.

Depth	# Seeds	Total	Average
2	8	4416	552
3	75	41400	552
4	348	192121	552.072
5	1227	677426	552.099

Table 5.16: Disk space used to hold the set of seed states

The file system tested used a 4.0 KiB block size. This means that the on disk size for storing 1227 seeds was, in fact, 4.9 *MiB*, rather than the total file size of only 652 *KiB*. From this we can draw the conclusion that our distributed implementation requires almost 8 times the disk space than if a small block size was used but, by today's standards, 4.9 *MiB* is virtually nothing.

On average, the size for an individual seed is approximately 552 bytes. This allows us to realistically rule out any real network overhead for results previously, with respect to time, taken. Our justification for this is

[10]We feel obliged to point out that not *every* seed state has an enabled joint action available from it. This is why the total number of states (51017544) is less than the number of seeds multiplied by the fixed point states ($41681 * 1227$). In this case, there are 3 seed states which do not have an enabled action and, as such, do not have any successor states. (The fixed point of states is immediately reached and a path formula quantifed with **A** is true in a state with no successors).

that transferring such a small file over a gigabit connection would be virtually instantaneous.

5.7 Qualitative Evaluation

5.7.1 Effectiveness of deliverables

The solution we have presented here performs as expected; when possible, it can falsify properties early and, when not possible, it pays minimal overhead for exploring the whole of the model. Although this is a desirable goal, the means to an end, turning off garbage collection and asynchronous reordering may not be justifiable in other circumstances.

5.7.2 Elegance of solution

MCMAS's code, whilst not being the cleanest of code bases, was amenable to the implementation of all three of the techniques which were implemented. Our solution could have been a more *modular* and *cleaner* solution if MCMAS utilised the *object oriented* paradigm more. For instance, only having a single class representing a "formula", with a field specifying its "type", is not an ideal solution. Using virtual methods and dynamic dispatch would have allowed for various aspects of the code to be cleaner.

To be able to support the work presented, MCMAS's code has been "broken up". Originally the code was simply one *huge* main method, containing switch statements to decide which method of verification to attempt, with all the code inlined. Our solution is now more modular and, although it could be extended further, has already improved the maintainability of code and the provision of further verification techniques into MCMAS.

The distributed aspects of the model checker, in an attempt to avoid re-inventing the wheel and to keep MCMAS's code slightly cleaner, were implemented as an external Java application. Although this can be seen as a not particularly favourable solution, due to the lack of tight knit integration and the need for the application to understand MCMAS's output, it does have one benefit. It can now easily be extended to support other model checkers which support partial state space verification using "seed states".

5.7.3 Scalability

As shown in Section 5.4.5, when CUDD's optimisations are turned off, MCMAS is not able to verify large models. The methods presented for distributed bounded model checking could be used to alleviate this problem. Rather than using depth as a heuristic for seed state generation, we could generate seed states when the memory used, or the size of the current reach set, exceeds a given threshold.

Summary

In this chapter we have not only shown that the *foundations* of the approaches we have taken are sound, we have also evaluated our implementations of these approaches.

The performance of our BDD based implementation of bounded model checking has been shown to be favourable when used on models which have formulae which can be falsified prior to reaching the entire state space. We have also shown results which allow us to conclude that bounded model checking, when checking formulae which are satisfiable on the model, although possibly paying a slight overhead, still performs with a resource requirement equivalent to that of conventional satisfiability model checking. .

The crux of this problem is that the evaluation of these formulae on models is not known prior to starting the verification process. Model checking is generally used with properties which the author *believes* are true, with the intention of "bug hunting". In these circumstances, attempting to use bounded model checking may be a better approach to take.

Chapter 6
Conclusions

6.1 Project Review

6.1.1 Contributions

The goal of this project was the development of BDD based bounded model checking techniques for a branching time logic and its epistemic extensions. The main contributions which this project provides are:

- Theoretical contributions: Three different possible ROBDD based bounded model checking techniques ("full", "one shot" and distributed), as well as the BDD based methods required to also be able to check properties pertaining to knowledge upon these state spaces. Considerations have been given towards the validity of calculation of the satisfiability sets for forumulae based on existing fixed point methods. The technique put forward in this report for evaluating ECLTK formulae on partial state spaces is a wholly original contribution. The related recent developments in this field have only looked at LTL invariant properties and these methods, unlike the method here, are *not* complete.

- Deliverables: The main output of this project has been an extension to the existing model checker for multi-agent systems, `MCMAS`. An implementation of all three types of bounded model checking has been provided and discussed. An auxiliary script to allow for automated "one shot" BMC until a counterexample. A novel Java infrastructure allowing for the distribution of *any* model checker supporting seed states has been developed and used to show the effectiveness of verifying the logic A^GCTLK over a "grid".

- Examples: The introduction of a new *scalable* model has been discussed, and how it can be used to benchmark bounded model checking with a configurable number of steps, until the given formulae can be falsified, has been shown.

6.1.2 Comparisons

In Chapter 5, the implementation devised was evaluated and figures detailing performances were produced. It was shown that, with variable reordering turned off, our BMC implementation out-performs traditional model checking when a counterexample can be found, and, when it cannot, the overhead required for performing the iterative checks has been shown to be minimal. Performance benefits, including results which demonstrated reduction in the order of magnitude, were presented.

It is unfair to simply state a single value and claim that our BDD based BMC implementation performed that many times better (or worse) than the conventional methods.

As noted in the previous chapter, in the majority of cases in which bounded model checking is used in real systems, it is the case that there are errors to be found, which is the reason for performing model checking in the first place. In these cases, it should be immediately obvious that using a bounded model checking approach is a preferential selection.

6.1.3 Limitations, challenges and applications

When used without reordering, and as shown in Section 5.4.5, when performing bounded model checking, MCMAS can now check models in which it would have previously failed. This shows us that the implementa-

tion *can* show the expected benefits, and that the requirement of exploring less states results in a lower memory usage.

The problem arises when reordering is enabled (as shown in Section 5.4.1) When comparing our bounded implementation we can see that the extra processing checks required for performing bounded model checking are completely unfavourable. We feel that this is an *unfair* comparison, due to the majority of optimisations present in a library, such as CUDD, having been developed and researched with conventional model checking in mind. With this in mind it is highly unsurprising that we under-perform.

6.2 Further Work

6.2.1 Adding a visualiser to MCMAS

Currently, there is no way to *visualise* the internals of MCMAS during execution – this means that there could be some hidden anomolies happening "behind the scenes" that we are unable to detect.

To be able to properly evaluate the effectiveness shown between bounded, and conventional, model checking, it would be advantagous to have an internal visualiser for MCMAS. There have already been attempts at state space visualisation, but this is only half the story.

It has been shown that for bounded model checking to be shown as advantaneous, we have to disable variable reordering (and to a lesser extent garbage collection) in the underlying library.

Although this step We have provided conjecture as to why this leads to more favourable results for bounded model checking, it would be helpful if we could have more of insight to the BDDs which are used to represent the state space.

Such a tool, which could also be hooked into CUDD, could be used to analyse the variable reorderings used at run time – this would paint a better picture for *model checker designers* such that they can be aware of the behind the scenes optimisations the library provides.

6.2.2 Counterexample generation for $\overline{\mathrm{K}}$

In our current implementation, we do not directly support counterexample generation. When we come to find a witness to the negated formulae, we generate this from the satisfiability of the K modality and not the $\overline{\mathrm{K}}$ modality. A simple extension would be to see if generating the counterexamples with $\overline{\mathrm{K}}$ can lead to shorter, or more understandable, counterexamples.

6.2.3 Common and distributed Knowledge

The work presented here was mainly an attempt at a proof of concept for showing that BDD based bounded model checking could be at all effective when checking epistemic modalities – which we were successful in doing so. a possibility of an extension would be to allow for the checking of both Common and distributed Knowledge (C and $\overline{\mathrm{C}}$, and D and $\overline{\mathrm{D}}$).

6.2.4 Heuristics for seed state generation

In our distributed model checker, we do not handle seed state generation at all intelligently – in an attempt to find a counterexample quicker, we could prioritise certain seeds in the order of verification. For example, in an attempt to falsify $\mathbf{AG}(\neg p \vee \neg q)$, seed states in which p or q hold could be "priortised" – this would possible allow for a state in which the invariant ceases to hold being found sooner (i.e. If p holds in the seed state, then BMC attempts to to find q)

6.2.5 Itersection based BMC

The previous attempts at BDD based BMC all attempted an intersection based approach – that is, being able to represent the set of "bad" states, and then check if any of these states are inside the current reach.

Now we have a way of saving the states to disk, we could use this to create an hold a single error state, allowing for a an intersection based approach to model checking. We could also use MCMAS's "`RedStates`" to represent the set of errors states, and if an intersection of the reachable states and these red states is found, then an error has been located.

6.2.6 Saving Reach to disk in "one shot" BMC

In the same vein as the pervious point, given that MCMAS can now save a set of states to disk using DDDMP, when we perform "one shot" BMC, rather than throwing away the entire set of reachable states, we could save the current reach set to disk. When we start a new instance of MCMAS, we can get a cleared CUDD cache, but we do not loose the previous reach – this would save on over calculation.

6.2.7 More models/benchmarks

To provide a more through evaluation of our BMC implementation, we could attempt to check with more benchmarks. A set of examples such as "BEEM" (BEnchmarks for Explicit Model checkers) [66] could be used. The original train-gate-controller model which our version was based upon is included in this set.

6.2.8 Better Use of CUDD

CUDD provides API calls to clean up temporary variables from the cache, to limit the amount of memory which is used. The two calls `Cudd_RecursiveDeref` and `Cudd_Deref` can be used to protect a return result, but delete intermediate results calculated by a function.

Bibliography

[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* URL http://mtc.epfl.ch/software-tools/mocha/doc/c-doc/.

[2] **Nina Amla, Robert Kurshan, Kenneth L. McMillan** and **Ricardo Medel**. Experimental Analysis of Different Techniques for Bounded Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619/2003 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-00898-9. ISSN 0302-9743 (Print) 1611-3349 (Online), 2003 pp. 34–48. doi:10.1007/3-540-36577-X_4. URL http://www.springerlink.com/content/600uvxx254xlk8ta/.

[3] **Christel Baier** and **Joost-Pieter Katoen**. *Principles of Model Checking*. The MIT Press, May 2008. ISBN 026202649X.

[4] **M. Benerecetti, F. Giunchiglia, L. Serafini, Massimo Benerecetti** and **Luciano Serafini**. Model checking multiagent systems. In *Journal of Logic and Computation* volume 8(1998):pp. 8–3.

[5] **A. Biere, A. Cimatti, E. Clarke, O. Strichman** and **Y. Zhu**. Bounded Model Checking. In *Advances in Computers* volume 58(2003).

[6] **A. Biere, A. Cimatti, E. M. Clarke, M. Fujita** and **Y. Zhu**. Symbolic model checking using SAT procedures instead of BDDs. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. ACM, New York, NY, USA. ISBN 1-58133-109-7, 1999 pp. 317–320. doi:http://doi.acm.org/10.1145/309847.309942.

[7] **Armin Biere, Alessandro Cimatti, Edmund Clarke** and **Yunshan 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*. Springer Berlin / Heidelberg. ISBN 978-3-540-65703-3. ISSN 0302-9743 (Print) 1611-3349 (Online), 1999 pp. 193–207. doi:10.1007/3-540-49059-0_14. URL http://www.springerlink.com/content/vf286k9mq0jp05dh/.

[8] **Randal E. Bryant**. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Trans. Comput.* volume 35(1986)(8):pp. 677–691. ISSN 0018-9340. doi:http://dx.doi.org/10.1109/TC.1986.1676819.

[9] **Gianpiero Cabodi, Paolo Camurati** and **Stefano Quer**. Can BDDs compete with SAT solvers on bounded model checking? In *DAC '02: Proceedings of the 39th conference on Design automation*. ACM, New York, NY, USA. ISBN 1-58113-461-4, 2002 pp. 117–122. doi:http://doi.acm.org/10.1145/513918.513949.

[10] **A. Cimatti, E. Clarke, F. Giunchiglia** and **M. Roveri**. NuSMV: a new symbolic model checker. In *International Journal on Software Tools for Technology Transfer* volume 2(2000):p. 2000.

[11] **Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani** and **Armando Tacchella**. NuSMV 2: An OpenSource tool for symbolic model checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-43997-4. ISSN 0302-9743 (Print) 1611-3349 (Online), 2002 pp. 241–268. doi:10.1007/3-540-45657-0_29. URL http://www.springerlink.com/content/7hrq3m38utrrgywb/.

[12] **Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani** and **Armando Tacchella**. Integrating BDD-Based and SAT-Based symbolic model checking. In *Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-43381-1. ISSN 0302-9743 (Print) 1611-3349 (Online), 2002 pp. 265–276. doi:10.1007/3-540-45988-X_5. URL http://www.springerlink.com/content/ncfgrv2rf9kgwvqh/.

[13] **E. M. Clarke, E. A. Emerson** and **A. P. Sistla**. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Trans. Program. Lang. Syst.* volume 8(1986)(2):pp. 244–263.

[14] **Edmund Clarke, Armin Biere, Richard Raimi** and **Yunshan Zhu**. Bounded Model Checking Using Satisfiability Solving. In *Formal Methods in System Design* volume 19(2001)(1):pp. 7–34. ISSN 0925-9856 (Print) 1572-8102 (Online). doi:10.1023/A:1011276507260. URL http://www.springerlink.com/content/6r6m9pf34jh1a229/.

[15] **Edmund M. Clarke** and **E. Allen Emerson**. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. 1982 pp. 52–71.

[16] **Edmund M. Clarke, Orna Grumberg** and **Doron Peled**. *Model Checking*. MIT Press, 1999.

[17] **Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella** and **Moshe Y. Vardi**. Benefits of Bounded Model Checking at an Industrial Setting. In *Computer Aided Verification*, volume 2102/2001 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-42345-4. ISSN 0302-9743 (Print) 1611-3349 (Online), 2001 pp. 436–453. doi:10.1007/3-540-44585-4_43. URL http://www.springerlink.com/content/4p130cddq2jjtrr9/.

[18] **Cindy Eisner**. Using Symbolic CTL Model Checking to Verify the Railway Stations of Hoorn-Kersenboogerd and Heerhugowaard. In *Software Tools for Technology Transfer* volume 4(1):pp. 107 – 124. URL www.haifa.ibm.com/dept/svt/papers/trainssttt.ps.

[19] **J. Ezekiel** and **A. Lomuscio**. Combining fault injection and model checking to verify fault tolerance in multi-agent systems. 2009 .

[20] **Johannes Faber**. Verifying real-time aspects of the european train control system. In *17th Nordic Workshop On Programming Theory*. 2005 pp. 67–70.

[21] **Ronald Fagin, Joseph Y. Halpern, Yoram Moses** and **Moshe Y. Vardi**. *Reasoning About Knowledge*. MIT Press, 1995.

[22] **Alex Groce** and **Daniel Kroening**. Making the most of bmc counterexamples. In *Electronic Notes in Theoretical Computer Science* volume 119(2005)(2):pp. 67 – 81. Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC 2004), URL http://www.sciencedirect.com/science/article/B75H1-4FNNN9F-6/2/9aac90bbe0e97fe2453c29c665bc2349.

[23] **Wiebe van der Hoek** and **Michael Wooldridge**. Tractable multiagent planning for epistemic goals. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*. ACM, New York, NY, USA. ISBN 1-58113-480-0, 2002 pp. 1167–1174. doi:http://doi.acm.org/10.1145/545056.545095.

[24] **Michael Huth** and **Mark Ryan**. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004. ISBN 052154310X.

[25] **Subramanian Iyer**, **Jawahar Jain**, **Debashis Sahoo** and **E. Allen Emerson**. Under-approximation heuristics for grid-based bounded model checking. In *Electronic Notes in Theoretical Computer Science* volume 135(2006)(2):pp. 31 – 46. ISSN 1571-0661. doi:DOI:10.1016/j.entcs.2005.10.017. Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC 2005), URL http://www.sciencedirect.com/science/article/B75H1-4J77J57-4/2/084089258526bf0b960e083ff42c4a74.

[26] **Subramanian K. Iyer**, **Jawahar Jain**, **Mukul R. Prasad**, **Debashis Sahoo** and **Thomas Sidle**. Error detection using BMC in a parallel environment. URL http://www.stanford.edu/~sahoo/Research/papers/subbu-deep-bmc.pdf.

[27] **Subramanian K. Iyer**, **Jawahar Jain**, **Mukul R. Prasad**, **Debashis Sahoo** and **Thomas Sidle**. Error detection using BMC in a parallel environment. In *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg. ISBN 978-3-540-29105-3. ISSN 0302-9743 (Print) 1611-3349 (Online), 2005 pp. 354–358. doi:10.1007/11560548_30. URL http://www.springerlink.com/content/cl2jyu4mfg269lr4/.

[28] **Rune M. Jensen**. A Comparison Study between the CUDD and BuDDy OBDD Package. Applied to AI-Planning problems, September 2002.

[29] **M. Kacprzak**, **A. Lomuscio**, **T. asica**, **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*. Springer Berlin / Heidelberg. ISBN 978-3-540-24422-6. ISSN 0302-9743 (Print) 1611-3349 (Online), 2005 pp. 189–212. doi:10.1007/b105317. URL http://www.springerlink.com/content/8d9etbvuymu8nu6x/.

[30] **M. Kacprzak**, **A. Lomuscio** and **W. Penczek**. From Bounded to Unbounded Model Checking for Temporal Epistemic Logic. In *Fundam. Inf.* volume 63(2004)(2-3):pp. 221–240. ISSN 0169-2968.

[31] **Magdalena Kacprzak**, **Alessio Lomuscio** and **Wojciech Penczek**. Verification of multiagent systems via unbounded model checking. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. IEEE Computer Society, Washington, DC, USA. ISBN 1-58113-864-4, 2004 pp. 638–645. doi:http://dx.doi.org/10.1109/AAMAS.2004.296.

[32] **Joost-Pieter Katoen**. Concepts, Alogirthms and Tools for Model Checking, Semester 1998–1999.

[33] **Saul Kripke**. Semantical Considerations on Modal Logic. In *In Proceedings A Colloquium on Modal and Many-Valued Logics, Helsinki*. 1962 .

[34] **L. Lamport**. Proving the Correctness of Multiprocess Programs. In *IEEE Trans. Softw. Eng.* volume 3(1977)(2):pp. 125–143. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.1977.229904.

[35] **A. Lomuscio**, **T. asica** and **W. Penczek and**. Bounded model checking for interpreted systems: Preliminary experimental results. In *Formal Approaches to Agent-Based Systems*, volume 2699 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-40665-5. ISSN 0302-9743 (Print) 1611-3349 (Online), 2002 pp. 115–125. doi:10.1007/b11729. URL http://www.springerlink.com/content/qc9jek1am3w0tmgp/.

[36] **A. Lomuscio**, **F. Raimondi** and **M. J. Sergot**. Towards model checking interpreted systems. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*.

ACM, New York, NY, USA. ISBN 1-58113-683-8, 2003 pp. 1054–1055. doi:http://doi.acm.org/10.1145/860575.860792.

[37] **Alessio Lomuscio**, **Charles Pecheur** and **Franco Raimondi**. Automatic verification of knowledge and time with nusmv. In *IJCAI*. 2007 pp. 1384–1389.

[38] **Alessio Lomuscio**, **Hongyang Qu** and **Monika Solanki**. Towards verifying compliance in agent-based web service compositions. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC. ISBN 978-0-9817381-0-9, 2008 pp. 265–272.

[39] **Alessio Lomuscio**, **Hongyang Qu** and **Monika Solanki**. Towards verifying contract regulated service composition. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*. IEEE Computer Society, Washington, DC, USA. ISBN 978-0-7695-3310-0, 2008 pp. 254–261. doi:http://dx.doi.org/10.1109/ICWS.2008.115.

[40] **Alessio Lomuscio** and **Franco Raimondi**. MCMAS: A Model Checker for Multi-agent Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. ISBN 978-3-540-33056-1. ISSN 0302-9743 (Print) 1611-3349 (Online), 2006 pp. 450–454. doi:10.1007/11691372_31. URL http://www.springerlink.com/content/hr800h4080771487/.

[41] **Alessio Lomuscio** and **Marek Sergot**. The bit transmission problem revisited. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*. ACM, New York, NY, USA. ISBN 1-58113-480-0, 2002 pp. 946–947. doi:http://doi.acm.org/10.1145/544862.544961.

[42] **Alessio Lomuscio** and **Marek Sergot**. A formalisation of violation, error recovery, and enforcement in the bit transmission problem. In *Journal of Applied Logic* volume 2(2004)(1):pp. 93 – 116. ISSN 1570-8683. doi:DOI:10.1016/j.jal.2004.01.005. The Sixth International Workshop on Deontic Logic in Computer Science, URL http://www.sciencedirect.com/science/article/B758H-4C2R2K0-1/2/a86cd71f9a06e11819c2deddf21e6f5a.

[43] **Alessio R Lomuscio**. Notes from the course, 303: Software Engineering - Systems Verification, Spring 2008. URL http://www.doc.ic.ac.uk/~alessio/teaching/08/sv/sv.html.

[44] **Stephan Merz**. Model Checking: A Tutorial Overview. In *Modeling and Verification of Parallel Processes* (editor **F. Cassez et al.**), volume 2067 of *Lecture Notes in Computer Science*, pp. 3–38. Springer-Verlag, Berlin, 2001.

[45] **Charles Pecheur** and **Franco Raimondi**. Symbolic model checking of logics with actions. In (2007):pp. 113–128. doi:http://dx.doi.org/10.1007/978-3-540-74128-2_8.

[46] **W. Penczek** and **A. Lomuscio**. Bounded model checking for interpreted systems, 2002. URL http://www.doc.ic.ac.uk/~alessio/papers/Penczek-Lomuscio-TR.ps.

[47] **W. Penczek** and **A. Lomuscio**. Verifying epistemic properties of multi-agent systems via bounded model checking, 2003. URL http://citeseer.ist.psu.edu/article/penczek03verifying.html.

[48] **Wojciech Penczek**, **Bozena Wozna** and **Andrzej Zbrzezny**. Bounded model checking for the universal fragment of CTL. In *Fundam. Inf.* volume 51(2002)(1):pp. 135–156. ISSN 0169-2968.

[49] **Franco Raimondi**. *Model Checking Multi-Agent Systems*. Ph.D. thesis, 2006. http://www.cs.ucl.ac.uk/staff/f.raimondi/pubs/thesis.pdf.

[50] **Franco Raimondi**. Notes from the course, GS03/4203: Verification and validation, 2007 – 2008. URL http://www.cs.ucl.ac.uk/staff/F.Raimondi/teaching/.

[51] **Franco Raimondi** and **Alessio Lomuscio**. A tool for specification and verification of epistemic properties in interpreted systems. In *Electronic Notes in Theoretical Computer Science* volume 85(2004)(2):pp. 176 – 191. ISSN 1571-0661. doi:DOI:10.1016/S1571-0661(05)82609-X. LCMAS 2003, Logic and Communication in Multi-Agent Systems, URL http://www.sciencedirect.com/science/article/B75H1-4G6932F-5V/2/5e7d156deb72017230487f10e4e7d1f0.

[52] **Franco Raimondi** and **Alessio Lomuscio**. Verification of Multiagent Systems via Ordered Binary Decision Diagrams: An Algorithm and Its Implementation. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. 2004 pp. 630–637.

[53] **Franco Raimondi** and **Alessio 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*. Springer Berlin / Heidelberg. ISBN 978-3-540-24422-6. ISSN 0302-9743 (Print) 1611-3349 (Online), 2005 pp. 213–221. URL http://www.springerlink.com/content/h5tnkadaw0bgx0my/.

[54] **Franco Raimondi**, **Alessio Lomuscio** and **Hongyang Qu**. MCMAS v0.9.6: User Manual. URL http://dfn.dl.sourceforge.net/sourceforge/ist-contract/mcmas-0.9.6.2.tar.gz.

[55] **Richard Rudell**. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, Los Alamitos, CA, USA. ISBN 0-8186-4490-7, 1993 pp. 42–47.

[56] **Marek Sergot**. Notes from the course, 499: Modal and temporal logic, Autumn 2008. URL http://www.doc.ic.ac.uk/~mjs/teaching/499.html.

[57] **M. Sirjani**, **A. Movaghar**, **A. Shali** and **F. S. de Boer**. Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models. In *Journal of Universal Computer Science* volume 11(2005)(6):pp. 1054–1082.

[58] **Fabio Somenzi**. CUDD: CU Decision Diagram Package Release 2.4.1, May, 2005. URL http://www.cs.ubc.ca/~ajh/courses/cpsc513/cudd.ps.

[59] **Francesca Toni**. Notes from the course, 474: Multi-agent Systems, Autumn 2008. URL http://www.doc.ic.ac.uk/~ft/teaching.html.

[60] **Ingo Wegener**. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. ISBN 0-89871-458-3.

[61] **Michael J. Wooldridge**. *Reasoning about Rational Agents*. MIT Press, 2000.

[62] **Bozena Wozna**, **Alessio Lomuscio** and **Wojciech Penczek**. Bounded model checking for deontic interpreted systems. In *Electronic Notes in Theoretical Computer Science* volume 126(2005):pp. 93 – 114. doi:DOI:10.1016/j.entcs.2004.11.015. Proceedings of the 2nd International Workshop on Logic and Communication in Multi-Agent Systems (2004), URL http://www.sciencedirect.com/science/article/B75H1-4FKXPY9-K/2/0a4c45371f53831ab07794690ed1fae0.

Web References

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

[64] Advanced Model Checking. http://www-i2.informatik.rwth-aachen.de/i2/amc09/.

[65] Akka. http://staff.science.uva.nl/~lhendrik/SystemDescription.html.

[66] BEEM: BEnchmarks for Explicit Model checkers. http://anna.fi.muni.cz/models/.

[67] BuDDy: A BDD package. http://buddy.sourceforge.net/manual/main.html.

[68] CUDD: Colorado University Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/.

[69] Eclipse. http://www.eclipse.org/.

[70] Flex. http://flex.sourceforge.net/.

[71] GNU Bison. http://www.gnu.org/software/bison/.

[72] Graphviz. http://www.graphviz.org/.

[73] Lufthansa Technik: Aircraft maintenance. http://www.lufthansa-technik.com/applications/portal/lhtportal/lhtportal.portal?requestednode=424&_pageLabel=Template7_8&_nfpb=true&webcacheURL=TV_I/Media-Relations/Media-Archive/Archive-Press-Releases/Previous-Press-Releases/Press-Releases-1997/Maintenance_e.xml.

[74] MCMAS 0.9.6.2. http://dfn.dl.sourceforge.net/sourceforge/ist-contract/mcmas-0.9.6.2.tar.gz.

[75] MCMAS v0.9.8.2: User Manual. http://www-lai.doc.ic.ac.uk/mcmas/manual.pdf.

[76] NuSMV User Guide. http://nusmv.irst.itc.it/NuSMV/papers/sttt_j/html/node21.html.

[77] PDMC – Parallel and Distributed Methods in verifiCation. http://pdmc.informatik.tu-muenchen.de/.

[78] Rebeca : Reactive Objects Language. http://khorshid.ece.ut.ac.ir/~rebeca/.

[79] The DDDMP package. http://fmgroup.polito.it/quer/research/tool/tool.htm.

Appendix A

BMC Implementation in MCMAS

```
// A pair of a formula - pair.first = actlk, pair.second = ectlk
typedef std::pair<modal_formula *, modal_formula *> modal_formula_p_pair;

// A vector of formula pairs
typedef std::vector<modal_formula_p_pair> modal_formula_p_pair_vector;
```

Figure A.1: The extended type system of MCMAS

```
// Pairs of formulae we need to check
modal_formula_p_pair_vector *bmc_formulae;

// Pairs of formulae which we've found counterexamples to
modal_formula_p_pair_vector *bmc_false_formulae;

// Initialise them ...
bmc_formulae = new vector<modal_formula_p_pair>;
bmc_false_formulae = new vector<modal_formula_p_pair>;
```

Figure A.2: New global variables to store formulae to prove

```
// Loop over all the given formulae
for (unsigned int i = 0; i < is_formulae->size(); i++)
{
    // Pick up the ACTL formula
    modal_formula *actl = &(*is_formulae)[i];

    // Push through any existing negations
    modal_formula *actl_pushed = actl->push_negations(0);

    // Create the negation of the formula
    modal_formula *actl_pushed_negated = new modal_formula(3,
                actl_pushed);

    // Push those negations through
    // We now have an ECTL formula
    modal_formula *ectl = actl_pushed_negated->push_negations(0);

    // Double check that we've got an ACTL and ECTL
    if (actl->is_ACTLK_BMC() && ectl->is_ECTLK_BMC())
        bmc_formulae->push_back(modal_formula_p_pair(actl, ectl));

}
```

Figure A.3: The calculation of the ECLTK formulae from the ACTLK

```
void check_formulae_BMC(void)
{
    // _init atom old holds at the the intial states
    string str = "_init";
    (*is_evaluation)[str] = is_istates;

    // Construct iota
    modal_formula *init = new modal_formula(new atomic_proposition(&str));

    // Where we're going to store the results
    BDD result;

    for (modal_formula_p_pair_vector::iterator iter =
                bmc_formulae->begin();
                iter != bmc_formulae->end();)
    {
        // Dereference the iterator
        modal_formula_p_pair temp = (modal_formula_p_pair) (*iter);

        // Construct iota -> phi
        modal_formula * f = new modal_formula(4, init, temp.second);

        // Check the implication
        result = f->check_formula();

        // Delete the formulae
        if (f)
            delete f;

        // If phi holds in the initial state
        if (result == reach)
        {
            // We've found a counterexample to the original formulae
            // We save it in the vector of falsified formulae
            bmc_false_formulae->push_back(temp);

            // And remove it from the formulae to check
            // and update iter to be the next item in the vector
            iter = bmc_formulae->erase(iter);
            continue;
        }

        // Move the iterator on
        ++iter;
    }

    // Delete the init formula
    if (init)
        delete init;
}
```

Figure A.4: A function to check all the ECTLK formulae

```
void bdd_bmc(void)
{
    // The current reachable states are the initial states
    reach = in_st;

    // q1 is the new set of next states
    BDD q1 = bddmgr->bddZero();

    // Initial next states is the initial states
    BDD next1 = in_st;

    // Start at a depth of 0
    int k = 0;

    // Whilst we still have formulae to check
    while (!bmc_formulae->empty())
    {
        // Check them with respect to the current reach
        check_formulae_BMC();

        // If we satisfy them, we exit the loop
        if (bmc_formulae->empty())
            continue;

        // We're now searching a deeper bound
        ++k;

        // Construct the next set of states
        for (unsigned int i = 0; i < agents->size(); ++i)
            next1 *= (*vRT)[i];
        next1 = Exists(v, next1);
        next1 = next1.SwapVariables(*v, *pv);
        next1 = Exists(a, next1);

        // Construct the new set of reachable states
        // From the union of the current reach and the next
        q1 = reach + next1;

        // If the set of reachable states hasn't change
        // We've reached a fixed point
        if (q1 == reach)
        {
            cout << "Fix point reached" << endl;
            break;
        }
        else
            reach = q1; // If not, store the new reachable states
    }

    /*****************************************************************
    * SNIP ----->
    *****************************************************************/
```

Figure A.5: The first half of the BMC method in MCMAS

```
    /*************************************************************************
    * <----- SNIP
    *************************************************************************/

    // When we reach here, we've either:
    // Reched a fix point of the state space
    // Or disproved all of the formulae

    // If we've disproven any formulae with BMC
    if (bmc_false_formulae->size())
    {
        // We print them out
        cout
                << "The following formulae have been verified"
                << "using BMC on the negation:"
                << endl;
        for (modal_formula_p_pair_vector::iterator iter =
                bmc_false_formulae->begin();
                iter != bmc_false_formulae->end(); ++iter)
        {
            modal_formula_p_pair temp = (modal_formula_p_pair) (*iter);
            modal_formula actl = *temp.first;

            cout << "  Formula " << actl.to_string()
                    << " is FALSE in the model" << endl;
        }
    }

    // If we have remaining unchecked formulae
    if (!bmc_formulae->empty())
    {
        // Clear the old formulae
        is_formulae->clear();

        // And construct a new is_formulae vector
        for (modal_formula_p_pair_vector::iterator iter =
                bmc_formulae->begin();
                iter != bmc_formulae->end(); ++iter)
        {
            modal_formula_p_pair temp = (modal_formula_p_pair) (*iter);
            modal_formula actl = *temp.first;
            is_formulae->push_back(actl);
        }

        // Use the regular check_formulae method
        check_formulae();
    }
}
```

Figure A.6: The second half of the BMC method in MCMAS. Printing out the values for each formula, and a final check for formulae we have been unabled to falsify