Outsourcing Report

Exploring the State Space:

Towards Faster Model Checking

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

Supervisor: Dr Alessio R Lomuscio ⟨alessio@doc.ic.ac.uk⟩

January 16, 2009

Abstract

We aim to investigate methods of reducing the memory usage and amount of time required by a model checker. More specifically, this report will attempt to explore a method for bounded model checking using boolean decision diagrams in place of existing boolean satisfiability methods when checking multi-agent systems. Consideration will be given to the implementation of this method in the MCMAS [56] model checker.

This report examines the feasibility of such a project, providing an overview of existing work in the same field and leading to an implementation specification, in addition to exploring methods and metrics for evaluating such a project.

Contents

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

Our daily lives are becoming more and more dependant upon computerised systems, but without any reassurance of the reliability of these devices. The systems which humans generally interact with are classed as *reactive* systems, because of their continual interaction with their environment. It is quite obvious 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 dependant on these systems, and the systems themselves become increasingly more complex, bugs in these systems can be easily overlooked and missed.

Currently, there is a migration from "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.

Ariane-5

One of the biggest, and most infamous, software incidents was the failure of Ariane 501 in June 1996, which exploded just seconds after launch. At an altitude of nearly 4000m, the rocket left its flight path, broke up and exploded. The problem was caused by the conversion of a 64-bit floating point number, to a 16-bit signed integer. This fault caused an unhanded exception in the inertial reference system, and also in the back-up system which contained the same bug. The destruction of the rocket was automatic due to the rocket leaving the flight path.

This problem was due to the reuse of parts of the system from an older version of the control software, without proper *verification*.

Figure 1.1: The launch ...

Figure 1.2: ... and subsequent failure

Model Checking

Model checking is a automated way of checking the validity of a given property, upon a model of a system, both of which been formalised in a given logic. Model checking has been applied to both hardware and software problems.

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" [47]. 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*.

As with reactive systems, we need a method of verifying these systems, such that we can know that they behave as expected. This has lead to the development of methods for model checking multi-agent systems, along with logics for knowledge and belief.

1.2 Report Structure

The remainder of this report is separated into three main sections:

- **Background** – The next chapter aims at reviewing some of the theory behind this introduction
- **Specification** – This chapter defines a set of goals for the project
- **Evaluation** – The final chapter looks at providing a way of evaluating if the project has been successful in meeting the specified goals

2 Background

2.1 Temporal Logics

It is possible to describe a finite state system (e.g. a reactive one) as a *Kripke structure* [30]:

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

Where

- $\mathcal{S}$ is a finite set of states
- $\mathcal{I}$ is the set of initial states $(\mathcal{I} \subseteq \mathcal{S})$
- $\mathcal{R}$ is a transition relation between two sets of 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 appending states reachable from $\mathcal{R}$ to an initial state $s \in \mathcal{I}$. Given that $\mathcal{R}$ is total, all the traces of the behaviours of a Kripke structure are also infinite.

This "infinite" behaviour of systems, modelled with Kripke Structures, led to Lamport's [31] classification of the requirements of these systems to fall into two categories:

- **Safety** "something bad never happens" – A system will satisfy the stated property, if one of the behaviours of the system does not allow this property.
- **Liveness** "something good will eventually happen" – in this case the system must exhibit a specific behaviour to satisfy the property.

There are two 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:

$\mathbf{F}\varphi \equiv \text{true}\mathbf{U}\varphi$

$\mathbf{G}\varphi \equiv \neg\mathbf{F}\neg\varphi$

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. Satifcation, $\vDash$, is defined as follows:

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

Where $\mathcal{R}^0(s) = s$, 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 [17] - CTL is a branching time logic. It is able to express the existence of, and properties upon, runs of a system.

[1] Adapted from [28], see also [36, 14]

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:

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

Semantics of CTL

Definition 4. A *path* [18] in a Kripke structure $\mathcal{M}$ is an infinite series of states $\pi = s_0, s_1, \ldots$ such that $\forall i \geq 0, (s_i, s_{i+1}) \in \mathcal{R}$.

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

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

2.2 Multi-Agent Systems

2.2.1 "Agents"

Definition 6. Agents [46] 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* [35] 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. [46] 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." [40, 42] 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 systems is an synchronous one, that is, all of the agents within the system transition at the same time, then we can define the global transition function:

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

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. [40] 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. [40]
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:

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

For this relation to hold, the global states g and g' are *indistinguishable* for agent i.

Modelling an interpreted system [40] 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 $l_i(g) = l_i(g')$). That is, for a global state, the local state of the agent does not change
- $\mathcal{V}$ is the valuation function for the propositional atoms

2.2.3 CTLK

Interpreted systems provide "computationally grounded semantics that has been used to model knowledge..." [42]. CTLK is a epistemic logic, it can deal with the notion of *knowledge*.

Syntax of CTLK

Definition 11. BNF definition of the CTLK language

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

Semantics of CTLK

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

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

CTLK is enriched with a further two epistemic operators, for a set of agents Γ:

"Everybody Knows" [24] The modal operator $\mathbf{E}_\Gamma\varphi$ is exactly true when all members of the group Γ know φ, formally:

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

This modality shows that every agent (or every agent in the set A) knows φ. It is sometimes referred to as "mutual knowledge".

"Common Knowledge" [24] 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:

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

Fagin et al [24] provided the following useful equivalence:

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

Definition 12. Semantics of CTLK [40]

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 [17, 15] 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, 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*" [28]

> "*Model checking is an effective technique to expose potential design errors*" [5]

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 algorithms below have been adapted from [28, 26, 39]. For further information, the reader is advised to consult these texts.

Algorithm 1 CTL Model Checking [5]

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

Definition 13. Sub-formulae of a CTL-formula [28]
Let $p \in \mathbf{AP}$, and φ, ψ be CTL formulae, then:

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

Algorithm 2 $\text{SAT}(\varphi : \text{FORMULA}) : \textit{set of}\ \text{STATE}$

1: **if** $(\varphi = \text{TRUE})$ **then**
2: **return** S
3: **else if** $(\varphi = \text{FALSE})$ **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** $\mathcal{S} \setminus \text{SAT}(\varphi)$
9: **else if** $(\varphi = \mathbf{EX}\varphi_1)$ **then**
10: **return** $\text{SAT}_{\mathbf{EX}}(\varphi_1)$
11: **else if** $(\varphi = \mathbf{E}[\varphi_i\mathbf{U}\varphi_2])$ **then**
12: **return** $\text{SAT}_{\mathbf{EU}}(\varphi_1, \varphi_2)$
13: **else if** $(\varphi = \mathbf{EG}(\varphi_1))$ **then**
14: **return** $\text{SAT}_{\mathbf{EG}}(\varphi_1)$
15: **end if**

State pre-image functions

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

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 $\text{SAT}_{\mathbf{EX}}(\varphi : \text{FORMULA}) : \textit{set of}\ \text{STATE}$

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

Algorithm 4 $\text{SAT}_{\mathbf{EU}}(\varphi : \text{FORMULA}, \psi : \text{FORMULA}) : \textit{set of}\ \text{STATE}$

1: $W \leftarrow \text{SAT}(\varphi)$
2: $X \leftarrow \mathcal{S}$
3: $Y \leftarrow \text{Sat}(\psi)$
4: **while** $X \neq Y$ **do**
5: $\quad X \leftarrow Y$
6: $\quad Y \leftarrow Y \cup (W \cap \text{pre}_{\exists}(Y))$
7: **end while**
8: **return** Y

Algorithm 5 $\text{SAT}_{\mathbf{EG}}(\varphi : \text{FORMULA}) : \textit{set of}\ \text{STATE}$

1: $X \leftarrow \text{Sat}(\varphi)$
2: $Y \leftarrow \mathcal{S}$
3: $Z \leftarrow \emptyset$
4: **while** $Z \neq Y$ **do**
5: $\quad Z \leftarrow Y$
6: $\quad Y \leftarrow X \cap \text{pre}_{\exists}(X)$
7: **end while**
8: **return** Y

For each node of the parse tree (i.e. for each sub formula ψ of φ) the set of states $\text{SAT}(\psi)$ is calculated.

2.3.2 Counterexamples and witnesses

One advantage of CTL model checking is the ability of the model checker to generate *counterexamples* and *witnesses*. 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" [18]. 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" [18].

2.3.3 Symbolic Model Checking

Binary Decision Diagrams

Binary Decision Diagrams (**BDDs)** [11, 5, 26] 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 the 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 [36, 26].

Semantics of BDDs The semantics of a BDD $\mathcal{B}_f$ is the value of 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 [35, 5]:

- **Elimination** – For two inner nodes u, v, for which $succ_1(u) = succ_0(u) = v$, all of the incoming edges to u and 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 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. [26] 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 [36] for details.

Tests of BDDs [26] 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 [26].

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 $2^{n-1} < |\mathcal{S}| \leq 2^n$). The boolean vector can be based around the propositional formulae which hold at that state ($\mathcal{L}(\mathcal{S})$). If there are not enough boolean variables to give each state a unique boolean vector, then it should be padded with

addition 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 usually a function mapping of $\mathcal{S}$ onto propositional atoms in **AP**. It is more convenient to consider it as the opposite, mapping atoms to subsets of $\mathcal{S}$ which satisfy that atom. This set of states $\mathcal{L}_p = \{s \in \mathcal{S} \mid p \in \mathcal{L}(s)\}$ [18]. This set can now be represented in the same way as $\mathcal{I}$.

2.3.4 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**) [10, 8, 9, 16]. BMC searches for the minimum length counterexample which violates the system specification. The algorithm looks for 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 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** [38]: 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 [38]: a k-path π is a *loop* if $\exists l : 0 < l \leq k$ and $(\pi(k), \pi(l))$

Definition 17. ***k*-model** [38]: Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ be a model, and $k \in \mathbb{N}^+$. $\mathcal{M}_k = (\mathcal{S}, \mathcal{I}, Paths_k, \mathcal{L})$, 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 [38]

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:

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

$|\mathcal{M}|$, 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 [38]) is that if $\mathcal{M}, s \vDash \varphi$, then $\mathcal{M}_k, s \vDash \varphi$ when $k = |\varphi|$.

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

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

Definition 20. Sub-models of $\mathcal{M}$ [38]
$\mathcal{M}_k = (\mathcal{S}, \mathcal{I}, \mathit{Paths}_k, \mathcal{L})$ is a k-model of $\mathcal{M}$. The structure $\mathcal{M}_k = (\mathcal{S}', \mathcal{I}, \mathit{Paths}'_k, \mathcal{L}')$ is a *sub-model* of $\mathcal{M}_k$, such that $\mathit{Paths}'_k \subseteq \mathit{Paths}_k$, $\mathcal{S}' = \mathit{States}(\mathit{Paths}_k)$, and $\mathcal{L} = \mathcal{L}|_{\mathcal{S}'}$

Definition 21. The function f_k : CTL Formula $\rightarrow \mathbb{N}$ [38]

- $f_k(p) = f_k(\neg p) = 0$
- $f_k(\varphi \vee \psi) = max\{f_k(\varphi), f_k(\psi)\}$
- $f_k(\varphi \wedge \psi) = f_k(\varphi) + f_k(\psi)$
- $f_k(\mathbf{EX}\varphi) = f_k(\varphi) + 1$
- $f_k(\mathbf{EG}\varphi) = (k+1) \cdot f_k(\varphi) + 1$
- $f_k(\mathbf{E}[\varphi \mathbf{U} \psi]) = k \cdot f_k(\varphi) + f_k(\psi) + 1$

Algorithm 6 BMC($\mathcal{M}$: KRIPKE STRUCTURE, ψ : ACTL FORMULA) [38]

1: $\varphi \leftarrow \neg\psi$ {φ is an ECTL formula}
2: **for** $k \leftarrow 1$ to $|\mathcal{M}|$ **do**
3: $\quad \mathcal{M}_k \leftarrow k$-model of $\mathcal{M}$
4: $\quad$ Select sub-models of $\mathcal{M}'_k$ of $\mathcal{M}$ with $|Path'_k| \leq f_k(\varphi)$
5: $\quad [\mathcal{M}^{\varphi,\iota}]_k \leftarrow$ propositional formula of the transition relation of all the sub-models of $\mathcal{M}'_k$
6: $\quad [\varphi]_{\mathcal{M}_k} \leftarrow$ propositional formula of the translation of φ over all the sub-models of $\mathcal{M}'_k$
7: $\quad [\mathcal{M}, \varphi]_k \leftarrow [\mathcal{M}^{\varphi,\iota}]_k \wedge [\varphi]_{\mathcal{M}_k}$
8: $\quad$ 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* [38] 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 [38]:

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

$I_s(w)$ encodes the state s of the model, $s[i] = 1$ is encoded by $w[i]$, and $s[i] = 0$ is encoded by $\neg w[i]$. $L_{k,j}(l)$ 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 [38]:

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

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 distinquished 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 [38]:

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

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.

2.3.5 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 met, allow an agent to change the local state. For more details see [40].

The model checking algorithm in Section 2.3.5 requires a representation R_t of the global transition relation between two global states (g, g'):

$$R_t(g, g') \text{ iff } \exists a \in \mathcal{P}(l_i(g)) : t(g, a, g')$$

Model Checking CTLK

The algorithms from the section below have been adapted from [40].

Algorithm 7 $\text{SAT}_{\text{CTLK}}(\varphi : \text{FORMULA}) : \textit{set of}\ \text{STATE}$

1: **if** $(\varphi \in \mathbf{AP})$ **then**
2: **return** $\mathcal{L}(\varphi)$
3: **else if** $(\varphi = \neg\varphi_1)$ **then**
4: **return** $\mathcal{G} \setminus \text{SAT}_{\text{CTLK}}(\varphi)$
5: **else if** $(\varphi = \mathbf{EX}\varphi_1)$ **then**
6: **return** $\text{EX}_{\text{CTLK}}(\varphi_1)$
7: **else if** $(\varphi = \mathbf{E}[\varphi_i \mathbf{U} \varphi_2])$ **then**
8: **return** $\text{EU}_{\text{CTLK}}(\varphi_1, \varphi_2)$
9: **else if** $(\varphi = \mathbf{EG}(\varphi_1))$ **then**
10: **return** $\text{EG}_{\text{CTLK}}(\varphi_1)$
11: **else if** $(\varphi = \mathbf{K}_i(\varphi_1))$ **then**
12: **return** $\text{EG}_{\text{CTLK}}(\varphi_1)$
13: **else if** $(\varphi = \mathbf{E}_\Gamma(\varphi_1))$ **then**
14: **return** $\text{EG}_{\text{CTLK}}(\varphi_1)$
15: **else if** $(\varphi = \mathbf{C}_\Gamma(\varphi_1))$ **then**
16: **return** $\text{EG}_{\text{CTLK}}(\varphi_1)$
17: **end if**

The functions EX_{CTLK}, EG_{CTLK} and EU_{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 $\mathbf{K}$. 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

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

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

Algorithm 8 $\text{K}_{\text{CTLK}}(\varphi : \text{FORMULA}, i : \text{AGENT}) : \textit{set of}\ \text{STATE}$

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

Algorithm 9 $\text{E}_{\text{CTLK}}(\varphi : \text{FORMULA}, \Gamma : \textit{set of}\ \text{AGENT}) : \textit{set of}\ \text{STATE}$

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

Algorithm 10 $C_{CTLK}(\varphi : \text{FORMULA}, \Gamma : \textit{set of}\ \text{AGENT}) : \textit{set of}\ \text{STATE}$

1: $X \leftarrow SAT_{CTLK}(\neg\varphi)$
2: $Y \leftarrow \mathcal{G}$
3: **while** $X \neq Y$ **do**
4: $\quad X \leftarrow Y$
5: $\quad Y \leftarrow pre_{C_\Gamma}(X, \Gamma)$
6: **end while**
7: **return** Y

MCMAS

Model Checking Multi-Agent Systems (**MCMAS**) [32] is a model checker which allows automatic verification of multi-agent systems. 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 [40], using an external BDD library. It is based around the Colorado University Decision Diagram (CUDD) [49] package.

CUDD is a C++ based BDD library which allows for easy code reuse. CUDD provides:

- The data structures necessary for BDD creation, handing 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 [54] diagrams for the BDDs it is used to represent.

Figure 2.1: An example C++ program using the CUDD library [39]

```
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";
  }
}
```

ISPL - Interpreted systems programming language MCMAS accepts descriptions of multi-agent 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 set of a formulae which the user wishes to check. The structure of ISPL files is *roughly* based upon the work presented in [6].

Syntax of an ISPL file [23]

- `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 [52] and GNU's Bison [53]. 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. Much like the CUDD library, MCMAS is able to generate Graphviz Dot files which represent the found counterexample.

MCMAS also provides an Eclipse [51] interface which supports the creation of skeleton MCMAS files, as well 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.

MCMAS, as of version 0.9.6 [57], does not include any kind of build system, it comes with a simple `make` file, which the user is required to manually edit to hard-code the location of the CUDD library.

2.4 Project Directions

There are a number of different research areas which have been looked at with respect to not only increasing what can be model checked (i.e. reducing the state space, or the time taken, to allow for larger systems to be checked) but also different methodologies and algorithms for use within a model checker. With respect to looking at the MCMAS model checker, there were various areas which I looked at to explore, and then implement and use in this project.

2.4.1 A Knowledge Compilation Map

Knowledge compilation [21] has arisen as a method for propositional reasoning in which a propositional theory can be compiled *off-line* into an intermediate target language, which can then be used to answer a number of queries in *polytime*. The advantage of this is that most of computational overhead is processed during the off-line phase, which then provides computational benefits during the query stage.

A *representation* language is one which is human readable, while a *target compilation* language does not require to be human readable, but it should be able to perform a number of queries in poly time.

Definition 22. Negation Normal Form (NNF) [22]
A sentence in $\mathbf{NNF_{AP}}$ is a rooted, directed acyclic graph (DAG) where each leaf node is labelled with true, false, X or $\neg X$, $X \in \mathbf{AP}$; and each internal node is labelled with $\wedge$ or $\vee$ and can have arbitrarily many children. The size of a sentence Σ in $\mathbf{NNF_{AP}}$, denoted $|\Sigma|$, is the number of its DAG edges. Its height is the maximum number of edges from the root to some leaf in the DAG.

Darwiche [21] suggested one form of *target compilation* languages called decomposable negation normal form (DNNF), a generalisation of disjunctive normal form (DNF) and a specialisation of NNF. Deciding the satisfiability of a DNNF can be done in a linear time.

Definition 23. A decomposable negation normal form DNNF is a negation normal form satisfying the decomposable property: for any conjuction $\bigwedge_i \alpha_i$ appearing in the form, no atoms are shared by the conjuncts α_i.

A BDD is another form of NNF. There exist more than a dozen subsets, the main question asked, and a solution for proposed, in [22]:

> *"What subset [of NNF] should one adopt for a particular application?"*

The rest of the paper goes on to discuss the benefits and disadvantages of the all these *target compilation* languages and how one should be selected.

Tree of BDDs

Subbarayan et al [44] devised a NNF called a "Tree-of-BDDs" (ToB). Their method was developed to alleviate the variable ordering problem which can greatly effect the size of BDD. A tree decomposition scheme was used to decompose the larger monolithic-bdd into a tree of related clustered variables. Re-ordering a variable within one of these sub trees would only effect the sub tree which contained it. Then, in 2007, Subbarayan et al [45] developed a method of how this form could be used as a *target compilation* language such that it could be used as a target language for knowledge compilation. This paper also introduced a method of conjunctive normal form (CNF) to ToB compilation.

2.4.2 Compositional Model Checking

Clarke et al [7] reasoned that one of the main causes of the state space explosion problem is the parallel composition of the sub-components of a system. "The problem occurs because the number of states in the global model is exponential in the number of component processes". *Compositional reasoning*, is a "divide-and-conquer" approach which attempts to alleviate this problem by attempting to infer properties of the global system from verification of the individual components in isolation. They discuss a various number of ways of compositional verification, one of these includes *assume-guarantee reasoning*.

Assume Guarantee

Assume-guarantee breaks up a system into sub-components, and then attempts to verify each of them in isolation against an "assumption" which is supposed to represent the other components in the system. Assume-guarantee is based around formulae of the kind $\langle A \rangle M \langle P \rangle$ [19] where M is a subsystem, P is a property, and A is an assumption of M's environment. The formula holds if M constrained by A, satisfies P. To verify if two systems M_1, M_2, running parallel $M_1 \| M_2$, satisfy P, the following assume-guarantee rule is used:

$$\frac{\begin{array}{c}\langle A \rangle\ M_1\ \langle P \rangle \\ \langle true \rangle\ M_2\ \langle A \rangle\end{array}}{\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

The assumption A should be [19]:

1. Strong enough to constrain the behaviour of M_1 so that $\langle A \rangle\ M_1\ \langle P \rangle$ holds, and
2. Weak enough such that $\langle true \rangle\ M_2\ \langle A \rangle$ holds

L* Algorithm

The L* algorithm was developed by Augluin [4] and then further improved by Rives and Chapire [41]. It attempts to learn an unknown language $\mathcal{U}$, over an alphabet Σ by interacting with a *minimally adequate teacher* [19]. The language produces a finite state automaton $\mathcal{C}$ such that the language accepted by $\mathcal{C}$ is that of $\mathcal{U}$.

L* is based upon a model called "exact learning from membership and equivalence queries" [29]. In this model, the learner can ask the teacher two types of queries:

- **Membership queries** – Asks if a given string is in the unknown language
- **Equivalence queries** – Checks if a candidate DFA is identical to the unknown language.

Once the algorithm has constructed a table containing information about various strings belonging to the language (membership), it then constructs a candidate automaton, and asks the teacher if the conjecture is correct (equivalence)[2]. L* is used with assume-guarantee to build the assumption such that it meets the criteria specified in 2.4.2. Alur et al. [2] developed a symbolic implementation of L* in NuSMV.

Applying assume-guarantee to interpreted systems, and implementation in MCMAS, would allow for the individual verification of agents without verifying the whole state space. While this would be advantageous in principle, it has been shown that obvious system decompositions (i.e. the individual agents in interpreted systems) may not be the best, see [43].

[2]For further information on L* , see [29].

2.4.3 BMC with BDDs

In 2001, Fady Copty et al [20] 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[3] 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 ascertain if the *frontier* set (the current reach set) and the error set are disjoint. Given a bound k, their algorithm is as follows:

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

1: $\text{Frontier}_0 \leftarrow \mathbf{S}$
2: **for** $(i = 0;\ i < k;\ i{+}{+})$ **do**
3: **if** $(\text{Frontier}_i \cdot \mathbf{T} \neq \emptyset)$ **then**
4: **return** (FAILURE)
5: **end if**
6: $\text{Frontier}_{i+1} \leftarrow \text{IMG}(\mathbf{TR}, \text{Frontier}_i)$
7: **end for**
8: **return** (PASS)

Where IMG is the function to calculate the set of reachable states, from an originating set given the transition relation.

Among other topics Amal et al 2003 [3] 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 2002 [12]. They discuss the idea of not only *forward* bounded model checking – from the initial set to the target set (FWDBMC) – 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 [14]. Whilst only considering *safety* properties, their results seem to suggest that BDD-based BMC scales better with an increasing bound of k. These results were backed up by Amal et al 2003 [3], when they undertook a more through SAT vs BDD based test in which they also tested *liveness* properties.

[3] see [20] for details

3 Specification

3.1 Project Outline & Aims

The project idea discussed in Section 2.4.3 was one which was felt to be the most attainable in the timescale which this project has. It will also provide a chance to work on not only formal algorithms for model checking, but also on an implementation of this algorithm on the chosen model checker. The main aim of the project would be to develop a method for bounded model of the epistemic logic CTLK. Instead of creating a boolean formula equivalent to the model, and subsequently proving that it is satisfiable, the new implementation would look at creating the BDDs of the k-model, and then using existing model checking techniques to prove that the k-formula does or does not hold in the sub-model. The main *aim* of the project will be to develop an algorithm and auxillary methods for bounded model checking of CTLK, using BDDs rather than SAT. This project will attempt to amalgamate the bounded checking of multi-agent systems, and therefore epistemic logic, as presented in [40, 37], while attempting to use a bounded model checking with BDDs methods as discussed in [20, 12, 3]. The main *deliverable* will be an implementation in the MCMAS model checker of this algorithm from the previous section – **BMCMAS** (**B**ounded **M**odel **C**hecking **M**ulti-**A**gent **S**ystems).

3.2 Deliverables

The output from this project can be broken up into the following parts:

Develop an algorithm

I will have to develop an algorithm which will allow for the creation of BDDs from k-models based on interpreted systems, rather than propositional satisfiability methods. One possible way of achieving this would be to consider the methods for representing interpreted systems as boolean formulae, and then look at creating the frontier sets in an iterative manor towards an error state. This would be an adaption of the BOUNDEDTRAVERSAL method (Algorithm 11). Another way, although I have been unable to find any literature to suggest if this would even be possible, would be to consider the BMC algorithm (Algorithm 6), but lines `5-7` should be reconsidered as these deal with the generation of propositional formulae. One possible solution to these lines would be to create a symbolic representation of the k-model ($\mathcal{M}_k$), as per [40], and then a translation of the property φ we wish to check over $\mathcal{M}_k$, which could then be checked using an adaption of SAT_{CTLK} (Algorithm 7).

An algorithm implementation in MCMAS

Once a suitable algorithm has been devised, I will need to create an implementation of this algorithm within the MCMAS model checker, see Section 2.3.5. This implementation of the algorithm will entail a three step process:

1. Further investigation into MCMAS internals, to allow myself to understand and feel comfortable with developing and extending the current internals
2. Allowing for another model checking algorithm within the internals, including an argument to the interface to allow for the selection of this method[1]
3. Actual implementation of the algorithm from Section 3.2

It will be important not to break any of the existing functionality within MCMAS, as well as supporting all of the functionality available in the model checking approaches supported. This includes but is not limited to:

Counterexamples as GraphViz Dot files

One very beneficial feature is the creation of *graphical* counterexamples in the form of GraphViz Dot [54] files. I believe that every attempt should be made to continue to support this in a BMC with BDDs approach. Depending on how it is implemented within MCMAS, and its direct dependence upon CUDD's Dot file generation, this might be very simple to generate (i.e. generate the formula representing the counterexample, create a BDD from that, then let CUDD generate the Dot file).

Adapt the MCMAS Eclipse interface

As previously mentioned, MCMAS contains an Eclipse interface, which allows for the creation of ISPL files, as well as the graphical display of output from MCMAS. Once the Eclipse interface has been extended such that it allows for the selection of different model checking approaches, if the selection of BMC with BDDs generates a different than expected `stdout`, then the parsing should be adapted such that MCMAS interface understands the new output.

3.3 Extensions

If I am successful in completing all of the work from Section 3.2 which I have aimed to complete, then there are a number of extensions to the project are possible.

MCMAS Improvements

There are two main improvements which, even though they do not fall within the area of this project, I would consider developing to the MCMAS code base, given enough available time:

1. Currently, the only way for an application to interface with MCMAS is by executing MCMAS with a set of arguments, capturing the `stdout` from MCMAS, and then analysing this output. I feel that this is a very inelegant method. It would be nice to develop some form of API allowing for applications to call and run MCMAS without parsing `stdout`

[1] MCMAS already supports three approaches to model checking, so called "experiments", so this step might be a very simple one

themselves. Allowing for remote calls to a running MCMAS application would also be beneficial (i.e. there would be one instance of MCMAS, external application could call exported methods, with the method return values being sent over a message interface).

2. A build system. As previously stated, MCMAS contains only a single Makefile which has to be hand edited to set the location of the CUDD library. Even a simple `configure` script would alleviate this issue, and could also be extended to check for the existence of Flex and Bison.

Theoretical Proofs

An important process of developing an algorithm of this kind is proving that the algorithm developed is, in fact, correct. One crucial and primary step in proving this correctness is to prove the correctness of the translation of the formula over the k-model of $\mathcal{M}$.

Reductions in state space

Copty et al [20] discuss using Cone-of-influence, and Bounded Cone-of-influence, to eliminate variables which do not have any effect in the reachability of the *error* states. In their paper they only discussed *safety* properties, which can be easily represented as a set of *error* state. It would be interesting to consider the possibility of also using Cone-of-influence methods while looking at *liveness* properties. Cabodi et al [12] also discuss using "aggressive strategies to reduce the BDD size of the frontier sets". They look at optimising not only the traversal of the state space, but also the generation of the pre- and post- image sets.

SAT vs. BDDs – A BMC Perspective

As stated previously in Section 2.4.3, the gains provided by BMC could either come from the use of SAT over BDDs, or from bounded vs. unbounded model checking. It would be interesting to try and provide some form of perspective on this issue from a interpreted systems point of view. This would require the implementation of *another* model checking approach, in the form of BMC with SAT. This could be a similar implementation of the algorithm presented in [37], which would then allow for a direct comparison for the model checking gains of SAT vs BDDs.

An Interpreted Systems Knowledge Compilation Map

Creating a knowledge compilation map, as discussed in Section 2.4.1 for interpreted systems, would allow for the selection of particular NNF representation for a particular problem. To do this an approach to model checking interpreted systems based around a *target compilation* language such as dNNF (Definition 23) or Tree-of-BDDs (Section 2.4.1) would have to be developed, in either a bounded or an unbounded setting. This would then allow for a *partial* implementation of a knowledge compilation map for the selection between BDDs and the chosen target language.

3.4 Development Environment

Version Control

In a complex project such as the one discussed, it is important to use some form of version control system for the software being written. Version control systems allow for the logging of source code changes, roll-back in case an error has been introduced, as well as the easy synchronisation of source code across multiple development environments. For this project I have chosen to use Subversion [58].

Documentation

To allow for the maintainability of the changes which I make to the system, I will be required to document the code which I work on. One possible solution to this problem is to use a system which automatically extracts and generates documentation from comments written in a particular style within the code. Given that MCMAS is written in C++, I will consider using Doxygen [50] for this purpose.

Bug Tracking

A tool should be used to track all of the current and known bugs in a system to allow for their management and resolution. There exist several tools for this job such as BugZilla [48] or Trac [59], although a much simpler approach to this problem would be use a file documenting the bug and its associated line number.

3.5 Final Report

At the end of the project I will be required to document my project in the form of a final report. This document should examine the background of the area to which the project is involved, including a motivation for my project and a review of any previous work which may have attempted to tackle some of the problems my project attempts to overcome. The design decisions for the project should be presented, together with their justification.

A specification of the final project should be included, along with a detailed description of that implementation. It should also contain an evaluation of the final deliverables against the specified aims. This should be displayed through the use of benchmarks, examples and comparisons between what I have created and existing work in the field.

The report should also discuss any problems which faced my project from inception to completion, and how these hurdles were overcome. Areas for improvement which might exist in the project should be included, and further work which could be used to extend the work presented should be proposed.

4 Evaluation

The project's main aim is to create an algorithm which would provide model checking functionality for interpreted systems, and epistemic formulae upon these systems, which is based upon bounded model checking, and binary decision diagrams. In as such, this project will be deemed *successful* if an implementation of that algorithm functions as expected (i.e. it can model check ISPL files).

4.1 Verifying Correctness

Prior to implementing the devised algorithm, I should consider base- and edge- cases which would apply to my algorithm and see if the algorithm *should* still function as I expect. Without having already devised the algorithm, it is very hard to consider these cases.

Without creating some form of *formal* proof that the algorithm I have devised, "verifying" my implementation of the algorithm will be difficult. One of the simplest ways of attempting to prove any form of correctness would be to show if the model checker functions as *expected*.

There would be two ways in which I could show this:

1. The creation of a simple, and elementary, ISPL model in which, prior to running to code, I formally prove if the formula will hold or not, and manually calculate the expected generated counterexample. I could then compare this to the output of my implementation, and check the correctness of my implementation

2. By comparison between of the output, and results show in it, of my MCMAS implementation against that of the existing MCMAS implementation. This should be able to show me if formulae which are said hold in my model checker in fact do. This will not work for the generated counterexamples because bounded model checking creates the *minimum* possible counterexample path, whereas this may not be the case for unbounded model checking

4.2 Performance Benchmarks

As part of this project, I will attempt to evaluate the effectiveness of BMC with BDDs versus existing model checking methods for multi-agent systems. Below are a selection of problems (with the exception of one) which could be used in a performance benchmarking scenario to compare the existing MCMAS implementation with my implementation of bounded model checking using BDDs.

Bit Transmission Problem [24]

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*.

The above can easily be formalised in CTLK [34]:

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

It is quite easy to see how the bit transmission problem can be represented as a MAS (and as such, in ISPL). In this case, the environment models the possibly faulty channel, while the two agents are S and R.

Attacking Generals [24]

The attacking generals problem is that of a *coordination* problem. The problem is as follows: there are two divisions, each commanded by a separate general. If both attack simultaneously, they will win. If they do not, they will lose. As a result, neither general will attack without knowing if the other general will also attack. It takes an hour for a messenger to get from one general to another (with no assurance he will make it, he might get lost). How long does it take them to co-ordinate the attack? This problem can easily be modelled as a MAS using epistemic logic;

> *"No general will attack before it is **common knowledge** that they will both attack."* [37]

There are various variations upon this problem; [25] provides a detailed analysis.

Hongyang Qu [55] has developed a suite of C++ files to generate ISPL with a varying number of agents. These could be used to provide various benchmarks. Below are three "problems" that he has provided generators for:

Dining Cryptographers [13]

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 this cryptographer will announce the opposite to he sees. If an even number of "same" then 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. This can be represented by the MCMAS formula [23]:

```
((odd and !c1paid) -> (K(DinCrypt1,(c2paid or c3paid)))
and !K(DinCrypt1,c2paid) and !K(DinCrypt1,c3paid));
```

Muddy Children [24]

There are n children playing together and k of them have mud on their foreheads. All of the children can see each other, but they cannot communicate with each other, nor can they lie. The father now repeatedly asks the question "Does any of you know whether you have mud on your own forehead?" In all of the rounds below k, all the children answer "no", but in round k all the children with muddy foreheads answer "yes". Again, this problem can be modelled as an interpreted systems problem.

Card Game [33, 27]

In this game an agent plays against the environment in a simple card game. There exist just 3 cards in the deck: Ace, King and Queen. Ace wins over King, King over Queen, and Queen over Ace. In the first round, the player takes a card, as does the environment. In the second round, the player is able to change his card for another. The MAS is then modelled around the idea that from the initial state, the agent knows a strategy such that it can play in way that it knows will result in a win. This uses a form of logic called ATL, which will not be discussed here (see [1]).

Metrics

The bit transmission problem, attacking generals, dining cryptographers, and muddy children all provide interesting examples which could be used to compare the two versions of MCMAS. In this report I have not discussed ATL, which is required for the Card Game problem, and have not considered it in a bounded model checking scenario. This makes this problem unsuitable to use as a performance benchmark between the two programs.

To do a quantitative analysis between the two implementations, I will require a set of metrics which will allow for a direct comparison between both implementations, when running the same example problem. All of these metrics should be very easy to generate for both implementations. These metrics will only be comparable if the output from the model checker (except the counterexample, for reason stated previously) is the same (i.e. the evaluations of the formulae is the same).

For all of these metrics, *lower* is considered better:

- **CPU Time** – The time in seconds which it takes for the implementation to check, and output the results, on the chosen model
- **Number of states considered** – The k-model $\mathcal{M}_k$, which will be considered by the BMC algorithm, may have drastically less states than the full model $\mathcal{M}$
- **BDD Stats** – CUDD can generate BDD stats of the represented state space, some of the ones which *may* provide a useful comparison:

  - Number of BDD variables
  - Number of BDD nodes
  - Peak number of nodes
  - Number of nodes allocated
  - BDD memory use

- **Memory Usage** – The peak and average memory use, in MB of the model checker. This would be the additional memory, beyond the BDDs which each method consumed, used by each method; this could be considered as the memory overhead for each implementation

From these metrics, it would be interesting to consider scenarios which provide better results for different methods. For example, does a BMC method using BDDs outperform unbounded model checking with BDDs when a *longer* counterexample is generated? Or is the converse true (i.e. BMC with BDDs performs better for shorter counterexamples)? Penczek [38] shows that if $\mathcal{M}, s \vDash \varphi$, then $\mathcal{M}_k, s \vDash \varphi$ where the bound $k = |\varphi|$. From this, it might be worth considering that generating formulae with a larger $|\varphi|$ may have longer counterexamples.

It might also be worth considering that the number of agents in a system might effect which method is superior. Using Hongyang's suite for generating problems with an arbitrary number of agents would be an interesting way to test this, comparing the two implementations with an increasing number of agents. Generating larger and larger models in this way, is a possible method of trying to find circumstances under which one method fails to check the model in a time constraint, while the other method still completes.

Amla et al [3] look at comparing SAT based BMC against BDD based BMC. For this they consider the benchmarks with the following characteristics:

- Where deep counterexamples were required
- Where BDDs performed poorly, and others where BDDs performed well
- Where the number of state variables in the designs varied from 11 up to 1273
- Ones which were based on real software models, and not toy examples
- Where both safety and liveness properties needed to be checked

I believe that it would be highly beneficial to equally consider these charaterstics in the selection of the benchmarks I choose.

4.3 Conclusion

The project's success should be determined by the performance of the solution, based on a large selection of benchmarks. This performance does not have to be an *increase*, but enough to show under what circumstances the solution should, or shouldn't, be adopted. The performance comparison might provide an analysis which advocates the benefits of one model checking method over another.

Bibliography

[1] **Rajeev Alur**, **Thomas A. Henzinger** and **Orna Kupferman**. Alternating-time Temporal Logic. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*. IEEE Computer Society, Washington, DC, USA. ISBN 0-8186-8197-7, 1997 p. 100.

[2] **Rajeev Alur**, **P. Madhusudan** and **Wonhong Nam**. Symbolic Compositional Verification by Learning Assumptions. In *Computer Aided Verification, LNCS 3576*. 2005 pp. 548–562.

[3] **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/.

[4] **D Angluin**. Learning regular sets from queries and counterexamples. In *Information and Computation* (1987)(75).

[5] **Christel Baier** and **Joost-Pieter Katoen**. *Principles of Model Checking*. The MIT Press, May 2008. ISBN 026202649X.

[6] **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.

[7] **Sergey Berezin**, **Sergio Campos** and **Edmund M. Clarke**. Compositional Reasoning in Model Checking. In *Compositionality: The Significant Difference, LNCS 1536*. 1998 pp. 81–102.

[8] **A. Biere**, **A. Cimatti**, **E. Clarke**, **O. Strichman** and **Y. Zhu**. Bounded Model Checking. In *Advances in Computers* volume 58(2003).

[9] **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.

[10] **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/.

[11] **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.

[12] **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.

[13] **David Chaum**. The dining cryptographers problem:Unconditional sender and recipient untraceability. In *Journal of Cryptology* volume 1(1988)(1):pp. 65–75. ISSN 0933-2790 (Print) 1432-1378 (Online). doi:10.1007/BF00206326.

[14] **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.

[15] **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.

[16] **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/.

[17] **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.

[18] **Edmund M. Clarke**, **Orna Grumberg** and **Doron Peled**. *Model Checking*. MIT Press, 1999.

[19] **Jamieson M. Cobleigh**, **George S. Avrunin** and **Lori A. Clarke**. Breaking Up is Hard to Do: An Evaluation of Automated Assume-Guarantee Reasoning. In *ISSTA*. 2006 .

[20] **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/.

[21] **Adnan Darwiche**. Compiling knowledge into decomposable negation normal form. In *proceedings International Joint Conference on Artificial Intelligence, IJCAI'99*. Morgan Kaufmann, 1999 pp. 284–289.

[22] **Adnan Darwiche** and **Pierre Marquis**. A Knowledge Compilation Map. In *Journal of Artificial Intelligence Research* volume 17(2002):pp. 229–264.

[23] **Hongyang Qu F. Raimondi, A. Lomuscio**. MCMAS v0.9.6: User Manual. URL http://dfn.dl.sourceforge.net/sourceforge/ist-contract/mcmas-0.9.6.2.tar.gz.

[24] **Ronald Fagin**, **Joseph Y. Halpern**, **Yoram Moses** and **Moshe Y. Vardi**. *Reasoning About Knowledge*. MIT Press, 1995.

[25] **Joseph Y. Halpern** and **Yoram Moses**. Knowledge and common knowledge in a distributed environment. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA. ISBN 0-89791-143-1, 1984 pp. 50–61. doi:http://doi.acm.org/10.1145/800222.806735.

[26] **Michael Huth** and **Mark Ryan**. *Logic in Computer Science: modelling and reasoning about systems (second edition)*. Cambridge University Press, 2004. ISBN 052154310X.

[27] **W. J. Jamroga**. *Using Multiple Models of Reality. On Agents who Know how to Play Safer*. Ph.D. thesis, University of Twente, Enschede, July 2004.

[28] **Joost-Pieter Katoen**. Concepts, Alogirthms and Tools for Model Checking, Semester 1998–1999.

[29] **M. J. Kearns** and **U. V. Vazirani**. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, 1994.

[30] **Saul Kripke**. Semantical Considerations on Modal Logic. In *In Proceedings A Colloquium on Modal and Many-Valued Logics, Helsinki*. 1962 .

[31] **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.

[32] **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/.

[33] **Alessio Lomuscio** and **Franco Raimondi**. Model checking knowledge, strategies, and games in multi-agent systems. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, New York, NY, USA. ISBN 1-59593-303-4, 2006 pp. 161–168. doi:http://doi.acm.org/10.1145/1160633.1160660.

[34] **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.

[35] **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.

[36] **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.

[37] **Wojciech Penczek** and **Alessio Lomuscio**. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Fundam. Inf.* volume 55(2002)(2):pp. 167–185. ISSN 0169-2968.

[38] **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.

[39] **Franco Raimondi**. Notes from the course, GS03/4203: Verification and validation, 2007 – 2008. URL http://www.cs.ucl.ac.uk/staff/F.Raimondi/teaching/.

[40] **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.

[41] **R L Rivest** and **R E Schapire**. Inference of finite automata using homing sequences. In *Information and Computation* (1993).

[42] **Marek Sergot**. Notes from the course, 499: Modal and temporal logic, Autumn 2008. URL http://www.doc.ic.ac.uk/~mjs/teaching/499.html.

[43] **Nishant Sinha** and **Edmund M. Clarke**. SAT-Based Compositional Verification Using Lazy Learning. In *CAV* (editors **Werner Damm** and **Holger Hermanns**), volume 4590 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-540-73367-6, 2007 pp. 39–54. URL http://dblp.uni-trier.de/db/conf/cav/cav2007.html#SinhaC07.

[44] **Sathiamoorthy Subbarayan** and **Henrik Reif Andersen**. Integrating a Variable Ordering Heuristic with BDDs and CSP Decomposition Techniques for Interactive Configurators.

[45] **Sathiamoorthy Subbarayan**, **Lucas Bordeaux** and **Youssef Hamadi**. Knowledge Compilation Properties of Tree-of-BDDs, 2007.

[46] **Francesca Toni**. Notes from the course, 474: Multi-agent Systems, Autumn 2008. URL http://www.doc.ic.ac.uk/~ft/teaching.html.

[47] **Michael J. Wooldridge**. *Reasoning about Rational Agents*. MIT Press, 2000.

Web References

[48] Bugzilla. http://www.bugzilla.org/.

[49] CUDD: Colorado University Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/.

[50] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[51] Eclipse. http://www.eclipse.org/.

[52] Flex. http://flex.sourceforge.net/.

[53] GNU Bison. http://www.gnu.org/software/bison/.

[54] Graphviz. http://www.graphviz.org/.

[55] Hongyang Qu : Tools. http://www.doc.ic.ac.uk/~hongyang/Tools.html.

[56] MCMAS. http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/.

[57] MCMAS 0.9.6.2. http://dfn.dl.sourceforge.net/sourceforge/ist-contract/mcmas-0.9.6.2.tar.gz.

[58] Subversion. http://subversion.tigris.org/.

[59] Trac. http://trac.edgewall.org/.